Julius–Maximilians–Universität Würzburg
Institut für Informatik
Lehrstuhl für Kommunikationsnetze (Informatik III)

Deep Reinforcement Learning for Configuration of Time-Sensitive-Networking

Bachelor's Thesis submitted by

Jan Hofmann

Julius–Maximilians–Universität Würzburg
Institut für Informatik
Lehrstuhl für Kommunikationsnetze (Informatik III)

Deep Reinforcement Learning for Configuration of Time-Sensitive-Networking

Bachelorarbeit im Fach Informatik
vorgelegt von

Jan Hofmann

Angefertigt am
Lehrstuhl für Kommunikationsnetze (Informatik III)
Julius–Maximilians–Universität Würzburg

Betreuer:
Prof. Dr. Tobias Hoßfeld
Dr. rer. nat. Michael Seufert
Alexej Grigorjew M. Sc.
Nikolas Wehner M. Sc.

Abgabe der Arbeit:
27.07.2020

Deutsche Zusammenfassung

Zuverlässige Echtzeitnetzwerke spielen eine zentrale Rolle im heutigen industriellen Umfeld. Während sich in anderen Anwendungsbereichen Ethernet als Technik für Kommunikationsnetze durchsetzen konnte, basiert industrielle Kommunikation bis heute häufig noch auf teuren Feldbus-Systemen. Mit der Einführung von Time-Sensitive-Networking (TSN) wurde Ethernet schließlich um eine Reihe von Standards erweitert, die die hohen Anforderungen an Echtzeitkommunikation erfüllen und Ethernet damit auch im industriellen Umfeld etablieren sollen. Doch für eine zuverlässige Kommunikation, besonders im Hinblick auf die Übertragungsverzögerung von Datenpaketen (Latenz), ist die richtige Konfiguration von TSN entscheidend.

Dynamische Netzwerke zu konfigurieren ist ein Optimierungsproblem, das verschiedene Herausforderungen birgt. Verfahren wie die lineare Optimierung liefern zwar optimale Ergebnisse, jedoch steigt der Zeitaufwand exponentiell mit der Größe der Netzwerke. Moderne Lösungsansätze wie Machine Learning (ML) können sich einer optimalen Lösung annähern, benötigen jedoch üblicherweise große Datenmengen, auf denen sie trainiert werden (Supervised Learning).

Diese Arbeit untersucht die Anwendung von Deep Reinforcement Learning (DRL) zur Konfiguration von TSN. DRL kombiniert Reinforcement Learning (RL), also das selbstständige Lernen ausschließlich durch Interaktion, mit dem Deep Learning (DL), dem Lernen mittels tiefer neuronaler Netze. Die Arbeit beschreibt, wie sich eine Umgebung für DRL zur Simulation und Konfiguration von industriellen Netzwerken implementieren lässt, und untersucht die Anwendung zweier unterschiedlicher Ansätze von DRL auf das Problem der TSN-Konfiguration.

Beide Methoden wurden anhand von zwei unterschiedlich komplexen Datensätzen ausgewertet und die Ergebnisse sowohl mit den zeitaufwändig generierten Optimallösungen als auch mit den Ergebnissen zweier Supervised Learning-Ansätze verglichen. Es konnte gezeigt werden, dass DRL optimale Ergebnisse auf kleinen Netzwerken erzielen kann und insgesamt in der Lage ist, Supervised Learning bei der Konfiguration von TSN zu übertreffen. Weiterhin konnte in der Arbeit demonstriert werden, dass sich DRL schnell an fundamentale Veränderungen der Umgebung anpassen kann, was mit Supervised Learning nur durch deutlichen Mehraufwand möglich ist.

Contents

1
Introduction

Reliable, deterministic real-time communication is fundamental to most industrial systems today. In many other domains Ethernet has become the most common platform for communication networks, but has been unsuitable to satisfy the requirements of industrial networks for a long time. This has changed with the introduction of Time-Sensitive-Networking (TSN), a set of standards utilizing Ethernet to implement deterministic real-time networks. This makes Ethernet a viable alternative to the expensive fieldbus systems commonly used in industrial environments. However, TSN is not a silver bullet. Industrial networks are a complex and highly dynamic environment and the configuration of TSN, especially with respect to latency, is a challenging but crucial task.

Various approaches have been pursued for the configuration of TSN in dynamic industrial environments. Optimization techniques like Linear Programming (LP) are able to determine an optimal configuration for a given network, but the time consumption exponentially increases with the complexity of the environment. Machine Learning (ML) has become widely popular in the last years and is able to approximate a near-optimal TSN configuration for networks of different complexity. Yet, ML models are usually trained in a supervised manner which requires large amounts of data that have to be generated for the specific environment. Therefore, supervised methods are not scalable and do not adapt to changing dynamics of the network environment.

To address these issues, this work proposes a Deep Reinforcement Learning (DRL) approach to the configuration of TSN in industrial networks. DRL combines two different disciplines, Deep Learning (DL) and Reinforcement Learning (RL), and has gained considerable traction in the last years due to breakthroughs in various domains (Mnih et al. 2013, Silver et al. 2016). RL is supposed to autonomously learn a challenging task like the configuration of TSN without requiring any training data. The addition of DL allows to apply well-studied RL methods to a complex environment such as dynamic industrial networks.

There are two major contributions made in this work. In the first step, an interactive environment is proposed which allows for the simulation and configuration of industrial networks using basic TSN mechanisms. The environment provides an interface that allows to apply various DRL methods to the problem of TSN configuration. The second contribution of this work is an in-depth study on the application of two fundamentally different DRL methods to the proposed environment. Both methods are evaluated on networks of different complexity and the results are compared to the ground truth and to the results of two supervised ML approaches. Ultimately, this work investigates if DRL can adapt to changing dynamics of the environment in a more scalable manner than supervised methods.

The remainder of this work is structured as follows. Chapter 2 gives an introduction to the key components of industrial networks and TSN, introduces the functionality of DL, and covers the foundations of RL along with the two DRL methods that are studied in this work. The chapter concludes with an overview over related RL methods that have already been applied to the domain of dynamic networking. Chapter 3 proposes the RL environment for simulation and configuration of TSN and describes the specific implementation details of the two DRL methods used in this work. Chapter 4 covers the training and evaluation of the two methods on networks of different complexity. The chapter presents the results on two different data sets and compares the results to the ground truth and the results of two supervised ML methods. Ultimately, the scalability of the DRL approach is investigated by changing the dynamics of the proposed environment. Chapter 5 concludes with a brief summary and an outlook on possible further enhancements to the methods presented in this work.

2
Background

This chapter gives an overview of the key technologies used in industrial networks and introduces Time-Sensitive-Networking as a set of standards to implement deterministic real-time communication using Ethernet. This is followed by an insight into Deep Learning and the functionality of Deep Neural Networks as powerful computational models. Then, the chapter describes the foundations of Reinforcement Learning and the value-based Q-Learning algorithm which serves as a basis for the Deep Q-Network method. This is followed by the Policy Gradient algorithm as a policy-based learning approach which serves as a basis for the Actor-Critic method. The chapter concludes with a brief overview of related problems in the domain of dynamic networking where Reinforcement Learning methods have already been successfully applied.

2.1 Time-Sensitive-Networking

This work studies Ethernet-based networks in an industrial environment. Ethernet is the most common type of wired local network and was originally developed as a platform for providing best-effort services, i.e., services that do not meet real-time requirements. Since then, Ethernet has been constantly evolving and has become practicable for providing deterministic real-time services in industrial environments.

Communication in Ethernet-based networks is divided into seven layers, specified in the Open System Interconnection (OSI) model. The OSI model provides layers of abstraction and reduces the complexity of network communication. Each layer serves the next layer and uses the services of the previous layer.

In this work, only the physical layer and the data link layer (cf. IEEE 802.3 2018) are of interest. The physical layer ensures the transmission of raw bits between different systems using a physical transmission medium. The data link layer, on the other hand, is responsible for the reliable communication between two systems.

In order to directly communicate with a specific system, a media access control (MAC) address is used. This address is usually assigned by the manufacturer and uniquely identifies a network interface of a system.

Besides correct addressing, reliable communication also requires identifying transmission errors and controlling the flow of data. In order to provide such functionality, the data link layer encapsulates raw data into units called *frames*. An Ethernet frame includes the actual payload along with additional meta information that is necessary to ensure a correct transmission to the destination system (cf. Figure 2.1).

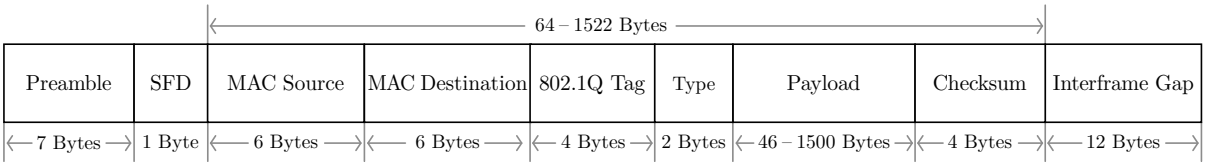

Figure 2.1. Structure of an Ethernet frame as specified in (IEEE 802.3 2018).

An Ethernet frame has a size of at least 64 bytes up to a maximum of 1522 bytes and includes the following additional meta information:

- **Preamble and SFD**. Each frame is preceded by a preamble, which is a well-defined pattern of 7 bytes for the purpose of synchronization with the receiver of the frame. The Start Frame Delimiter (SFD) is a single byte which indicates the start of the frame.
- **MAC Source and Destination**. Both addresses uniquely identify the sender of the frame and the destination system, respectively.
- **802.1Q Tag**. Includes the Class of Service (CoS), which can be used to assign different priorities to the frames.
- **EtherType**. Indicates the protocol that the next layer utilizes for processing the payload of the frame.
- **Checksum**. The checksum allows for identification of bit transmission errors.
- **Interframe Gap**. After transmitting a frame, there is a gap of 12 bytes before the transmission of the next frame.

This work considers industrial networks that are build from two basic components, bridges and endpoints. Both connect to physical links and exchange streams of data frames at different rates. This work uses the commercial term *switch* synonymously for a bridge. A switch connects multiple networks or network components and allows segmentation of the network. The switch receives frames on an incoming port and forwards them to the correct outgoing port, based on the destination MAC address.

For this purpose, the switch learns the MAC address of all network participants and stores them in a MAC address table. If the destination address is not known to the switch yet, the frame is flooded to all outgoing ports.

Endpoints, on the other hand, are components that send and receive frames. In the context of this work, there are two types of endpoints to distinguish between. Sensors and actuators, hereafter referred to as *sensors*, send and receive data and are able to detect changes in the industrial environment and interact with it. Programmable logic controllers (PLCs), in this work referred to as *controllers*, are devices that are equipped with microprocessors and are able to control and monitor multiple sensors. Controllers are programmed in advance to fulfil specific tasks and communicate with sensors over bidirectional streams.

There are one or multiple switches on the path of each stream, forwarding frames to the destination endpoint. Each switch has multiple incoming ports, which means that multiple frames can arrive at the same time. Therefore, one of the most important tasks of a switch is to queue and schedule incoming frames, i.e., select the next frame for transmission. Traffic scheduling, also referred to as *transmission selection*, is not to be confused with traffic shaping, a processing step which delays certain frames in order to optimize the total network utilization or increase bandwidth for other streams.

When it comes to industrial environments, communication often has to meet application-specific real-time requirements with regards to the network latency. Latency denotes the total transmission delay of a frame from source to destination when being transmitted over physical links and being forwarded through switches. Traffic with bounded end-to-end latency, hereafter referred to as *time-sensitive traffic*, and traffic without such requirements, referred to as *best-effort traffic*, usually coexist within the same network.

In order to meet real-time requirements and ensure a deterministic transmission behaviour for time-sensitive traffic, the IEEE 802.1 working group proposed a set of standards under the term Time-Sensitive-Networking (TSN, IEEE 802.1Q 2018). TSN provides four key components (cf. Figure 2.2) that allow to implement deterministic real-time services suited for industrial use, based on Ethernet.

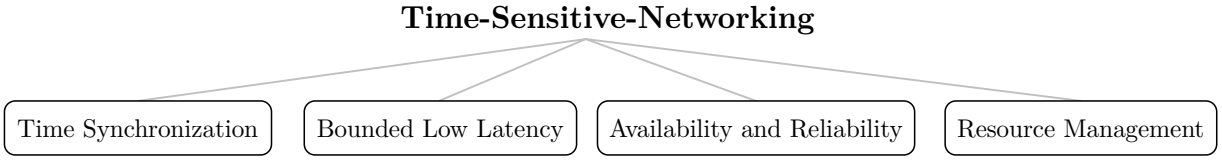

Figure 2.2. Categorization of TSN into four key components (cf. Farkas 2018).

Time synchronization. TSN utilizes the Precision Time Protocol (PTP, IEEE 1588 2008) which distributes time information across all network participants. This is a prerequisite for mechanisms like the Time Aware Shaper (TAS, IEEE 802.1Qbv 2016) which introduces timed gates to each queue of a switch port.

Bounded Low Latency. In order to guarantee bounded end-to-end latency, TSN proposes various traffic shaping mechanisms besides TAS. The Credit-based Shaper (CBS, IEEE 802.1Qav 2009) smooths out bursting traffic and ensures that services do not exceed bandwidth. The Asynchronous Traffic Shaper (ATS, IEEE 802.1Qcr 2020, Specht et al. 2016) improves link utilization, especially for mixed traffic types. Unlike CBS and TAS, the ATS mechanism does not require the network-wide distribution of time information but works with bridge-local information and therefore reduces implementation complexity.

Besides traffic shaping, TSN also includes a more simple transmission selection method, the Strict Priority Transmission Selection (SP, IEEE 802.1Q 2018). Each switch manages multiple queues dedicated to frames of different priority. (IEEE 802.1Q 2018) proposes up to eight classes that can be used to prioritize the network traffic. Frames in different queues are strictly forwarded by priority. Within a single priority queue, the switch follows the first-in-first-out (FIFO) principle.

SP can be configured regarding the guaranteed maximum per-hop latency that each of the priority classes provide. The mechanism works with bridge-local information and therefore does not require time synchronization, and in contrast to ATS, it is widely supported by current hardware.

High Availability and Reliability. TSN provides mechanisms to ensure reliability and fault tolerance of the network. This includes redundancy mechanisms (IEEE 802.1CB 2017) as well as filtering and policing (IEEE 802.1Qci 2017), which protects against excessive bursts and bandwidth usage.

Resource Management. TSN includes protocols like the Stream Reservation Protocol (SRP, IEEE 802.1Q 2018) and the Resource Allocation Protocol (RAP, IEEE 802.1Qdd 2018), which specify admission control, i.e., ensure the availability and reservation of required network resources. RAP can be used to include bridge-local information in the reservation process, which is applicable to SP transmission selection.

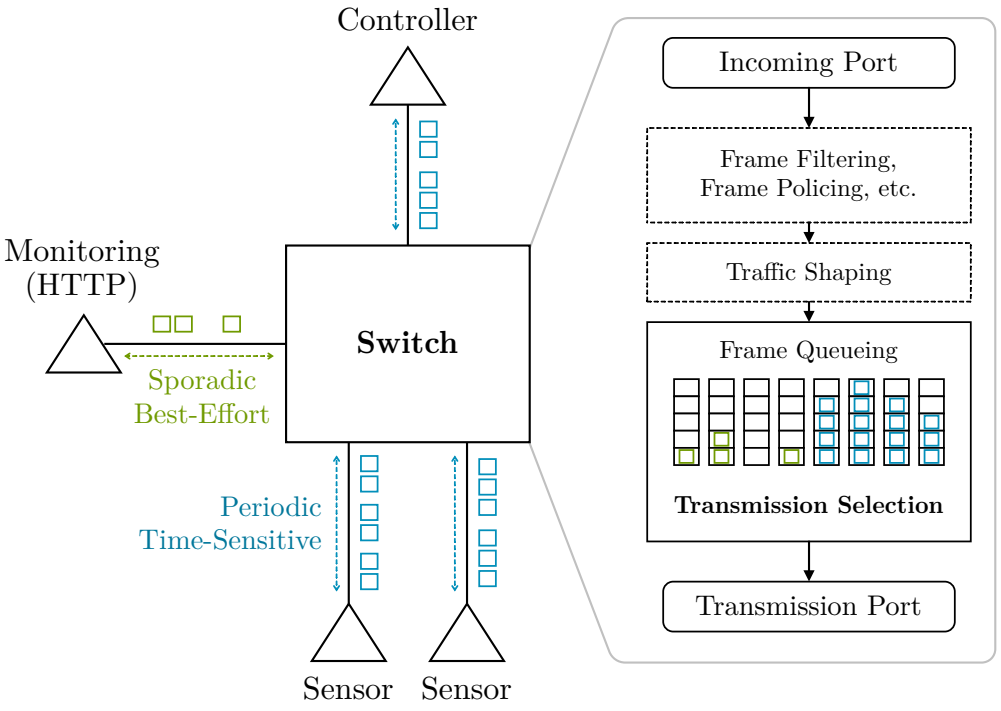

Figure 2.3. Frame processing steps of a switch in a network with mixed traffic types. An exemplary source of best-effort traffic is a monitoring dashboard which is accessed via a browser using HTTP.

In summary, Figure 2.3 shows a schematic representation of a network with sporadic best-effort traffic and periodic time-sensitive traffic. Sensors and controllers periodically send and receive frames with end-to-end latency requirements. Switches perform various processing steps on each frame before forwarding it to its destination. Besides filtering, policing and traffic shaping, the task this work focuses on is the transmission selection.

More precisely, networks studied in this work do not rely on traffic shaping mechanisms but rather utilize SP transmission selection only. Of the eight available priority classes (cf. IEEE 802.1Q 2018), this work utilizes up to four classes for time-sensitive traffic and the other classes for best-effort traffic. Given a network, the objective is to identity an optimal SP configuration that maximizes the number of streams which can be added to the network without violating latency requirements. This work investigates the application of Machine Learning techniques to this configuration optimization problem.

2.2 Deep Learning

Machine Learning (ML) is a major subset of Artificial Intelligence (AI) and focuses on algorithms that use data to automatically improve themselves over time. The goal of ML is to learn a computational model that can perform some task without explicitly being programmed to do so. This primarily applies to complex problems that cannot be solved by some algorithm. The basic concepts described in this section are covered in great detail by (Goodfellow et al. 2016).

In general terms, the goal of ML is to learn some unknown and in most cases highly complex function $f : \mathcal{X} \to \mathcal{Y}$, where $x \in \mathcal{X}$ is some input and $f(x) \in \mathcal{Y}$ is the calculated output. The input can be anything from an image to a series of measurement data and the output can be anything from a single binary number to a probability distribution.

One key concern with traditional ML techniques is their limited capability in processing raw data. Extracting the most important features from data is not a trivial task and therefore requires both domain expertise and careful engineering.

Deep Learning (DL) is a ML technique that utilizes Deep Neural Networks (DNN) as a computational model for learning multiple levels of representation from raw data, thus extracting the key features. A DNN is composed of multiple layers of simple processing units, called *neurons*, that are connected to neurons of the previous and next layer. Each connection is weighted and represents an adjustable parameter within the network. While the first and last layer of the network correspond to the input and output, layers in between are called *hidden layers*. The NN is considered *deep* if it has at least one hidden layer, as illustrated in Figure 2.4.

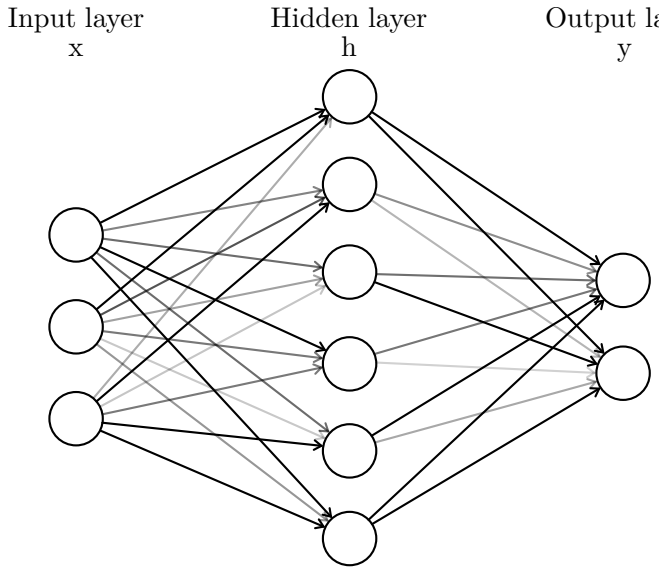

Figure 2.4. DNN architecture with one hidden layer and weighted connections between the neurons, which is indicated by different levels of opacity.

Each non-input neuron in the network computes a simple vector-to-scalar function that combines multiple input values and produces a single output value, which is then fed to the next layer of neurons.

DNNs are more powerful than NNs without a hidden layer. The reason is the non-linearity, which is introduced by applying an activation function to the hidden layer. This ensures that the output cannot be reproduced from a linear combination of the inputs and allows to learn much more complicated functions. The most commonly used activation function is the rectified linear unit (ReLU), which is defined in Equation 2.1 with x being the input to a neuron.

$$f(x) = \max(0, x) \tag{2.1}$$

Using ReLU as an activation function, a non-input network layer performs a non-linear transformation $h = f(a)$ with $a = W \cdot x + b$. x is a vector of input values, W is a matrix of weights for every connection between the previous layer and the current layer, and b denotes the bias.

In order for the DNN to actually learn an unknown function, it is trained on a number of existing input-output pairs. As the desired output is already known, such data is considered *labelled* and the method of training is generally known as *supervised learning*. The DNN learns an unknown function by example and generalizes in order to predict output values from input values it was not trained.

For the purpose of learning, a loss function is defined which calculates the distance between a predicted output value and the desired output. A commonly used loss function is the mean squared error (MSE), which is defined in Equation 2.2 with $\hat{y}$ being the desired value and y being the actual prediction.

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.2}$$

Throughout the training, the DNN gradually adjusts its weight parameters in order to minimize the loss and increase the accuracy of the prediction. The adjustment is based on a method called *gradient descent*. It computes a gradient based on the loss with respect to the network parameters, which is then propagated back through the network. This adjusts the parameters in a direction which ultimately minimizes the loss (Rumelhart et al. 1988).

Derived from this computational method, several types of DNNs have been introduced, specialized on solving different types of problems. Convolutional Neural Networks (CNNs) have been proposed by (LeCun et al. 1995) in order to process images by utilizing convolutional layers which learn filters with small receptive fields that are able to detect features like edges and shapes in images. Recurrent Neural Networks (RNNs, Jordan 1986) use internal memory to process input sequences with variable length. Based on this concept, (Hochreiter et al. 1997) introduced Long Short-Term Memory (LSTM) which allows for processing long sequential input data.

DNNs have proven to be powerful computational models with the capability to generalize and process large amounts of various raw data, which does not require much engineering by hand. Along with decreasing computational cost and continuous technological improvements, there is a wide range of domains where DNNs have become state-of-the-art and led to significant breakthroughs. This includes speech recognition (Hinton et al. 2012), image recognition (Krizhevsky et al. 2012), and machine translation (Wu et al. 2016).

2.3 Deep Reinforcement Learning

Reinforcement Learning (RL) is another major subset of AI that does not deal with learning from pre-labelled data but rather with sequential decision-making and interaction with an environment. This section introduces the most important concepts of RL as covered in (Sutton et al. 2018, Part I).

The key component of RL is an agent which, at each time step t, interacts with its environment by taking an action $a_t \in \mathcal{A}$. The agent then observes a transition from the current state $s_t \in \mathcal{S}$ to a consecutive state $s_{t+1} \in \mathcal{S}$ and a reward signal r_t for its action (cf. Figure 2.5). Rewards may be positive or negative and correspond to the labels used in supervised learning. The difference is that the reward signal for an action is not known from the start and has to be discovered by trial and error. After an episode of multiple actions, the agent ultimately stops when reaching a terminal state.

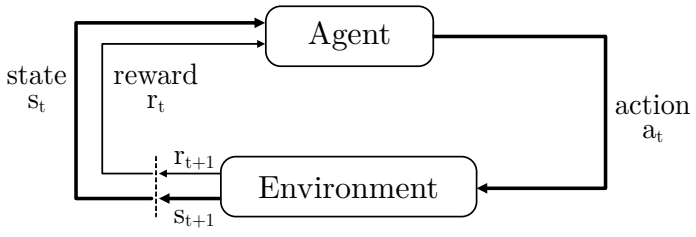

Figure 2.5. Agent interacting with its environment.

Furthermore, the agent initially does not know how an action will affect the current state. This makes RL a form of *unsupervised learning*, where the agent requires numerous interactions with its environment to actually acquire experience in which action to take being in a specific state. For this, the agent generally does not require any knowledge of the environment in advance.

The agent is supposed to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$, which is a function that maps a state to the action, the agent should take when in this state. The goal of RL is to learn an optimal policy π^* which maximizes the cumulative reward the agent receives with respect to some objective.

One may differentiate between two types of environments, stochastic and deterministic. In a stochastic environment, the agent does only have limited impact to the outcome of an action. Taking the same action in the same state at different times may result in different observations. This is often true when modelling real-world scenarios that involve some stochastic elements that cannot be fully known in advance. This is further discussed in (Kuang et al. 2019).

In a deterministic environment, on the other hand, the transition to the next state depends solely on the current state and the action the agent takes. In this work, a deterministic environment is proposed where the same choice of action in a state always leads to the same observation.

The problem of learning from interaction with an environment in order to achieve some goal can be formulated as a Markov Decision Process (MDP, Bellman 1957). In the context of this work, only the deterministic case is considered and the MDP is defined as a 4-tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where

- $\mathcal{S}$ is a state space,
- $\mathcal{A}$ is a finite action space,
- $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a transition function that maps a state-action pair to a consecutive state,
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a reward function.

The MPD allows to model decision making with the primary goal of maximizing the cumulative reward. The most important property of MPDs is the *Markov property*, which claims that the impact of an action solely depends on the current state and not on past decisions. The current state always includes all relevant information required for the transition to a consecutive state. This property is a fundamental prerequisite for the RL methods covered in the following sections.

This work studies methods where the agent learns in an *online* setting. This means that labelled data become available successively in form of observations when the agent interacts with its environment. This is opposed to an *offline* setting where labelled data are available from the start. An important characteristic of an online setting is the fact that the agent can actually decide how to gather new data, e.g., by randomly exploring different states instead of solely following its current policy.

Usually, it is desired to give the agent an incentive to reach its goal in as little time as possible. Therefore, a discount factor $\gamma \in [0, 1]$ is introduced which discounts future rewards with every time step. At time step t, the discounted reward for taking an action a_t is defined as $r_t = \gamma^t R(s_t, a_t)$. Discounting future rewards is an effective way to affect the behaviour of the agent.

- $\gamma = 1$ gives the agent no incentive to solve the MDP in reasonable time because the number of steps taken does not affect the future reward,
- $\gamma = 0$ lets the agent thoughtlessly maximize the next reward because any future reward becomes zero.

A second crucial hyperparameter is the learning rate $\alpha \in [0, 1]$. This determines to what extent the agent replaces previous knowledge with new experience.

- $\alpha = 0$ prevents the agent from acquiring any new knowledge,
- $\alpha = 1$ forces the agent to overwrite previous knowledge at each time step.

The right choice for γ and α depends on several factors including the characteristics of the environment and the utilized RL method.

2.3.1 Q-Learning

Q-Learning (Watkins et al. 1992) is one of the best studied algorithms that solves an MDP. The algorithm allows an agent to learn an optimal policy without prior knowledge of the environment, the reward function, or the transition function. With Q-Learning, the agent does not learn a model of its environment. Hence, the algorithm falls under the category of *model-free* learning.

Instead, the agent learns an action-value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that denotes the quality of an action in a given state. This quality value $Q^\pi(s_t, a_t)$ with respect to a policy π is equivalent to the maximum expected cumulative reward when taking action a_t in state s_t and following π thereafter.

At the beginning of the training, the Q-values for every state-action pair are arbitrarily initialized and stored in a table. At each time step t, the agent chooses an action a_t in the current state s_t and observes some consecutive state s_{t+1} along with a reward signal r_t. It then uses this observation to update its current Q-value for the just taken action a_t according to Equation 2.3.

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha(\underbrace{r_t + \gamma \max_{a \in \mathcal{A}} Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t)}_{\textit{temporal difference value}}) \tag{2.3}$$

$Q^\pi(s_t, a_t)$ denotes the current value while the term $r_t + \gamma \max_{a \in \mathcal{A}} Q^\pi(s_{t+1}, a)$ corresponds to the *target value*. The learning rate α specifies to which extent the old value is overwritten by the target value while γ discounts the maximum expected future reward for the next state.

Q-Learning is considered an *off-policy* algorithm. The reason is that the algorithm only learns the value function Q but does not explicitly learn a policy π. Instead, in Equation 2.4, the policy is derived from Q by greedily choosing the action with the highest Q-value in a given state.

$$\pi(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}(Q^\pi(s, a)) \tag{2.4}$$

The overall goal is to learn an optimal Q-function Q^* from which an optimal policy π^* can be derived which maximizes the cumulative reward.

Arbitrarily initializing the Q-values can lead to undesirable behaviour, because in a given state, the agent will always greedily choose the action based on its highest Q-value even if another action would lead to a higher cumulative reward in the long run. This raises the question if, especially in early phases of the training, the agent should trust its current Q-values or instead explore some other actions that could potentially lead to higher future reward.

For this purpose, an exploration rate $\varepsilon \in [0, 1]$ is introduced. ε is used as a probability for the agent not to greedily choose its next action by the highest Q-value but rather explore a completely random action. This strategy, called *ε-greedy*, is a trade-off between exploration and exploitation. In order to obtain high rewards, the agent should follow a behaviour that has proven beneficial in the past, which is termed *exploitation*. But, especially in early phases of the training, such actions must first be discovered by *exploration*. ε is an essential hyperparameter when utilizing the Q-Learning algorithm.

- $\varepsilon = 0$ lets the agent greedily choose the action with the highest Q-value every time, which can lead to a suboptimal policy,
- $\varepsilon = 1$ lets the agent choose a random action every time, which prevents the agent from learning a reasonable policy.

(White et al. 1992) were the first to thoroughly discuss the trade-off between exploration and exploitation. As with all hyperparameters, choosing an appropriate exploration rate ε depends on the dynamics of the environment.

Algorithm 1 shows the tabular Q-Learning algorithm with ε-greedy strategy according to (Watkins et al. 1992 and Sutton et al. 2018).

Algorithm 1: Q-Learning

Input: learning rate $\alpha \in [0, 1]$, exploration rate $\varepsilon \in [0, 1]$
Initialize $Q(s, a)$ arbitrarily $\forall s \in \mathcal{S}, a \in \mathcal{A}$ except that $Q(\text{terminal}, *) = 0$;
foreach *episode* **do**
  Start in state s_t;
  while s_t *is not terminal* **do**
    Choose action a_t randomly with ε probability, else $a_t = \pi(s_t)$;
    Take action a_t;
    Observe next state s_{t+1} and reward r_t;
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t))$;
    $s_t \leftarrow s_{t+1}$;
  end
end
Output: Q

2.3.2 Deep Q-Network

When modelling real-world scenarios, there often is a large or even infinite number of possible states. The tabular Q-Learning algorithm is not suited for this kind of environment, because for a large state space, there is no efficient way of storing the Q-values for all state-action pairs in memory.

Since Q is a function, the issue can be resolved by introducing a function approximator like a DNN which takes the current state s_t as the input and approximates a vector of Q-values for every action and state. The first implementation of this idea was the Deep Q-Network (DQN, Mnih et al. 2013), which originated the class of Deep Reinforcement Learning (DRL) methods. DQN achieved human-level results on the complex task of controlling Atari 2600 video games by processing raw pixels.

DQN is a modification of the Q-Learning algorithm in the sense that it learns a parameterized Q-function $Q(s_t, a_t, \theta_t)$ where θ_t corresponds to the adjustable parameters of the Q-Network at time step t. Therefore, learning does no longer happen by updating the Q-values, but rather by adjusting the parameters in a way that the Q-Network moves towards approximating an optimal Q-function Q^*. Figure 2.6 resembles Figure 2.5 and illustrates the interaction cycle of an agent when utilizing a Q-Network for value approximation.

At each time step t, a loss $L_t(\theta_t)$ in dependence of the current network parameters θ_t is calculated according to Equation 2.5. The calculation uses the MSE of the target value, i.e., the expected maximum cumulative reward $r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a, \theta_t)$, and the actual Q-value.

$$L_t(\theta_t) = (r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a, \theta_t) - Q(s_t, a_t, \theta_t))^2 \tag{2.5}$$

Based on the loss, in Equation 2.6, a gradient is computed with respect to the network parameters which is then propagated back through the Q-Network.

$$\nabla_{\theta_t} L_t(\theta_t) = (r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a, \theta_t) - Q(s_t, a_t, \theta_t)) \nabla_{\theta_t} Q(s_t, a_t, \theta_t) \tag{2.6}$$

Equation 2.7 formulates this as an update rule that resembles the original Q-Learning update rule from Equation 2.3.

$$\theta_{t+1} \leftarrow \theta_t + \alpha(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a, \theta_t) - Q(s_t, a_t, \theta_t)) \nabla_{\theta_t} Q(s_t, a_t, \theta_t) \tag{2.7}$$

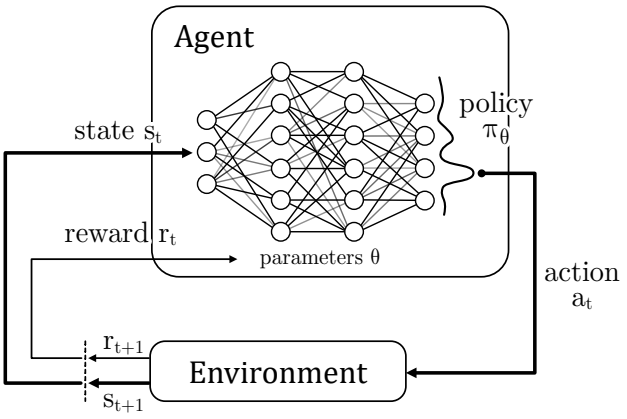

Figure 2.6. DRL using a Q-Network for Q-value approximation. A policy is derived from the Q-values by choosing the action with the highest value in the current state.

In supervised learning, a loss can simply be calculated as the distance between predicted output and expected output, while in RL the expected output has to be approximated. To be more precise, the Q-Network not only approximates the current Q-values but is also utilized to approximate the target values used for learning. This approach is called *bootstrapping*. Although the reward r_t from Equation 2.7 comes from actual experience, for the most part the target value is based on an approximation from the Q-Network. Therefore, the calculated gradient with respect to the network parameters is based on an approximated loss, which makes DQN a *semi-gradient* method where the gradient does not contain the whole information but is rather based on an approximation.

Since new data become available at every time step, the parameters of the Q-Network are constantly adjusted. Using this unstable Q-Network for approximating the target values in order to calculate a loss can lead to instabilities in training.

Therefore, (Mnih et al. 2013) propose a second Q-Network called *target network* with an identical architecture and its own set of parameters θ^-. Unlike the first Q-Network, which is referred to as the *online network*, the target network does not adjust its parameters after every time step. Instead, the parameters are frozen for a fixed number of steps after which the target network is synchronized with the online network. Using a target network, Equation 2.8 modifies the update rule from Equation 2.7 using the target parameters θ^- for approximating the target value.

$$\theta_{t+1} \leftarrow \theta_t + \alpha(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a, \theta_t^-) - Q(s_t, a_t, \theta_t))\nabla_{\theta_t} Q(s_t, a_t, \theta_t) \tag{2.8}$$

Despite having two sets of network parameters, only the online network is actively learning, which makes it still only one single model and therefore does not affect performance in a negative way.

(Mnih et al. 2013) propose another modification to the DQN method in order to stabilize training, which is experience replay (ER). During the training, consecutive observations might be highly correlated with each other, which can further destabilize training. Therefore, ER makes use of a memory to store observations. The agent then samples random batches from this memory for training the Q-Network, which breaks correlation between training data. ER not only stabilizes training but also improves sample efficiency by repeatedly reusing past observations in training. The method is further discussed in (Kalyanakrishnan et al. 2007).

Algorithm 2 shows the complete DQN implementation based on (Mnih et al. 2013) with limitation to a deterministic environment.

Algorithm 2: DQN with Experience Replay

Input: learning rate $\alpha \in [0, 1]$, exploration rate $\varepsilon \in [0, 1]$
Initialize replay memory M;
Initialize action-value function Q with arbitrary weights;
foreach *episode* **do**
  Start in state s_t;
  while s_t *is not terminal* **do**
    Choose action a_t randomly with ε probability, else $a_t = \pi(s_t)$;
    Take action a_t;
    Observe next state s_{t+1} and reward r_t;
    Store observation (s_t, a_t, r_t, s_{t+1}) in M;
    Sample random minibatch of observations (s_j, a_j, r_j, s_{j+1}) from M;
    Calculate target value

$$\hat{y}_j = \begin{cases} \mathrm{r}_j + \gamma \max_{a \in \mathcal{A}} Q(s_{j+1}, a, \theta_t^-) & \text{if } s_{j+1} \text{ is not terminal} \\ \mathrm{r}_j & \text{otherwise} \end{cases}$$

    Perform a gradient descent step on $(\hat{y}_j - Q(s_j, a_j, \theta_j))^2$;
    $s_t \leftarrow s_{t+1}$;
  end
end
Output: Q

While (Mnih et al. 2013) use a CNN architecture as a feature extractor for raw image input, DQN can be applied to various types of environments with a large or infinite state space. In chapter 3, DQN is applied to an industrial real-time network environment in order to learn an optimal Q-function and derive a policy for configuration of SP transmission selection.

2.3.3 Policy Gradient

Both Q-Learning and DQN aim to optimize an action-value function from which a policy is derived. This section introduces policy gradient as a different RL approach that aims to learn a parameterized policy directly, without consulting a value function. Such a method is considered *on-policy* and represents a second fundamental way of learning as opposed to the off-policy methods from the previous sections. Concepts described in this section are covered in (Sutton et al. 2018, Chapter 13).

The first policy gradient method was introduced by (Williams 1992). The goal of the policy gradient method is to directly learn a parameterized policy π_θ where θ denotes the parameter vector. The method makes use of a DNN as a function approximator for π_θ.

Value-based methods derive a deterministic policy from a learned value function. *Deterministic* refers to the fact that the derived policy maps a state to one specific action for the agent to take, i.e., the one action with the highest value.

Policy gradient methods, in contrast, learn a stochastic policy, which is a probability distribution $\pi_\theta(a|s) = P_\theta(a|s)$ for every action a when in state s. There are two advantages in learning a stochastic policy. The first is that action probabilities change smoothly during training whereas in ε-greedy selection, the action probabilities can change dramatically if the underlying estimated action values change. The second advantage is that there is no need for an explicit exploration strategy such as ε-greedy because a stochastic policy naturally results in exploration with a certain probability.

In order to actually learn a policy, Equation 2.9 defines a score function in dependence of the parameters θ.

$$J(\theta) = \mathbb{E}_{\pi_\theta}(\sum_t \gamma^t R(s_t, a_t)) \tag{2.9}$$

The score corresponds to the expected total reward with γ being the discount factor. The function can also be rewritten as Equation 2.10 where $V(s)$ is a value function.

$$J(\theta) = \mathbb{E}_{\pi_\theta}(\sum_t \gamma^t R(s_t, a_t)) = \mathbb{E}_{\pi_\theta}(V(s_t)) \tag{2.10}$$

Although policy gradient methods do not use a value function to derive a policy, there can still be a value function used in the learning process. This is not an action-value function $Q(s, a)$ but rather a state-value function $V(s)$ that corresponds to the expected cumulative future reward from a state s regardless of an action.

$J(\theta)$ evaluates the quality of the current policy π_θ during training. The goal of policy gradient is to optimize the parameters θ in order to maximize $J(\theta)$. The basic idea is that the parameters are gradually adjusted in a way that the probability of profitable actions increases over time, thus maximizing the total reward.

For this purpose, in Equation 2.11, a gradient with respect to the parameters θ is calculated that corresponds to the direction of the steepest increase of the score function.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(R(\tau)\nabla_\theta(\log \pi_\theta(a|s))) \tag{2.11}$$

The gradient is computed from the actual total reward $R(\tau)$ that was observed over the course of an episode multiplied by the probabilities according to π_θ. Using log probabilities $\log \pi_\theta(a|s)$ instead of the actual probability function makes it easier to differentiate. A high total reward $R(\tau)$ indicates that the actions taken during the episode, on average, led to high rewards. Therefore, it is desired to increase the probability of taking these actions. This is done with a gradient ascent step which Equation 2.12 formulates as an update rule that resembles Equation 2.7.

$$\theta_{t+1} \leftarrow \theta_t + \alpha R(\tau)\nabla_\theta(\log \pi_\theta(a|s)) \tag{2.12}$$

As with previous parameterized functions, α denotes the learning rate. The update based on the policy gradient adjusts the parameters in a direction that favors actions with a high return, thus gradually optimizing the score function $J(\theta)$.

2.3.4 Actor-Critic

The most significant drawback of the described policy gradient method is that the gradient ascent step only applies at the end of an episode. For an overall high value score $R(\tau)$, all actions taken in the episode are considered profitable, even if some actions actually led to low or negative rewards. As the method averages all actions based on the value score, it usually is not efficient in learning which specific actions are actually profitable and which are not. Thus, learning requires a large amount of samples and convergence is usually slow.

To address this issue, the Actor-Critic algorithm is introduced. It is a temporal difference method based on policy gradient that updates parameters at each time step rather than at the end of an episode. Actor-Critic requires learning the previously mentioned state-value function $V_w(s)$, which is used to update the parameters of the policy at each time step.

As $V_w(s)$ is a parameterized function, a second model is required for approximation. Therefore, the model is split into an actor and a critic. The actor learns the policy π_θ by adjusting its parameters θ, while the critic learns the state-value function $V_w(s)$ with its own set of parameters w. For the purpose of learning, a temporal-difference error is calculated from the state-values (cf. Equation 2.13) at each time step.

$$\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t) \tag{2.13}$$

Using the error, Equation 2.14 formulates an update rule for the actor that resembles Equation 2.12.

$$\theta_{t+1} \leftarrow \theta_t + \alpha_\theta \delta_t \nabla_\theta (\log \pi_\theta(a|s)) \tag{2.14}$$

This adjusts the actor's parameters such that the error δ is ultimately minimized. As the adjustments aim at minimizing an error rather than maximizing a score, this is a gradient *descent* step rather than a gradient ascent step. It is worth noting that both actor and critic use different learning rates α_θ and α_w respectively. For adjusting the critic's parameters, the same error is used (cf. Equation 2.15).

$$w_{t+1} \leftarrow w_t + \alpha_w \delta_t \nabla_{w_t} V_w(s_t) \tag{2.15}$$

Figuratively speaking, the critic observes the current state and provides feedback to the actor's policy. At the same time, the critic learns to optimize its estimation of the state-value to provide better feedback for the next time. All steps combined result in Algorithm 3 adopted from (Sutton et al. 2018).

Algorithm 3: Actor-Critic

Input: learning rates $\alpha_\theta, \alpha_w \in [0, 1]$
Initialize policy π with arbitrary weights;
Initialize state-value function V with arbitrary weights;
foreach *episode* **do**
  Start in state s_t;
  while s_t *is not terminal* **do**
    Choose action $a_t = \pi(s_t)$;
    Take action a_t;
    Observe next state s_{t+1} and reward r_t;
    Compute temporal-difference error $\delta_t = r_t + \gamma V_w(s_{t+1}) - V_w(s_t)$;
    Update actor parameters $\theta_{t+1} \leftarrow \theta_t + \alpha_\theta \delta_t \nabla_{\theta_t} (\log \pi_\theta(a|s))$;
    Update critic parameters $w_{t+1} \leftarrow w_t + \alpha_w \delta_t \nabla_{w_t} V_w(s_t)$;
    $s_t \leftarrow s_{t+1}$
  end
end

In chapter 3, Actor-Critic is applied to the industrial real-time network environment in order to learn an optimal policy for configuration of SP transmission selection. Using both DQN and Actor-Critic methods in the same environment allows for a comprehensive study on the difference between two fundamental ways of learning, off-policy and on-policy.

2.4 Related Work

The industrial network environment as proposed in this work requires deterministic transmission of frames with bounded end-to-end latency. Instead of using traffic shaping mechanisms like CBS, TAS, or ATS, the network only applies basic SP transmission selection.

Shaping methods like CBS and TAS depend on network-wide information to provide end-to-end-latency bounds. For SP, (Grigorjew et al. 2020) propose a mathematically proven method for calculating bounds with bridge-local information, which is fundamental to this work as it allows to meet real-time requirements in a network without the use of additional shaping mechanisms. It also allows the network environment to provide information that can be used to train an agent using the previously covered DRL methods.

The first attempt to apply DRL to dynamic networks was proposed by (Boyan et al. 1994) using the tabular Q-Learning algorithm in order to optimize packet routing, i.e., determine optimal paths for all packets. (Ferrá et al. 2003) were one of the first to apply Q-Learning to packet scheduling in routers. Both methods use a basic environment with a small state-action space and embed the RL module into the nodes of the switching network.

As a more modern approach, (Lin et al. 2016) propose an adaptive routing algorithm based on Q-Learning that takes Quality of Service (QoS) into consideration. The algorithm applies to software-defined networking (SDN), where routing and scheduling mechanisms are decoupled from the actual hardware.

(Feki et al. 2017) introduce a QoS-aware scheduling algorithm for LTE cellular networks based on Q-Learning, which outperforms traditional packet scheduling algorithms like Round Robin. (Kim et al. 2018) propose a method based on Q-Learning that improves scheduling in IoT environments and efficiently learns new scheduling policies even if the environment changes.

Ultimately, (Stampa et al. 2017) propose a routing algorithm based on DQN that produces near-optimal routing configuration and minimizes end-to-end delay in dynamic networks. (Xu et al. 2018) use an Actor-Critic method with prioritized experience replay which significantly reduces end-to-end delay and is also robust to changing environments.

Besides routing and scheduling, congestion control is a task for which RL methods have been successfully applied in the last years. Both (Li et al. 2016) and (Kong et al. 2018) investigate the use of tabular RL methods on congestion control, which, again, only allows for environments with a small state-action space. Still, both methods provide better throughput and delay performance compared to the TCP New Reno mechanism (Henderson et al. 2012).

(Jay et al. 2019) apply an Actor-Critic method to congestion control where the agent is a sender of traffic and actions represent changes to the traffic rate. The agent learns to optimize its sending rate such that network resources are utilized efficiently and outperforms state-of-the-art congestion control methods.

(Ruffy et al. 2018) propose a framework that applies DRL to congestion control in data centers. The work makes use of advanced methods based on policy gradient, namely Deep Deterministic Policy Gradients (DDPG, Lillicrap et al. 2015) and Proximal Policy Optimization (PPO, Schulman et al. 2017). As the proposed environment is deterministic, DDPG provides better results than PPO and exceeds the TCP New Vegas performance baseline (Sing et al. 2005).

As there is a number of alternatives to SP, the application of RL methods on networks solely using SP transmission selection has been neglected for the most part. This work aims to investigate the application of two different types of DRL methods on the configuration of SP in industrial networks and compare the results to two different supervised ML approaches.

3
Methodology

This chapter presents a framework that allows for simulating basic industrial networks with time-sensitive traffic using SP transmission selection. Based on the framework, an environment is proposed which allows to interact with the network. This enables an RL agent to adjust the network configuration and receive feedback based on the change in total network capacity. The chapter then continues with the implementation of both DQN and Actor-Critic to determine a near-optimal SP configuration for a given network environment. This includes the specific implementation details for both methods as well as all relevant hyperparameters.

3.1 Framework for Network Simulation

In order to build an interactive environment for the configuration of industrial networks, a framework is required that allows to simulate such networks and calculate the network capacity based on the current SP configuration. Mathematical symbols used to formalize the framework are listed in Table 3.1.

Table 3.1. Mathematical symbols used to formalize framework and environment

Symbol	Description
$\mathcal{T}$	Network topology
$\mathcal{S}$	Set of all streams $s \in \mathcal{S}$ in the system
$\mathcal{C}$	Network configuration
$N_{\mathcal{C}}^{\mathcal{S}}$	Network capacity under configuration $\mathcal{C}$ regarding streams $\mathcal{S}$
$h(s)$	Number of hops for stream s
$p(s)$	Priority assigned to stream s
$\delta(s)$	End-to-end latency requirement of stream s
$\delta^h(s)$	Per-hop latency requirement of stream s

Networks studied in this work are restricted to a linear topology $\mathcal{T} = (m, n)$. In detail, this means that the network is built upon a number of m linearly arranged switches. Each switch is connected with exactly one controller and a number of n sensors. An exemplary linear topology is illustrated in Figure 3.1, including bidirectional communication between sensors and controllers.

A network with linear topology is characterized by the fact that there is only one possible path between two different endpoints. This particularly means that every path between two endpoints is a shortest path. In networks with a more complex structure, there are usually multiple possible paths between two endpoints, which calls for a routing mechanism that determines optimal paths for the streams. As this work only considers networks with linear topology, routing is negligible.

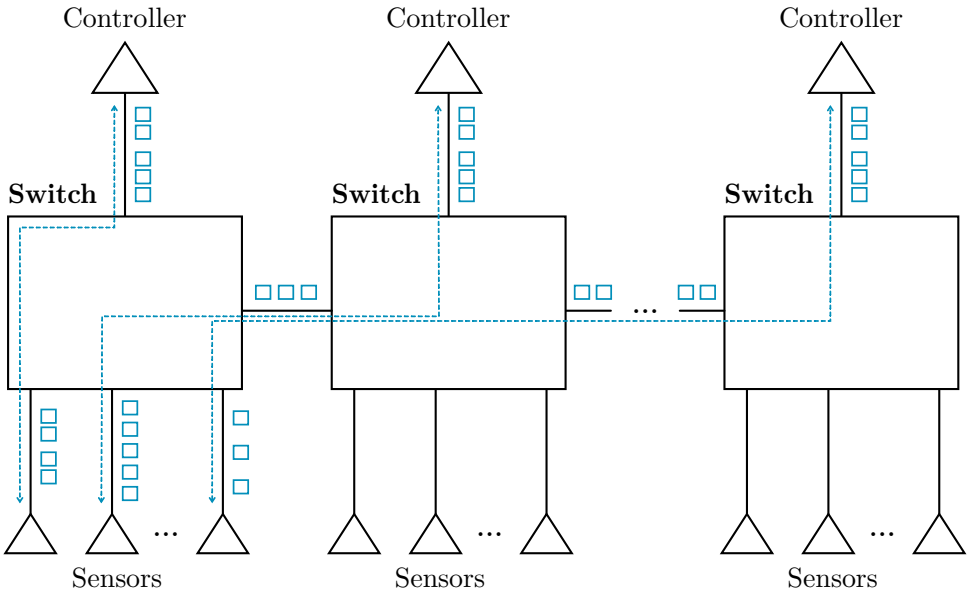

Figure 3.1. Linear network topology with one controller and multiple sensors per switch. Blue arrows indicate the bidirectional communication between endpoints.

Every sensor in the network communicates with exactly one controller via a bidirectional stream of data. More precisely, this corresponds to two unidirectional streams from sensor to controller and vice versa. Streams are periodically, i.e., each stream has a specific transmission interval at which frames are transmitted to the controller. Further properties of a stream are the range of the frame size as well as the end-to-end latency requirement.

In order to provide a relatively realistic simulation of a real network, five different application profiles are specified that can be applied to the streams (cf. Table 3.2). Each profile includes a transmission interval and an end-to-end latency requirement for frames as well as a minimum and maximum frame size, whereby all the properties double between consecutive profiles.

Table 3.2. Available application profiles for the streams.

	Transmission interval	Maximum latency	Minimum frame size	Maximum frame size
profile 1	250 μs	250 μs	64 bytes	128 bytes
profile 2	500 μs	500 μs	128 bytes	256 bytes
profile 3	1000 μs	1000 μs	256 bytes	512 bytes
profile 4	2000 μs	2000 μs	512 bytes	1024 bytes
profile 5	4000 μs	4000 μs	1024 bytes	1522 bytes

Properties for each stream $s \in \mathcal{S}$ are randomly chosen from one of the application profiles according to a uniform distribution. Furthermore, the end-to-end latency requirement $\delta(s)$ is identical to the transmission interval. As a stream can take only one distinct path through the network, the same is true for the number of hops $h(s)$ on this path. Therefore, the per-hop latency requirement for s can be calculated according to Equation 3.1.

$$\delta^h(s) = \frac{\delta(s)}{h(s)} \tag{3.1}$$

Lastly, the network uses burst transmission whereby a stream can exceed its transmission rate for a short period of time. Regardless of the application profile, however, the burst size cannot exceed the maximum frame size of 1522 bytes specified in (IEEE 802.3 2018).

The framework does not make use of any additional traffic shaping mechanisms but rather utilizes basic SP transmission selection. (IEEE 802.1Q 2018) specifies eight classes that can be used for streams of different priority. The framework makes use of up to four classes for traffic with tight end-to-end latency requirements while the other classes are reserved for best-effort traffic. For each outgoing port, a switch manages one dedicated queue for each of the four priority classes. Frames in a queue of higher priority always get transmitted first. Frames within the same queue get processed according to the FIFO principle.

The primary configuration option provided by the framework is the maximum per-hop latency that is guaranteed by each of the four priority classes. This section assumes that all four classes are used for the transmission of time-sensitive streams. Equation 3.2 defines the configuration of the network as a 4-tuple where C_i is the guaranteed maximum per-hop latency provided by priority class i.

$$\mathcal{C} = (C_0, C_1, C_2, C_3) \tag{3.2}$$

C_0 denotes the highest priority class with the lowest guaranteed per-hop latency. The priority $p(s)$ of a stream s is determined by the lowest priority class that satisfies the per-hop requirement $\delta^h(s)$.

If there is no priority class that meets this requirement, s is not assigned a priority and therefore cannot be added to the network under the current configuration. Table 3.3 shows the exemplary priority assignment for a small set of four streams given the stream profile and the path length. $\mathcal{C} = (75\,\mu s, 150\,\mu s, 250\,\mu s, 500\,\mu s)$ is used as an configuration for the example.

	Application profile	Path length	End-to-end requirement	Hops on path	Per-hop requirement	Priority assigned
s_0	profile 1	3	$\delta = 250\,\mu s$	2	$\delta^h = 125\,\mu s$	$p = 0$
s_1	profile 1	5	$\delta = 250\,\mu s$	4	$\delta^h = 62.5\,\mu s$	$p =$ n.d.
s_2	profile 2	4	$\delta = 500\,\mu s$	3	$\delta^h = 166.67\,\mu s$	$p = 1$
s_3	profile 5	5	$\delta = 4000\,\mu s$	4	$\delta^h = 1000\,\mu s$	$p = 3$

Table 3.3. Exemplary priority assignment for streams with different profile and path length under configuration $\mathcal{C}$. Priority $p(s_1)$ is undefined due to the fact that there is no priority class that meets the requirements of s_1.

The framework is initialized with a topology $\mathcal{T}$, a set of streams $\mathcal{S}$, and a configuration $\mathcal{C}$. The main task of the framework is to simulate the real-time communication between sensors and controllers solely utilizing SP, and to determine the actual network capacity $N_{\mathcal{C}}^{\mathcal{S}}$. The capacity is defined as the number of streams $s \in \mathcal{S}$ that can be added to the network under the current configuration $\mathcal{C}$ without violating the end-to-end latency requirements of any stream in the network.

As the previous example illustrated, streams cannot be added to the network if there is no priority class that meets its per-hop requirements. However, $N_{\mathcal{C}}^{\mathcal{S}}$ does not only depend on the number of prioritizable streams, but also on the percentage of those streams that can actually transmit frames at their transmission rate without violating any end-to-end requirements due to transmission delay.

In order to identify such streams, the framework uses the latency bound calculation method of (Grigorjew et al. 2020). The method is applied by iteratively adding the prioritizable streams to the network and calculating the occurring latency at every hop. If adding a stream violates the end-to-end requirements of the stream or any other stream, that has already been added to the network, the stream is rejected.

The framework allows to simulate and configure basic industrial networks with linear topology using SP transmission selection. It provides an interface that allows to adjust the current configuration and re-calculate the network capacity under the new configuration. This serves as a basis for the RL environment that is introduced in the next section.

3.2 Environment for Reinforcement Learning

Identifying an optimal configuration $\mathcal{C}^*$ for a given network topology and a set of streams is a hard optimization problem. Reinforcement Learning is expected to be an appropriate method that approaches a decent configuration in reasonable time. Therefore, this section proposes an interactive environment built upon the framework that allows to apply different RL methods to the problem.

Two essential components of such an environment are the action space and the state space. As the objective is to identify an optimal configuration $\mathcal{C}^*$, actions represent adjustments to the configuration or, in other words, to the guaranteed per-hop latency of the priority classes. The action space is defined in Equation 3.3 where p is the number of classes that are used by the framework.

$$\mathcal{A} = (a_0, a_1, ..., a_{2p-1}) \tag{3.3}$$

For every priority class, the environment provides one action to increase and one action to decrease the guaranteed per-hop latency of the class by $10\,\mu s$, respectively. To be precise, an action does not adjust the absolute value but rather the absolute distance between class C_i and the next lower priority class C_{i-1}. This results in an adjustment of $\pm 10\,\mu s$ to the class C_i itself as well as all higher priority classes. This may be illustrated by an exemplary configuration $\mathcal{C}_t$ and an action space $\mathcal{A} = (a_0, a_1, ..., a_7)$. Figure 3.2 illustrates a short, exemplary sequence of actions and the resulting adjustments to the network configuration $\Delta\mathcal{C}$. This shows that, e.g., action a_0 increases the configuration of all four priority classes by $10\,\mu s$.

$$\begin{aligned}
\mathcal{C}_t &= (50, 100, 200, 500) \\
&\quad \downarrow a_0 \quad \Delta\mathcal{C} = (+10,\ +10,\ +10,\ +10) \\
\mathcal{C}_{t+1} &= (\mathbf{60}, \mathbf{110}, \mathbf{210}, \mathbf{510}) \\
&\quad \downarrow a_2 \quad \Delta\mathcal{C} = (\pm 0,\ +10,\ +10,\ +10) \\
\mathcal{C}_{t+2} &= (60, \mathbf{120}, \mathbf{220}, \mathbf{520}) \\
&\quad \downarrow a_5 \quad \Delta\mathcal{C} = (\pm 0,\ \pm 0,\ -10,\ -10) \\
\mathcal{C}_{t+3} &= (60, 120, \mathbf{210}, \mathbf{510})
\end{aligned}$$

Figure 3.2. Exemplary sequence of actions and the resulting adjustments in configuration denoted as $\Delta\mathcal{C}$.

Figure 3.3 shows the interaction cycle with the proposed environment. An action a_t results in an adjustment of the configuration $\Delta\mathcal{C}$. The framework re-calculates the network capacity $N_{\mathcal{C}}^{\mathcal{S}}$ under the new configuration and passes this information back to the environment.

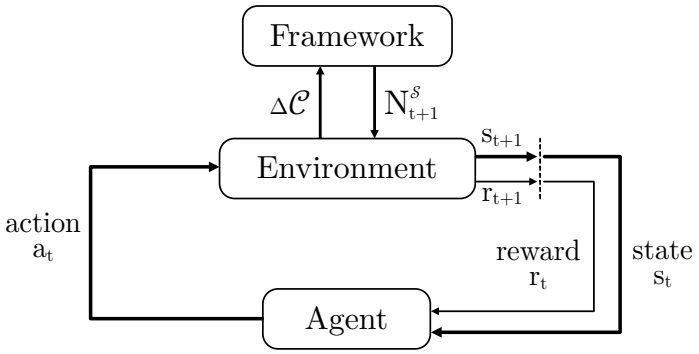

Figure 3.3. Interaction with the environment built upon the framework.

If the absolute distance between two classes is less than $10\,\mu s$, an action that would further decrease the distance between both classes results in a no-op action. In general, no-op actions are actions that are invalid in the current state and therefore do not affect the environment. This is particularly true for action a_1 if $C_0 \leq 10\,\mu s$ since the action would result in a negative configuration.

At the beginning of an episode, framework and environment are initialized with a topology and a set of streams. The initial configuration of the network is not chosen randomly, but rather as a rough estimation of a configuration that would match the given set of streams. This initial choice deserves some consideration because it can significantly speed up the training process if there is already a reasonable initial network configuration. In order to choose such a configuration, the actual occurring per-hop latency requirements for all streams are consulted. Let Equation 3.4 define an ordered list of all uniquely occurring per-hop requirements.

$$\Delta = (\delta^h(s) \,|\, s \in \mathcal{S}) \tag{3.4}$$

The initial network configuration $\mathcal{C}$ is then determined according to Equation 3.5.

$$\mathcal{C}_0 = \begin{cases} (\Delta_0, \Delta_{0.25}, \Delta_{0.5}, \Delta_{0.75}) & \text{if } p = 4 \\ (\Delta_0, \Delta_{0.33}, \Delta_{0.66}) & \text{if } p = 3 \\ (\Delta_0, \Delta_{0.5}) & \text{if } p = 2 \end{cases} \tag{3.5}$$

Δ_0 always denotes the lowest occurring per-hop requirement. $\Delta_{0.25}$, $\Delta_{0.33}$, $\Delta_{0.5}$, $\Delta_{0.66}$, and $\Delta_{0.75}$ denote the 25th, 33th, 50th, 66th, and 75th percentile of all occurring values, i.e., the smallest value that is higher than 25%, 33%, 50%, 66%, and 75% of all other values in the list.

Equation 3.6 shows an exemplary list of per-hop requirements and the corresponding initial configuration $\mathcal{C}_0 = (\Delta_0, \Delta_{0.25}, \Delta_{0.5}, \Delta_{0.75}) = (62.5\,\mu s, 125\,\mu s, 250\,\mu s, 1000\,\mu s)$.

$$\Delta = \{\underbrace{\mathbf{62.5}}_{\Delta_0}, 83.33, \underbrace{\mathbf{125}}_{\Delta_{0.25}}, 166.66, 200, \underbrace{\mathbf{250}}_{\Delta_{0.5}}, 333.33, 500, \underbrace{\mathbf{1000}}_{\Delta_{0.75}}, 1333.3\} \tag{3.6}$$

Besides an action space, the environment also provides a representation of the current network state. As an action results in an adjustment of the network configuration, it also results in a state transition that represents the change in network capacity.

The environment is meant to handle networks of different size and different numbers of streams. As the environment is used to apply DRL methods, the state representation is ultimately used as an input for a DNN that approximates either a value function or a policy. Such DNNs are restricted to a fixed number of input neurons, therefore, the state representation has to be of the same size regardless of the network topology or number of streams.

This means that the state representation must not involve a list of switches, endpoints, or streams as the number of elements may vary to a large extent. Instead, a more general representation has to be chosen as shown in Table 3.4. The state representation can be divided into four different groups. The first group includes topology-related features like the number of different network devices. The network diameter and path length provide additional information about the structure of the network. The second group includes features related to the set of all streams in the system. Information about the capacity of the network and the occurring transmission delays is provided by the third group. It uses both a hypothetical static configuration as well as the current dynamic configuration for the calculation. The last group directly represents the current configuration as it includes the guaranteed per-hop latency for all priority classes as well as the number of streams that have successfully been added to the network and the ones that have been rejected.

Altogether, the state is represented by a vector of 76 both static and dynamic features. As most of the features have a continuous range of values, this results in a large state space where most of the states are not expected to ever be observed when interacting with the environment.

Table 3.4. Representation of the current network state as a vector of 76 features.

Group	Value
Topology	Number of sensors
	Number of controllers
	Number of switches
	Network diameter *The number of hops of the longest path through the network.*
	Path length *Minimum, maximum, and mean number of hops.*
	Link speed *Minimum, maximum, and mean link speed.*
Streams	Number of streams in total
	Stream interval *Minimum, maximum, and mean of transmission intervals.*
	Stream bursts *Minimum, maximum, and mean of stream bursts.*
Latency	Static latency bound *Minimum, maximum, and mean latency bound for each priority calculated with the method of (Grigorjew et al. 2020) using a static configuration that utilizes a hypothetical number of 20 priority classes. This hypothetical calculation provides valuable information about the potential capacity of the network without considering the current configuration.*
	Dynamic latency bound *Minimum, maximum, and mean latency bound using the current configuration of the network. This calculation provides information about the actual capacity of the network.*
Configuration	Network capacity *Number of streams successfully added to the network.*
	Number of rejected streams *This includes streams that could not be prioritized and streams that have been rejected due to violation of latency requirements.*
	Current configuration *The configuration itself is also part of the current state.*

In RL, an episode of actions normally ends when a terminal state is reached. But as the problem studied in this work is an optimization problem, it is not trivial to determine if a state is actually terminal. This is due to the fact that neither the optimal configuration nor the maximum capacity of the network with respect to a set of streams are known in advance. Apart from the trivial case, where all streams can successfully be added and the current capacity at time step t is $N_t^{\mathcal{S}} = |\mathcal{S}|$, there is no way to determine if the current configuration already is optimal or not.

For this reason, an additional mechanism called *early stopping* is applied. Especially for large sets of streams, an optimal solution is not likely to allow all streams to be added to the network. Early stopping prevents episodes from continuing endlessly in case that a terminal state is impossible to reach.

The mechanism is implemented in a way that the agent is granted a minimum number of n actions. Thereafter, the current network capacity $N_t^{\mathcal{S}}$ is compared to the capacity before the sequence of actions, which is denoted as $N_{min}^{\mathcal{S}}$. For the case that $N_t^{\mathcal{S}} < N_{min}^{\mathcal{S}}$, the episode stops because the agent decreased the network capacity with its past sequence of actions. Otherwise, if $N_t^{\mathcal{S}} \geq N_{min}^{\mathcal{S}}$, the agent is granted another n actions to take and $N_{min}^{\mathcal{S}} := N_t^{\mathcal{S}}$.

The early stopping mechanism ensures that after the next sequence of actions, the agent must not continue if it fell back behind previously made progress. This is repeated until the episode ultimately stops at a maximum number of actions, regardless if the agent has made further progress or not. This ensures a reasonable time frame for episodes and prevents long sequences of no-op actions where the agent does not decrease capacity, but does not show progress either.

Utilizing the environment for training a RL agent requires a reward signal, which allows the agent to learn reasonable behaviour. Therefore, a reward function is defined that determines the objective the agent learns to optimize.

In this work, the objective is to adjust the network configuration in a way that maximizes network capacity. Equation 3.7 defines a reward function that corresponds to the percentage increase in capacity after adjusting the network configuration.

$$r_t = \begin{cases} \frac{N_t - N_{t-1}}{|\mathcal{S}|} \cdot 120 & \text{if } N_t = |\mathcal{S}| \\ \frac{N_t - N_{t-1}}{|\mathcal{S}|} \cdot 100 & \text{otherwise} \end{cases} \tag{3.7}$$

This function is designed for the agent to maximize the network capacity and additionally rewards the only certainly terminal state where $N_t^{\mathcal{S}} = |\mathcal{S}|$ by 20%. The environment provides immediate rewards, which means that the agent receives a reward signal after every single action rather than at the end of an episode.

3.3 Implementation of Deep Q-Network

The first RL method that is applied to the industrial network configuration problem is DQN. As an extension to the well-studied Q-Learning method it has already been successfully applied to various problems like video game control (Mnih et al. 2013) or robotic control (Gu et al. 2016).

As DQN is capable of dealing with the large state representation used in this work (cf. Table 3.4), it is expected that the method is applicable to the problem of SP configuration in the proposed industrial network environment. This section gives an overview over the implementation of the DQN method, which also serves as a representative for the group of off-policy learning methods.

One crucial implementation detail is the architecture of the Q-Network, specifically the number of hidden layers and neurons. Early investigations by (Cybenko 1989, Hornik 1991, Leshno et al. 1993) came to the conclusion that a DNN with one hidden layer is already sufficient to approximate any conceivable function. (Hinton et al. 2006) were the first to provide evidence that, depending on the application, DNNs with more than one hidden layer are able to learn more complex representations of data in a more efficient manner. (Heaton 2017) states that a DNN with two hidden layers is well suited to approximate a smooth mapping from a large state space to an action space. Additional layers are only needed for the extraction of complex features, as in the case of raw pixel input (Mnih et al. 2013).

In this work, the input of the Q-Network is the state vector as previously shown in Table 3.4. It is expected that a network architecture with two fully-connected hidden layers meets the given requirements. The size of the input and output layer correspond to the size of the state vector and the size of the action space, respectively. The number of neurons per hidden layer, however, is regarded as a tunable hyperparameter that is further examined in Chapter 4.

In order to introduce non-linearity to the Q-Network, ReLU activation is applied to both hidden layers. ReLU is one of the most common activation functions (LeCun et al. 2015) and does not involve expensive calculations. However, no activation is applied to the output layer. This allows for negative output values, which is desirable for the Q-Network as it approximates action-values. The values estimate the cumulative future reward when taking an action and may be either positive or negative, depending on the quality of the action.

According to (Mnih et al. 2013), a target network should be used with an architecture identical to the online network. The number of time steps after which the target network is synchronized with the online network, referred to as *target update rate*, is regarded as a tunable hyperparameter. The target network is utilized to approximate the target value that is used to calculate a loss. The loss is computed with the MSE, according to Equation 2.2. The loss is then used to compute a gradient that is propagated back through the network. To perform the gradient descent step, an optimizer is used.

Root Mean Square Propagation (RMSProp, Tieleman et al. 2012) is a widely used optimizer that maintains an adaptive learning rate on parameter-level, i.e., the learning rate is individual for every parameter in the network. RMSProp is supposed to be effective in online settings and is also used in the original DQN implementation (Mnih et al. 2013). Another commonly used optimizer is Adam (Kingma et al. 2014). Adam is closely related to RMSProp as it also maintains a learning rate for each network parameter. However, the authors demonstrate that Adam can outperform RMSProp and other methods based on gradient descent.

Both methods require another hyperparameter, the learning rate α. Although both methods use adaptive learning rates for the parameters, α provides an approximate upper bound and affects the initial learning, before the learning rate is ultimately adjusted by the optimizer. The utilized optimizer is regarded as a tunable hyperparameter and the performance of both RMSProp and Adam along with different initial learning rates is further evaluated in Chapter 4.

Industrial networks are a highly dynamic environment. Not only communication varies in that endpoints exchange frames of different sizes at different rates. Also, the structure and size of the network can vary in that new connections are established or new endpoints are added to the network. In order to account for this, the agent is trained on a number of different network topologies and for each topology, on a number of different sets of streams, each referred to as a *scenario*.

Methods like DQN require some exploration strategy like ε-greedy. The idea is that, especially in early phases of the training, the agent performs random actions with probability ε in order to discover valuable state-action pairs.

In a dynamic environment where the agent is trained on multiple different scenarios, the utilization of ε-greedy requires careful consideration. Normally, the strategy can be implemented by defining an initial exploration rate ε_0 and then decreasing the exploration rate with each action. This is problematic because when trained on a sequence of different scenarios, a high initial exploration rate would lead to a large number of random actions in early phases of the training and prevent the agent from acquiring reasonable behaviour in early scenarios.

Instead, it is desired to already balance between exploration and exploitation in early scenarios and to keep a steady decrease in exploration over the course of the whole training at the same time. To achieve this, this work proposes a strategy of *dynamic exploration* for DQN, which is illustrated in Figure 3.4. The figure shows both the simple approach, where the exploration rate steadily decreases with each action, as well as the dynamic approach. In both cases, the exploration rate ultimately approaches ε_{min} over the course of the training.

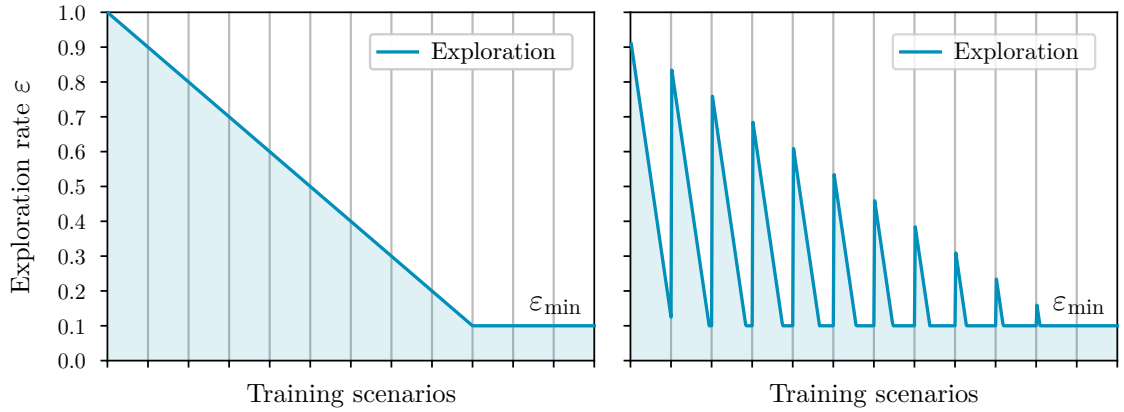

Figure 3.4. Decreasing exploration rate without considering the sequence of scenarios (left) and dynamic exploration with decreasing initial exploration for each network topology and additional reduction of ε with each action (right). Vertical lines indicate different network topologies the agent is trained on.

Let σ denote the total number of different network topologies the agent is trained on. For each network topology i an initial exploration rate $\varepsilon_0(i)$ for the first episode is calculated according to Equation 3.8. The calculation requires hyperparameters $\varepsilon_{\max}$ and $\varepsilon_{\min}$, i.e., the maximum initial exploration rate and the minimum rate, as well as the total number of network topologies σ for training.

$$\varepsilon_0(i) = \max(\varepsilon_{\min},\ \varepsilon_{\max} - i\frac{\varepsilon_{\max} - \varepsilon_{\min}}{\sigma}) \tag{3.8}$$

This results in a steady decrease of initial exploration rates with each network. While training on a network, ε is decreased by a value ε_{dec} after each action. This ensures a balance between exploration and exploitation. As the training progresses, the agent is expected to acquire more reasonable behaviour, thus the exploration rate ultimately approaches $\varepsilon_{\min}$. This work proposes values $\varepsilon_{\max} = 1.0$, $\varepsilon_{\min} = 0.1$, and $\varepsilon_{\text{dec}} \approx 0.001$ in order to keep the balance between exploration and exploitation.

3.4 Implementation of Actor-Critic

The second method implemented in this work is Actor-Critic, which serves as a representative for the group of on-policy learning methods. As with DQN, Actor-Critic is capable of dealing with large state spaces and is expected to be applicable to the proposed industrial network configuration problem.

A key difference to the DQN implementation is the network architecture. While DQN uses the Q-Network to approximate a single action-value function, Actor-Critic utilizes both an actor network to approximate a policy and a critic network that approximates a state-value function.

In general, the term *actor-critic* refers to a group of algorithms and does not make an assumption about the actual implementation of the actor and the critic. (Schulman et al. 2015) implement Actor-Critic utilizing two separate networks for the actor and the critic, respectively. (Mnih et al. 2016) propose a parallelized Actor-Critic implementation built upon a single shared network for both actor and critic. The approach has similarities to Dueling DQN (Wang et al. 2015) which uses a shared Q-Network that ultimately splits into a state-value and action-value approximator.

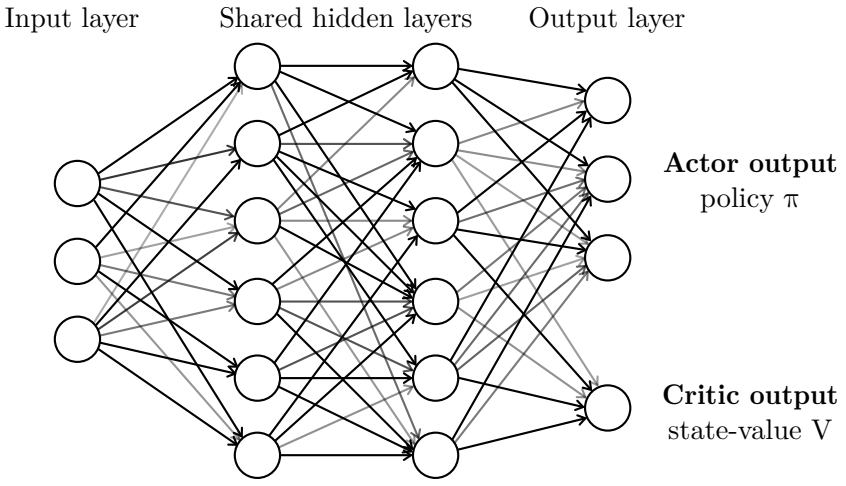

Figure 3.5. Basic Actor-Critic network with shared architecture and split output layer for actor and critic.

This work implements Actor-Critic using a single shared network for feature extraction and a split output layer for both actor and critic (cf. Figure 3.5). The approach is easy to implement, reduces the number of trainable network parameters, and also ensures better comparability since the network architecture resembles that of DQN for the most part. Therefore, different results can be attributed to the different methods of learning rather than the different number of learnable parameters.

The size of the input layer, again, corresponds to the size of the state vector. As with DQN, the network uses two hidden layers. The output layer of the actor has a size equal to the size of the action space while the output layer of the critic has a fixed size of 1 and outputs a single state-value.

ReLU activation is applied to both hidden layers. The split output layer, however, requires more consideration. For the critic output layer, no activation is applied, because the output is a state-value which can either be positive or negative based on the expected cumulative reward. For the actor, however, the output values correspond to a policy, which is a probability distribution.

For this reason, Softmax activation is applied as defined in Equation 3.9.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp x_j} \tag{3.9}$$

Softmax ensures that output values are non-negative values within the interval $[0, 1]$ and that all output values sum up to 1. This results in the desired probability distribution which represents the current policy.

For the purpose of learning, a temporal-difference error is calculated at each time step according to Equation 2.13. The gradient descent step basically follows Equations 2.14 and 2.15. Because the implementation utilizes one shared network for both actor and critic, the gradient is computed from the sum of the actor loss and the critic loss and is then propagated back through the shared network. This results in a simultaneous optimization of the policy and the accuracy of the state-value. For the gradient descent step, again, either RMSProp or Adam optimizer is used.

As opposed to DQN, Actor-Critic does not utilize a sample memory. This comes from the fact that Actor-Critic is an on-policy method which computes a gradient based on the current log probabilities. This is important as the adjustment to the parameters is supposed to increase the probability of actions that led to high rewards. An observation that is used to compute a gradient and adjust the network parameters must always be associated with the policy that led to this observation. Therefore, using batches of random samples is expected to prevent the algorithm from convergence. A target network is also not required as the approximation of a target value is done by the critic.

Another key difference is the lack of an exploration strategy like the dynamic exploration proposed in Section 3.3. Instead, the stochastic policy naturally results in exploration with a certain probability and still allows to associate an observation to the corresponding policy.

Overall, the Actor-Critic method used in this work is more straightforward than the DQN method as it involves less implementation details und therefore introduces less tunable hyperparameters.

4
Evaluation

This chapter describes the training procedure and the evaluation of the two implemented methods, DQN and Actor-Critic. For each method, the chapter gives an overview over the tunable hyperparameters and the experiment that was conducted in order to determine the effect of each hyperparameter on the results. The hyperparameters for both methods were then optimized in a way that maximizes the resulting network capacities. Results were obtained on two data sets of different complexity and were ultimately compared to two different supervised ML models that have been trained on ground truth data. The chapter concludes by demonstrating the flexibility of the two DRL methods when varying the number of priority classes utilized by the environment.

4.1 Training and Experimental Setup

This work utilizes two different data sets. The first one is a set of unique linear network topologies with $n \in \{2, ..., 7\}$ switches, $m \in \{5, 10, 15, ..., 40\}$ sensors, and one controller per switch. This results in a total of 48 different network topologies. As each sensor communicates with exactly one controller in a bidirectional manner, the number of streams in the system is defined as $|\mathcal{S}| = 2 \cdot m \cdot n$. For each network topology, the data set includes a large number of different scenarios with varying application profiles for the streams (cf. Table 3.2).

For the training, a subset of 24 network topologies was selected, hereafter referred to as *training set*. Networks with a small number of streams usually have an initial network capacity that is already optimal without additional configuration. Such networks, particularly the ones with $n \in \{5, 10, 15\}$ sensors per switch, were not included in the training set.

For the evaluation, another subset of the data set was selected. This set, hereafter referred to as *validation set*, consists of 400 scenarios that were randomly chosen from the data set. Although stream properties vary between training and evaluation, the validation set consists of network topologies on which the agent is also trained on.

Therefore, this work utilizes a second data set, hereafter referred to as *test set*. This set consists of linear network topologies on which the agent is not explicitly trained on and which generally are of larger size compared to the ones used for training. More precisely, the test set includes linear networks with $n \in \{3, ..., 9\}$ switches, $m \in \{10, 11, 12, ..., 60\}$ sensors, and one controller per switch. On average, this results in much more complex network environments in which the distance between the total number of streams and the actual network capacity is expected to be much larger. Evaluating on the test set not only accounts for overfitting to the training data, but also indicates if the model is able to generalize beyond the network topologies it has been trained on.

An issue that was identified in early phases of this work is the reproducibility of results. Despite the deterministic nature of the network environment, the overall training environment of the agent is still non-deterministic due to different factors. More precisely, the following sources of non-determinism were identified:

- Initialization of the DNN parameters
- Order of network topologies chosen from the training set
- Order of scenarios for each network topology
- Order of scenarios chosen from the validation set
- Choice of actions when following ε-greedy exploration (DQN only)
- Batches of samples for training the Q-Network (DQN only)

This leads to results that are generally not reproducible, even when using identical hyperparameters. To address this issue, deterministic random seeds were used for every source of non-determinism in the training environment.

Furthermore, it was observed that, using the same hyperparameters, a different random seed can lead to variations in the results. In order to obtain more robust results, each time, three models were trained in parallel using three different random seeds. The results were then calculated as the mean of all three results, which accounts for possible variations that can not be attributed to a variation of the hyperparameters.

4.2 Evaluation of Deep Q-Network

The first method to be evaluated was DQN. Section 3.3 covered the implementation of DQN in great detail. This involved a number of hyperparameters of which some require further examination:

A) **Epochs** ($epochs \in \mathbb{N}$).
This is the number of actions used for the early stopping mechanism. It serves both as a minimum number of actions and as the number of additional actions granted to the agent. The maximum length of an episode is defined as $10 \cdot epochs$. This reduces the configuration of the early stopping mechanism to a single hyperparameter.

B) **Episodes** ($episodes \in \mathbb{N}$).
This is the number of episodes, i.e., the number of scenarios the agent is trained on each network topology of the training set.

C) **Learning rate** ($\alpha \in [0, 1]$).
This value determines to which extent the Q-Network parameters are adjusted at each time step.

D) **Discount factor** ($\gamma \in [0, 1]$).
This value discounts future rewards and gives the agent an incentive to optimize the network capacity in as little time as possible.

E) **Optimizer** ($opt \in \{\text{Adam}, \text{RMSProp}\}$).
This determines which optimizer is used for the gradient descent step.

F) **Hidden nodes** ($nodes \in \mathbb{N}$).
This specifies the size of the hidden layers of the Q-Network. In order to reduce the number of hyperparameters, for now, *nodes* is used for both hidden layers.

G) **Target update rate** ($target\_update \in \mathbb{N}$).
This determines the number of time steps after which the target network is synchronized with the online network.

H) **Batch size** ($batch\_size \in \mathbb{N}$).
This determines the size of the sample batch used for training the Q-Network.

It is obvious that the number of hyperparameters could have been further increased, e.g., by differentiating the size of the two hidden layers, using other activation functions than ReLU or other loss functions than MSE. However, the number of factors exponentially increases the number of possible configurations and was therefore reduced to the ones that were expected to have the greatest effect on the results.

In order to determine the effect of the eight hyperparameters on the results, an experiment was conducted. The first approach that comes to mind is to study each factor separately while holding the others constant. However, when studying a number of eight factors, this is neither sufficient nor was it manageable in the time frame of this work.

Instead, a factorial design approach (Montgomery 2013) was used, where two discrete levels are specified for each individual factor. More precisely, for each factor there is one high level value, denoted as "+", and one low level value, denoted as "−". Table 4.1 shows the levels for each of the eight hyperparameters A to H, which were not expected to be optimal, but still chosen reasonably with respect to the network environment.

Table 4.1. High level and low level values for the eight DQN hyperparameters.

	A	B	C	D	E	F	G	H
+	16	200	0.01	0.99	RMSProp	100	200	100
−	4	20	0.001	0.8	Adam	40	50	25

The factorial design determines the effect of each individual factor on the results as well as the interdependence of effects, often referred to as *interactions*. Ideally, every possible combination of factors is studied in a full factorial design. Although the range of values for each factor is reduced to two discrete levels, a full factorial design involving eight factors still requires $2^8 = 256$ runs. This was, again, not manageable in the time frame of this work.

Instead, this work followed a fractional factorial design approach, where a subset of the full factorial design was chosen. (Montgomery 2013) states that, in a full factorial experiment involving a number of ≥ 6 factors, only a small proportion of the factors have significant effects on the results.

In a fractional factorial design, some of the effects or interactions are confounded, which means that they cannot be estimated independently of each other. It is desired for the fractional design to have the highest possible resolution. The design resolution determines the ability to separate main effects and low-order interactions from each other. The fractional factorial design used for the DQN experiment was a 2^{8-4}_{IV} factorial design (Myers et al. 2016). As it is a resolution IV design, main effects are confounded by two-factor interactions and two-factor interactions are aliased with each other (Montgomery 2013). The 2^{8-4}_{IV} design allows for studying a number of eight factors with only $2^4 = 16$ runs.

Table 4.2 shows the design matrix of the selected factorial design. The matrix includes the levels of the eight factors for each of the 16 runs. The experiment studied all combinations of the factors A to D, while the other four factors were represented by aliases $E = \pm BCD$, $F = \pm ACD$, $G = \pm ABC$ and $H = \pm ABD$. The table includes the results for each run on both the validation set and the test set. The results correspond to the maximum percentage increase in network capacity from an initial baseline using the specified levels for the hyperparameters.

Table 4.2. Design matrix of the selected 2^{8-4}_{IV} fractional factorial design adopted from (Myers et al. 2016, Table 4.13). The results on both data sets correspond to the maximum percentage increase in network capacity from an initial baseline.

Run	A	B	C	D	E	F	G	H	Val.	Test
1	+	+	+	+	+	+	+	+	2.117 %	1.608 %
2	+	+	+	+	−	−	−	−	1.792 %	1.340 %
3	+	+	−	−	+	+	−	−	1.655 %	1.403 %
4	+	+	−	−	−	−	+	+	2.117 %	1.608 %
5	+	−	+	−	+	−	+	−	2.117 %	1.608 %
6	+	−	+	−	−	+	−	+	2.117 %	1.608 %
7	+	−	−	+	+	−	−	+	0.899 %	0.789 %
8	+	−	−	+	−	+	+	−	1.661 %	1.268 %
9	−	+	+	+	+	+	+	+	0.759 %	0.541 %
10	−	+	+	−	+	−	−	−	0.000 %	0.003 %
11	−	+	−	+	−	−	+	−	0.023 %	0.130 %
12	−	+	−	−	−	+	−	+	0.838 %	0.702 %
13	−	−	+	+	−	−	−	+	0.775 %	0.628 %
14	−	−	+	−	−	+	+	−	0.779 %	0.589 %
15	−	−	−	+	+	+	−	−	0.121 %	0.245 %
16	−	−	−	−	+	−	+	+	0.031 %	0.145 %

For an experiment involving k runs, Equation 4.1 calculates the effect e_Z of a factor Z on the results when switching from low level value to high level value. $s_Z(i)$ is the sign of the factor in run i and $R(i)$ is the corresponding result of the run.

$$e_Z = \frac{1}{2^{k-1}} \sum_{i=1}^{2^k} s_Z(i) \cdot R(i) \tag{4.1}$$

For DQN, Figure 4.1 shows the calculated effects of the factors A to H on the maximum network capacity and the 95% confidence intervals based on the validation set and the test set. The effects have been calculated according to Equation 4.1 using the results of all $k = 16$ runs in Table 4.2. While negative values indicate a negative effect when switching from low level value to high level value, the value itself in comparison to the other values indicates the significance of the factor.

The figure indicates that the configuration of the early stopping mechanism (A) has the most significant effect of the results, followed by the learning rate (C), the choice of optimizer (E), and size of the hidden layers (F). While most of the factors improve the results when switching from low level to high level value, this is not true for the discount factor (D) and the optimizer (E).

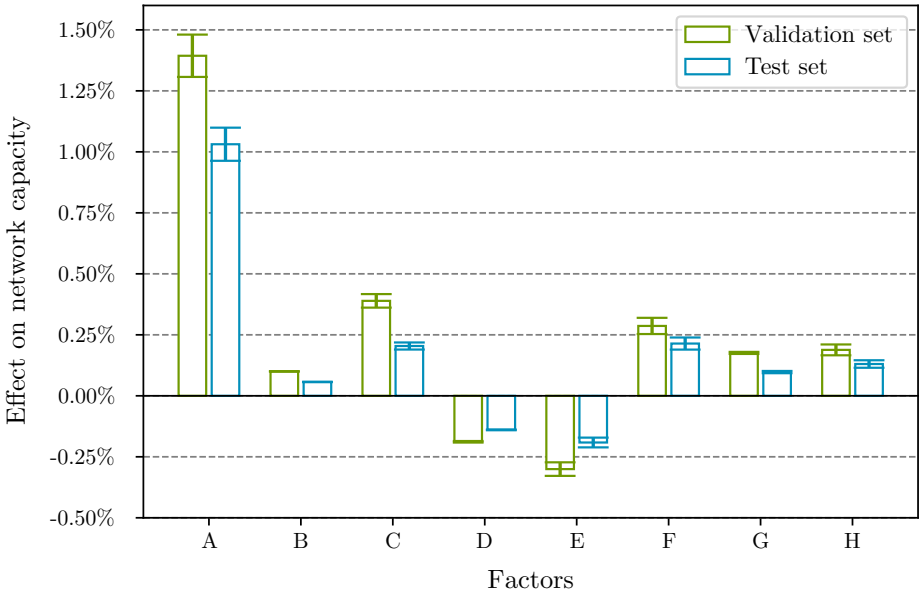

Figure 4.1. Mean effects and 95% confidence intervals for the eight factors A to H based on the results on the validation set and the test set.

Because of the significance of the early stopping configuration, a second experiment was conducted in order to identify an optimal value for the *epochs* hyperparameter. As the results indicated that a value $epochs > 4$ has a positive effect on the results, five different values ($epochs \in \{10, 12, 14, 16, 18\}$) were studied in the experiment. Figure 4.2 shows the results on the validation set and the test set when using the five different values for early stopping. The results, again, correspond to the percentage increase in network capacity from an initial baseline. The figure reveals that 14 epochs is the best choice for the configuration of the early stopping mechanism.

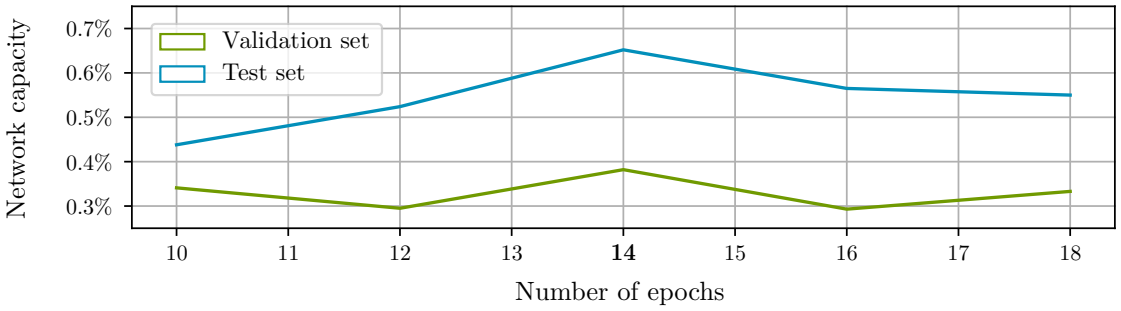

Figure 4.2. Results of the experiment on the validation set and the test set using 10, 12, 14, 16 and 18 epochs for the configuration of the early stopping mechanism.

In the next step, hyperparameters B to H were optimized in a way that maximizes the network capacities. For the number of episodes, 50 was chosen as a trade-off between the number of samples and a reasonable training time. As the agent was trained on a set of 24 different network topologies, $episodes = 50$ and $epochs = 14$ made for a total number of 1200 episodes and a minimum number of 16800 training samples, i.e., observations for the agent.

Figure 4.1 shows that the learning rate has the second most significant effect on the results and that $\alpha > 0.001$ increased the effect on the network capacity. Therefore, three values ($\alpha \in \{0.01, 0.025, 0.05\}$) were tested and $\alpha = 0.05$ led to the best results. The discount factor, on the other hand, only had a minor effect on the results. Three values ($\gamma \in \{0.9, 0.8, 0.75\}$) were tested and $\gamma = 0.8$ was chosen as the best one. For the gradient descent step, $opt =$ Adam was chosen as the use of RMSProp had a negative effect on the results.

For the size of the hidden layers, Figure 4.1 indicates a positive effect of a value $nodes > 40$. (Heaton 2017) states that the number of hidden neurons can be chosen with respect to the size of the input layer and output layer. As the size of the state representation is 76, two values 64 and 72 were chosen for testing. Moreover, the size of the second hidden layer was tested using 100% and 90% the size of the first hidden layer, respectively. Using a smaller size for the second hidden layer was expected to reduce overfitting as it enforces a further abstraction of the state representation. This results in four values ($nodes \in \{(64, 64), (64, 58), (72, 72), (72, 65)\}$) to test, of which $nodes = (72, 65)$ led to the best results.

Ultimately, Figure 4.1 shows that *target_update* and *batch_size* only had a minor positive effect on the results when using the higher level value, respectively. To keep things simple, the high level values $target\_update = 200$ and $batch\_size = 100$ were used for both hyperparameters.

In summary, the following hyperparameters were used for training the DQN model:

- $\sigma = 24$
- $\varepsilon_0 = 1.0$
- $\varepsilon_{\text{min}} = 0.1$
- $\varepsilon_{\text{dec}} \approx 0.001$
- $epochs = 14$
- $episodes = 50$
- $\alpha = 0.05$
- $\gamma = 0.8$
- $opt =$ Adam
- $nodes = (72, 65)$
- $target\_update = 200$
- $batch\_size = 100$

The model was evaluated on a number of 400 different scenarios from the validation set and the test set, respectively. It achieved an average maximum network capacity of **95.44%** on the validation set and **87.05%** on the test set.

4.3 Evaluation of Actor-Critic

The second model to be evaluated on both data sets was Actor-Critic. Section 3.4 covered the implementation details for Actor-Critic, which involved less tunable hyperparameters. More precisely, the Actor-Critic hyperparameters A to F correspond to the ones of DQN while G and H are not required as Actor-Critic utilizes neither a target network nor batch sampling.

Overall, this section closely follows Section 4.2. In the first step, an experiment was conducted in order to determine the effect of the hyperparameters A to F on the results of the Actor-Critic. With a number of 6 factors, a full factorial design stil requires a number of $2^6 = 64$ runs. Therefore, again, a fractional factorial design approach was followed in order to reduce the size of the experiment.

The DQN experiment proved 16 runs to be reasonable with respect to the time frame of this work. Therefore, a 2^{6-2}_{IV} factorial design (Myers et al. 2016) was chosen for Actor-Critic. Again, this is a resolution IV design which requires 16 runs in order to study a number of six factors. For factors A to D, all combinations are studied in the experiment while E and F are aliases with $E = \pm ABC$ and $F = \pm BCD$.

As with DQN, two discrete levels were specified for each of the six factors as shown in Table 4.3. Due to the fact that Actor-Critic is an on-policy method and was expected to be less sample efficient and more sensitive to the choice of hyperparameters, some low level and high level values differ from the ones used for DQN. Also, since the DQN experiment showed that a low number of early stopping epochs had a significant negative effect on the results, the low level value was increased for the Actor-Critic experiment.

Table 4.3. High and low value levels for the six Actor-Critic hyperparameters.

	A	B	C	D	E	F
	epochs	*episodes*	α	γ	*opt*	*nodes*
+	20	200	0.01	0.99	RMSProp	100
−	10	40	0.0001	0.8	Adam	40

Table 4.4 shows the design matrix for the 2^{6-2}_{IV} fractional design involving six factors and 16 runs. The table includes the results for every run on the validation set and the test set. Again, the results for each run correspond to the maximum percentage increase in network capacity from an initial baseline.

Table 4.4. Design matrix of the selected 2^{6-2}_{IV} fractional factorial design adopted from (Myers et al. 2016, Table 4.13) along with the results for both data sets.

Run	A	B	C	D	E	F	Val.	Test
1	+	+	+	+	+	+	2.094 %	1.530 %
2	+	+	+	+	−	−	2.032 %	1.518 %
3	+	+	−	−	−	+	1.639 %	1.301 %
4	+	+	−	−	+	−	0.183 %	0.404 %
5	+	−	+	−	−	+	2.443 %	1.759 %
6	+	−	+	−	+	−	2.187 %	1.625 %
7	+	−	−	+	+	+	1.699 %	1.412 %
8	+	−	−	+	−	−	0.801 %	0.738 %
9	−	+	+	+	−	+	2.005 %	1.410 %
10	−	+	+	−	+	−	2.245 %	1.689 %
11	−	+	−	+	+	−	0.464 %	0.658 %
12	−	+	−	−	−	+	0.855 %	0.696 %
13	−	−	+	+	+	−	2.151 %	1.658 %
14	−	−	+	−	−	+	2.014 %	1.454 %
15	−	−	−	+	−	+	1.699 %	1.358 %
16	−	−	−	−	+	−	0.464 %	0.658 %

As with DQN, Equation 4.1 was utilized in order to calculate the effects of the factors A to F on the maximum network capacity based on the validation set and the test set. This is shown in Figure 4.3 which resembles Figure 4.1 from the DQN experiment and also includes the 95% confidence intervals. The figure emphasizes that for Actor-Critic the learning rate (C) and the number of hidden nodes (F) had the most significant positive effect on the results. The number of episodes (B) and the use of the RMSProp optimizer (E) had a negative effect on the results.

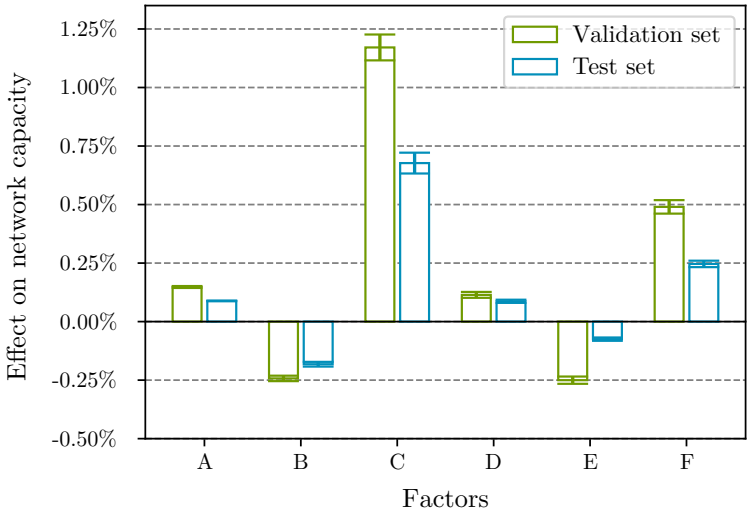

Figure 4.3. Mean effects and 95% confidence intervals of the six factors A to F based on the results on the validation set and the test set.

The experiment concluded that the learning rate requires careful consideration and that the effect of the early stopping mechanism was much smaller than with DQN, which can be attributed to the modified levels. A number of $epochs = 14$ proved to work with DQN and therefore, the configuration was maintained for Actor-Critic.

As the method cannot re-use samples from a memory, for the number of training episodes, a higher value of $episodes = 70$ was chosen in order to ensure that there were enough observations for the agent to be trained on. For the learning rate, three values ($\alpha \in \{0.005, 0.01, 0.025\}$) were tested and $\alpha = 0.025$ proved to be a reasonable choice. For the discount factor, two values ($\gamma \in \{0.85, 0.9\}$) were tested and $\gamma = 0.9$ led to better results. As with DQN, the hidden layers of the network were tested with 64 and 72 nodes as well as 100% and 90% the number of nodes for the second hidden layer, respectively. Again, $nodes = (72, 65)$ led to the best results. For the gradient descent step, Figure 4.3 shows that RMSProp, again, had a negative effect on the results. Therefore, $opt =$ Adam was also used for Actor-Critic.

In summary, the following hyperparameters were used for training the final model:

- $\sigma = 24$
- $\alpha = 0.025$
- $\gamma = 0.9$
- $opt =$ Adam
- $epochs = 14$
- $episodes = 70$
- $nodes = (72, 65)$

The model was evaluated on a number of 400 different scenarios from the validation set and the test set, respectively. It achieved an average maximum network capacity of **95.43%** on the validation set and **86.38%** on the test set. Compared to DQN, this is a slightly lower result.

4.4 Comparison to Supervised Learning

Sections 4.2 and 4.3 presented the absolute results for DQN and Actor-Critic. The significance of the results is debatable, because there was no way to determine the distance between the resulting network capacties and the actual maximum capacities. Especially for the large network topologies of the test set, it is expected that ground truth, on average, is much less than 100 % network capacity.

In order to assess the results of both DRL methods, this section presents a comparison between DRL, ground truth that has been computed by a brute force algorithm, and two supervised ML models that have been trained on the ground truth data. The ground truth as well as the ML results have been made available for the purpose of this work.

The brute force algorithm basically uses the occurring per-hop requirements of all streams (cf. Equation 3.4) and identifies an optimal SP configuration by trying out all four-value combinations of elements $\delta \in \Delta$. For the linear network scenarios used in the data sets of this work, the algorithm always identifies an optimal configuration that leads to the maximum network capacity.

This ground truth data could not only be utilized to assess the results of the DRL models trained in this work, but has also been used to train two different supervised ML models. The first model has been trained using random forests in order to predict the SP configuration for a given scenario. The model solves a classification task, which means that it outputs four discrete values from a range of 20 different values. This output corresponds to the configuration $\mathcal{C}$. The second model has been trained using a DNN with three fully-connected hidden layers. The model solves a regression task, which means that it predicts values of continuous range, which, again, correspond to the configuration $\mathcal{C}$. Hereafter, the two ML methods are also referred to as *classification* and *regression*, respectively.

Table 4.5. Ground truth and results of the DRL and supervised ML methods on the validation set and the test set in descending order. The results correspond to average network capacity.

	Ground Truth	DQN	Actor-Critic	Regression	Classification
Val.	97.28 %	**95.44 %**	**95.43 %**	92.79 %	92.10 %
Test	89.97 %	**87.05 %**	**86.38 %**	84.84 %	84.45 %

Table 4.5 shows the ground truth along with the resulting average network capacities for the DRL methods and the ML methods on the validation set and the test set. For the test set, the ground truth confirmed the assumption that the actual maximum capacity of the large networks is much less than 100 % on average.

The results show that the DRL methods, on average, outperformed both ML methods on the validation set and the more challenging test set. DRL was able to achieve network capacities that are fairly close to the maximum network capacities.

As the results were promising, it was further investigated how the size of the networks affected the performance of the DRL and ML models and which models were able to generalize beyond the network topologies they have been trained on. Therefore, Figure 4.4 groups the results on the validation set by the number of streams in the system, which is determined by the number of sensors and is closely related to the size of the network. The validation set includes scenarios with up to 560 streams (280 sensors). The results are shown for both DRL and both ML methods and correspond to the average network capacity relative to the ground truth for the validation set along with the 95% confidence intervals.

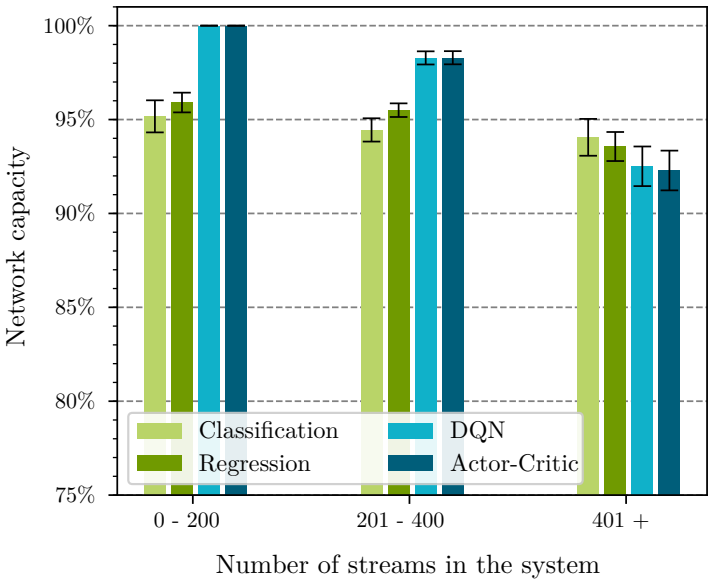

Figure 4.4. Results for DRL and ML on the validation set relative to the ground truth along with the 95% confidence intervals. Results are grouped by the number of streams in the system which is closely related to the size of the network.

The figure shows that for small scenarios up to at least 200 streams, both DRL models provided optimal results. Only for medium-sized scenarios with more than 400 streams in the system, supervised ML slightly outperformed DRL.

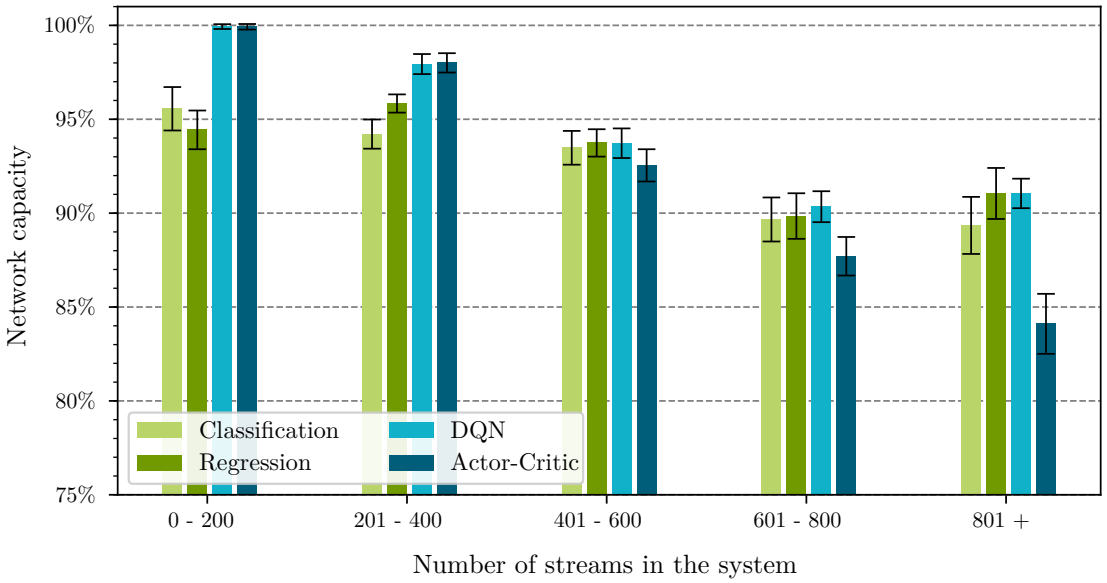

Figure 4.5. Results for DRL and ML on the test set relative to the ground truth along with the 95% confidence intervals. Results are also grouped by the number of streams in the system.

Figure 4.5 resembles Figure 4.4 and shows the results for both DRL and both supervised ML methods on the test set grouped by the number of streams in the system. The test set includes much larger scenarios with up to 1080 streams (540 sensors). The results, again, correspond to average network capacities relative to the ground truth for the test set and include 95% confidence intervals.

The figure confirms that DRL provided optimal results for small scenarios with up to at least 200 streams. It also reveals that the performance of Actor-Critic significantly dropped with the number of streams and the size of the network. This was not obvious in the results in Table 4.5, because most of the scenarios included in the test set are small or medium-sized. Lastly, the figure shows that DQN not only outperformed Actor-Critic on large scenarios, but also outperformed both ML methods on the test set. This proves that the DQN model is able to generalize well beyond the network topologies it has been trained on.

In summary, the results presented in this section prove that this work successfully applied DRL to the problem of SP configuration for linear industrial networks. DRL provided optimal results for small networks and was able to outperform two supervised ML models on small and medium-sized scenarios. Particularly the DQN model was able to outperform all other tested methods on the challenging test set including network scenarios with up to 1080 streams. Although Actor-Critic did not perform as well on large network scenarios, it is expected that further fine-tuning of the training environment and the hyperparameters can improve the results.

4.5 Variations on Priority Classes

This work successfully demonstrated that DRL can outperform supervised ML on the task of SP configuration for linear industrial networks. Besides the solid results, a major advantage of DRL methods over supervised ML is the level of flexibility.

Supervised learning requires ground truth data for the training, which often has to be generated using methods like brute force or linear programming. Generally, this is not only time-consuming, but the ground truth also loses its validity if a fundamental change is introduced to the environment. DRL, on the other hand, does not depend on ground truth data and can easily adjust to a changing environment.

This section completes this work by demonstrating the flexibility of the DRL approach. For this purpose, it introduces a variation to the number of priority classes used in the industrial network environment. Previously, both DQN and Actor-Critic have been trained on an environment that utilizes four priority classes for SP transmission selection. This also applies to the brute force algorithm and the ML methods introduced in Section 4.4.

Utilizing for example only three priority classes for time-sensitive traffic is a fundamental change to the dynamics of the environment. For ML, this requires the generation of new ground truth along with various adjustments to the ML method. For DRL, on the other hand, the change only requires an adjustment to the action space and to the output layer of the Q-Network or Actor-Critic network, respectively.

This provided the opportunity to study the performance of DQN and Actor-Critic when utilizing only two or three priority classes for SP transmission selection. To account for the smaller size of the action space, the number of epochs for early stopping was reduced to 12 for the training with two priority classes. Apart from that, all hyperparameters were chosen according to Sections 4.2 and 4.3, respectively.

Figure 4.6 shows the results of DQN and Actor-Critic on the validation set (a) and the test set (b) when utilizing four, three and two classes for SP transmission selection, respectively. For environments utilizing three or two classes, there were no ground truth data available. Therefore, results in the figure correspond to the average network capacity relative to the total number of streams in the system, not to the ground truth. The figure also includes 95% confidence intervals.

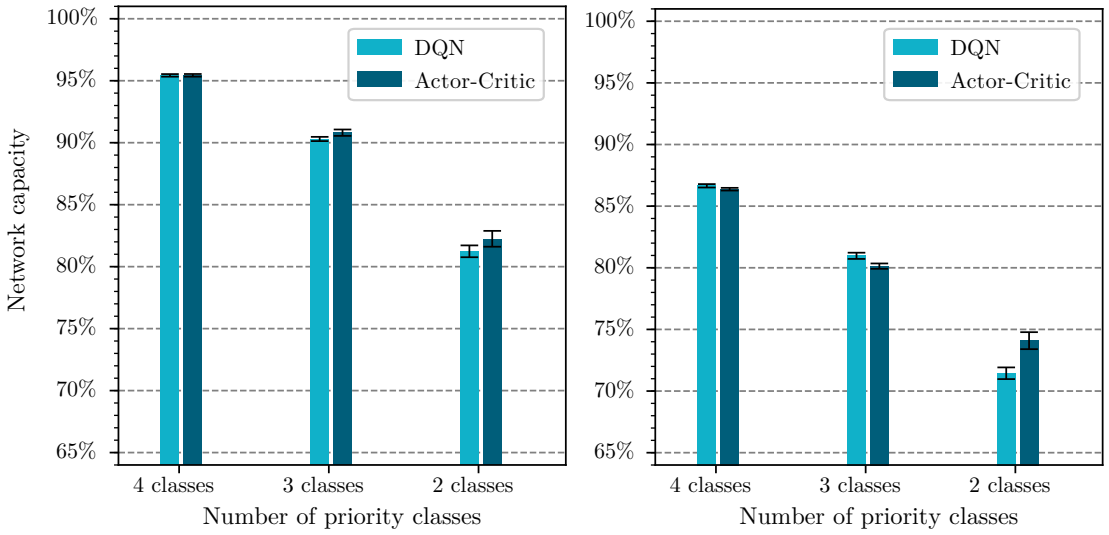

Figure 4.6. Average network capacities for DQN and Actor-Critic on the validation set (a) and the test set (b) when utilizing four, three, or two priority classes

The figure shows that, even when utilizing only two priority classes, DRL still ensured network capacities above 70 %. It is also worth noting that Actor-Critic was able to outperform DQN on both data sets when only two priority classes were used.

Overall, this further emphasizes the potential of DRL methods for the configuration of TSN in dynamic industrial networks. Although this work only studied the configuration of SP, the results of Section 4.5 indicate that DRL can also be applied to network environments with different dynamics. It is expected that this also applies to different configuration tasks beyond basic SP transmission selection.

5
Conclusion

This work studied TSN configuration for dynamic industrial networks utilizing two different DRL methods. For this purpose, a RL environment was proposed for simulation and configuration of industrial networks using SP transmission selection. Two fundamentally different DRL methods, DQN and Actor-Critic, were implemented and applied to the SP configuration problem.

Evaluating the two models on two data sets of different complexity showed that DQN and Actor-Critic were able to achieve optimal results on small networks and were able to outperform supervised ML on small and medium networks with up to 400 streams. On networks of larger size, DQN was still able to outperform Actor-Critic and the supervised classification model. In absolute numbers, the DRL models trained in this work outperformed supervised ML on both data sets and provided results that were much closer to the ground truth.

Ultimately, this work investigated the flexibility of the DRL approach and demonstrated that DRL not only provides reasonable results on the task of SP configuration, but also adapts to changing dynamics of its environment. For this purpose, DQN and Actor-Critic were trained on two environments that utilized only two and three priority classes for time-sensitive traffic, respectively. While supervised ML would have required the generation of new ground truth data for each of the environments, DRL adapted to the changes and proved to be applicable to network environments with different dynamics.

Besides the comparison between DRL and supervised ML, this work aimed to study the differences between on-policy and off-policy learning with regard to implementation and application to the SP configuration problem. For this purpose, DQN and Actor-Critic were chosen to represent the two fundamentally different ways of learning. This work utilized the baseline implementations adopted from (Mnih et al. 2013) and (Sutton et al. 2018), respectively. Yet, various enhancements to the baseline implementations have been proposed in the last years.

DQN is based on Q-Learning which (van Hasselt 2010) proved to suffer from an overestimation bias due to the use of the max-operator in Equation 2.3. To address this issue, van Hasselt proposed Double Q-Learning which was later implemented as Double DQN (DDQN, van Hasselt et al. 2015). Regarding the architecture of the Q-Network, (Wang et al. 2015) proposed a Dueling DQN architecture with two streams of computation to approximate both a state-value and an action-value which corresponds to the advantage of each action over the other actions in the current state. The approach bears a resemblance to the Actor-Critic architecture used in this work and was also shown to be successfully combinable with DDQN. Regarding sample-efficiency, the baseline DQN implementation in this work uniformly samples batches from memory in order to train the Q-Network. (Schaul et al. 2015) proposed a method of prioritized experience replay which samples observations from which the agent can learn the most with higher probability. The combination of Dueling DQN, DDQN, and prioritized sampling achieved state-of-the-art results on the Atari 2600 benchmark (Bellemare et al. 2013).

There are also various enhancements to the baseline Actor-Critic. (Mnih et al. 2016) proposed the Asynchronous Advantage Actor-Critic (A3C) which approximates a policy as well as the advantage of each action over the other actions in the current state. It also enables parallelization and drastically reduces training time and hardware requirements. Proximal Policy Optimization (PPO, Schulman et al. 2017) provides increased stability to the policy gradient method by using multiple epochs of gradient ascent to perform an adjustment to the policy. Deep Deterministic Policy Gradient (DDPG, Lillicrap et al. 2015) combines the advantages of DQN and Actor-Critic when using a continuous action space. Although this work proposed an environment with a discrete action space, PPO or DDPG could be used to adjust the guaranteed per-hop latency of the priority classes in a continuous range instead of using steps of $10\,\mu s$.

Ultimately, (Hessel et al. 2017) demonstrated that a combination of multiple techniques and enhancements to the baseline methods can lead to significant improvements over the individual results. It is expected that the utilization of more advanced methods based on DQN and Actor-Critic can further improve the results on the challenging SP configuration task and can also be applied to other configuration tasks in the domain of TSN. This was out of scope for this work but offers great potential for further research.

List of Figures

List of Tables

Bibliography

Technical Reports and Standards

Sing, J., & Soh, B. (2005). TCP New Vegas: Improving the Performance of TCP Vegas Over High Latency Links, In *Fourth IEEE International Symposium on Network Computing and Applications*.

IEEE 1588. (2008). IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*.

IEEE 802.1Qav. (2009). IEEE Standard for Local and Metropolitan Area Networks – Virtual Bridged Local Area Networks – Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)*.

Henderson, T., Floyd, S., Gurtov, A., & Nishida, Y. (2012). *The NewReno Modification to TCP's Fast Recovery Algorithm* (RFC No. 6582). Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc4180

IEEE 802.1Qbv. (2016). IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks – Amendment 25: Enhancements for Scheduled Traffic. *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014))*.

Specht, J., & Samii, S. (2016). Urgency-Based Scheduler for Time-Sensitive Switched Ethernet Networks, In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*.

IEEE 802.1CB. (2017). IEEE Standard for Local and Metropolitan Area Networks – Frame Replication and Elimination for Reliability. *IEEE Std 802.1CB-2017*.

IEEE 802.1Qci. (2017). IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks – Amendment 28: Per-Stream Filtering and Policing. *IEEE Std 802.1Qci-2017 (Amendment to IEEE Std 802.1Q-2014))*.

Farkas, J. (2018). IEEE 802.1 Time-Sensitive Networking (TSN) Task Group (TG) Overview.

IEEE 802.1Q. (2018). IEEE Standard for Local and Metropolitan Area Network – Bridges and Bridged Networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*.

IEEE 802.1Qdd. (2018). IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks – Amendment: Resource Allocation Protocol.

IEEE 802.3. (2018). IEEE Standard for Ethernet. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*.

Grigorjew, A., Metzger, F., Hoßfeld, T., Specht, J., Götz, F.-J., Schmitt, J., & Chen, F. (2020). *Technical Report on Bridge-Local Guaranteed Latency with Strict Priority Scheduling* (tech. rep.). Institut für Informatik.

IEEE 802.1Qcr. (2020). IEEE Draft Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks – Amendment: Asynchronous Traffic Shaping. *IEEE P802.1Qcr/D2.1, February 2020*.

Articles and Books

Bellman, R. (1957). A Markovian Decision Process. *Indiana Univ. Math. J.*, *6*, 679–684.

Jordan, M. I. (1986). Serial order: a parallel distributed processing approach. Technical report, June 1985-March 1986.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning Representations by Back-Propagating Errors. In *Neurocomputing: Foundations of Research* (pp. 696–699). Cambridge, MA, USA, MIT Press.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, *2*(4), 303–314.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, *4*, 251–257.

Watkins, C., & Dayan, P. (1992). Technical Note: Q-Learning. *Machine Learning*, *8*, 279–292. https://doi.org/10.1007/BF00992698

White, D., Sofge, D., & Thrun, S. (1992). The Role Of Exploration In Learning Control.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.*, *8*(3–4), 229–256. https://doi.org/10.1007/BF00992696

Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, *6*(6), 861–867. https://doi.org/"https://doi.org/10.1016/S0893-6080(05)80131-5

Boyan, J. A., & Littman, M. L. (1994). Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach (J. D. Cowan, G. Tesauro, & J. Alspector, Eds.). In J. D. Cowan, G. Tesauro, & J. Alspector (Eds.), *Advances in Neural Information Processing Systems 6*. Morgan-Kaufmann.

LeCun, Y., & Bengio, Y. (1995). Convolutional Networks for Images, Speech, and Time-Series.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Comput.*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Ferrá, H., Lau, K., Leckie, C., & Tang, A. (2003). Applying Reinforcement Learning to Packet Scheduling in Routers.

Hinton, & Osindero. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural computation*, *18*, 1527–54.

Kalyanakrishnan, S., & Stone, P. (2007). Batch reinforcement learning in a complex domain. https://doi.org/10.1145/1329125.1329241

van Hasselt, H. (2010). Double Q-learning (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, & A. Culotta, Eds.), 2613–2621.

Hinton, Deng, L., Yu, D., Dahl, G. E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., & Kingsbury, B. (2012). Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, *29*(6), 82–97.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks, In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, Lake Tahoe, Nevada, Curran Associates Inc.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, *47*, 253–279. https://doi.org/10.1613/jair.3912

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing Atari with Deep Reinforcement Learning. *CoRR*, *abs/1312.5602*arXiv 1312.5602. http://arxiv.org/abs/1312.5602

Montgomery, D. C. (2013). *Design and Analysis of Experiments* (8th ed). John Wiley.

Kingma, D., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, *521*, 436–44. https://doi.org/10.1038/nature14539

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *CoRR*.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized Experience Replay, arXiv 1511.05952.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-Dimensional Continuous Control Using Generalized Advantage Estimation.

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning, arXiv 1509.06461.

Wang, Z., de Freitas, N., & Lanctot, M. (2015). Dueling Network Architectures for Deep Reinforcement Learning. *CoRR*, *abs/1511.06581*arXiv 1511.06581. http://arxiv.org/abs/1511.06581

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning* [http://www.deeplearningbook.org]. MIT Press.

Gu, S., Lillicrap, T. P., Sutskever, I., & Levine, S. (2016). Continuous Deep Q-Learning with Model-based Acceleration. *CoRR*, arXiv 1603.00748. http://arxiv.org/abs/1603.00748

Li, W., Zhou, F., Meleis, W., & Chowdhury, K. (2016). Learning-Based and Data-Driven TCP Design for Memory-Constrained IoT, In *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*.

Lin, S., Akyildiz, I. F., Wang, P., & Luo, M. (2016). QoS-Aware Adaptive Routing in Multi-layer Hierarchical Software Defined Networks: A Reinforcement Learning Approach, In *2016 IEEE International Conference on Services Computing (SCC)*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning.

Myers, R., Montgomery, D., & Anderson-Cook, C. (2016). *Response Surface Methodology: Process and Product Optimization Using Designed Experiments* (Vol. 705).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., … Dean, J. (2016). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, arXiv 1609.08144.

Feki, S., Zarai, F., & Belghith, A. (2017). A Q-learning-based Scheduler Technique for LTE and LTE-Advanced Network, In *WINSYS*.

Heaton, J. (2017). *The Number of Hidden Layers*. Retrieved July 6, 2020, from https://www.heatonresearch.com/2017/06/01/hidden-layers.html

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning, arXiv 1710.02298.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *CoRR*, *abs/1707.06347*arXiv 1707.06347. http://arxiv.org/abs/1707.06347

Stampa, G., Arias, M., Sanchez-Charles, D., Muntés-Mulero, V., & Cabellos, A. (2017). A Deep-Reinforcement Learning Approach for Software-Defined Net-

working Routing Optimization. *CoRR*, *abs/1709.07080*arXiv 1709.07080. http://arxiv.org/abs/1709.07080

Kim, D., Lee, T., Kim, S., Lee, B., & Youn, H. (2018). Adaptive Packet Scheduling in IoT Environment Based on Q-learning. *Procedia Computer Science*, *141*, 247–254.

Kong, Y., Zang, H., & Ma, X. (2018). Improving TCP Congestion Control with Machine Intelligence, In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, Budapest, Hungary, Association for Computing Machinery. https://doi.org/10.1145/3229543.3229550

Ruffy, F., Przystupa, M., & Beschastnikh, I. (2018). Iroko: A Framework to Prototype Reinforcement Learning for Data Center Traffic Control. *CoRR*, arXiv 1812.09975. http://arxiv.org/abs/1812.09975

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (Second). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

Xu, Z., Tang, J., Meng, J., Zhang, W., Wang, Y., Liu, C. H., & Yang, D. (2018). Experience-driven Networking: A Deep Reinforcement Learning based Approach. *CoRR*, *abs/1801.05757*arXiv 1801.05757. http://arxiv.org/abs/1801.05757

Jay, N., Rotman, N., Godfrey, B., Schapira, M., & Tamar, A. (2019). A Deep Reinforcement Learning Perspective on Internet Congestion Control (K. Chaudhuri & R. Salakhutdinov, Eds.). In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th International Conference on Machine Learning*, Long Beach, California, USA, PMLR.

Kuang, N. L., Leung, C. H. C., & Sung, V. W. K. (2019). Stochastic Reinforcement Learning. *CoRR*, *abs/1902.04178*arXiv 1902.04178. http://arxiv.org/abs/1902.04178