LMU LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME
UND DATA MINING

Master Thesis
in Computer Science

EXPERIENCE BASED EXPLAINABLE AI

SEBASTIAN SCHÄFFLER

Aufgabensteller:	AUFGABENSTELLERNAME
Betreuer:	VOLKER TRESP
Abgabedatum:	30.09.2023

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.
This paper was not previously presented to another examination board and has not been published.

MUNICH, 30.09.2023

.....
SEBASTIAN SCHÄFFLER

Abstract

Why doesn't a child touch the hot stove top? Why doesn't a person ...? Why doesn't a reinforcement learning agent...? OR If you ask a child why it does not touch the hot stove top, the child will probably tell you what behaviour lead to this behaviour.. oder so ähnlich

Contents

Chapter 1

Introduction

1.1 Overview

Systematic approach of going from a simple classification problem to a reinforcement learning problem to a more complex reinforcement learning problem.

1.2 Motivation

1.3 Problem Statement

1.4 Research Questions and Objective

1.5 Thesis Contribution

- Most are about features. But we talk about experiences

1.6 Research Milestones

1.7 Structure of the Thesis

Chapter 2

Foundations

To understand the techniques applied in the implementation and the findings gathered in the experiments in this thesis, some key concepts have to be elucidated first. The objective of this chapter is to outline these concepts and to give the reader the foundational knowledge required to follow the subsequent sections.

2.1 Machine Learning

Machine learning (ML) is a sub-field of artificial intelligence (AI) that aims at enabling computers to learn without being explicitly programmed on the task at hand. This is done by training computer programs on data, which allows them to identify patterns and make predictions by approximating the data [3].

This area of research can be broken down into three primary paradigms: supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the learning is performed by looking at previously labeled data. Subsection 2.1.1 discusses this type of learning. As will be described in Section 2.1.2 in more detail, unsupervised learning is learning from unlabeled data exclusively. Reinforcement learning approaches problems where an agent interacts with an environment and gains rewards. This plays an essential role in this thesis and thus will be presented in its own section in Section 2.2.

A method for implementing the paradigms of ML that has found a wide range of applications in recent times is deep learning, explained in Subsection 2.1.3.

For a rough overview of the concepts introduced, refer to Figure 2.1.

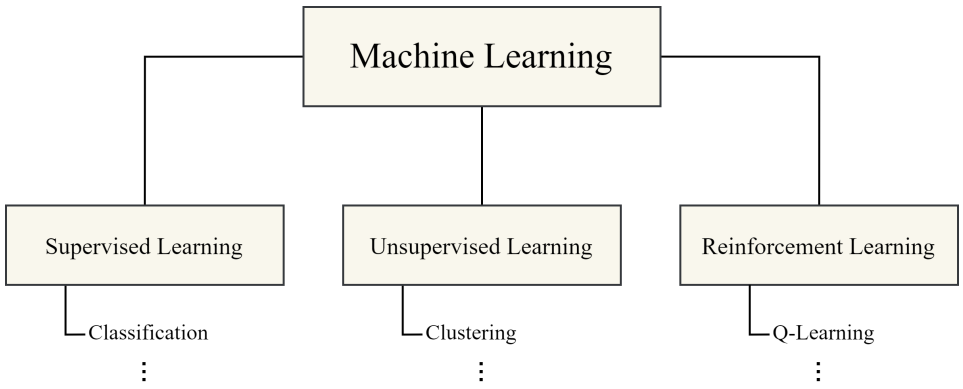


Figure 2.1: Overview of machine learning paradigms with implementation examples.

2.1.1 Supervised Learning

As outlined in the introduction in Section 2.1, the ML paradigm of supervised learning is concerned with learning from labeled data. To perform this kind of learning, the learner receives a training dataset of n labeled examples on which it is trained and then makes predictions on unseen data. This kind of training finds application in the fields of regression, classification, and other problems [41].

The training dataset D formalized in Equation 2.1 is comprised of training samples $(\mathbf{x}_i, y_i)$ where $\mathbf{x}_i$ refers to the i^{th} data point and y_i is the label of the i^{th} data point.

$$D = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\} \subseteq \mathbb{R}^d \times C \tag{2.1}$$

In the above equation, $\mathbb{R}^d$ refers to the d-dimensional feature space and C to the label space. In a classification scenario where K classes are possible, the label space C can be defined as $C = \{1, 2, ..., K\}$. For a regression problem where salaries are predicted $C = \mathbb{R}$.

It is assumed that the training dataset D is drawn from an unknown distribution $\mathcal{P}$, meaning $(\mathbf{x}_i, y_i) \sim \mathcal{P}$. The goal of the learning process is then to find a hypothesis function h shown in Equation 2.2 that is able to predict the label y of a new input sample $\mathbf{x}$ so that the function model's $\mathcal{P}$ and thus $h(\mathbf{x}) \approx y$.

$$h \to \mathbb{R}^d \times C \tag{2.2}$$

In an example for a multi-class classification scenario, the dataset $D = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ consists of pairs where $\mathbf{x}_i$ is a feature vector representing

the i^{th} image of a handwritten digit, and y_i is the corresponding label which can be $C = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ [12]. The features could include pixel intensities, or more abstract features obtained through some pre-processing. The goal is to learn a hypothesis function h that can correctly classify a new image of a handwritten digit that is not in the training set.

2.1.2 Unsupervised Learning

In the unsupervised learning setting, the main difference from supervised learning is that there are no labels y_i associated with each data point $\mathbf{x}_i$. We have a training dataset D which is formalized in Equation 2.3 and is comprised solely of n training samples $\mathbf{x}_i$.

$$D = \{\mathbf{x}_1, ..., \mathbf{x}_n\} \subseteq \mathbb{R}^d \tag{2.3}$$

As in the supervised learning case, $\mathbb{R}^d$ refers to the d-dimensional feature space. Different from supervised learning is the absence of the label space C. In the unsupervised learning case, the data points do not have labels associated with them.

Just as in supervised learning, it is assumed that the training dataset D is drawn from an unknown distribution $\mathcal{P}$, so $\mathbf{x}_i \sim \mathcal{P}$. The goal of the unsupervised learning process, however, differs from that of supervised learning. Rather than predicting a label, the objective is to learn the underlying structure or distribution of the data, to find hidden patterns or relationships, or to reduce the dimensionality of the data.

An example of a goal in unsupervised learning could be finding a function g (shown in Equation 2.4) that transforms the data into a lower dimensional space while preserving as much of the original data structure as possible, as in principal component analysis [23].

$$g : \mathbb{R}^d \to \mathbb{R}^m, m < d \tag{2.4}$$

K-means Clustering Another possible goal could be finding a partitioning of the data into K different clusters, as in K-means clustering [38]. In K-means, the algorithm aims to partition a dataset into K clusters by minimizing the within-cluster sum of squares. It starts by initializing K centroids and iteratively updates them in two steps: First, assign each data point to the nearest centroid, and second, recalculate the centroids as the mean of the assigned points. The algorithm converges when centroids stabilize or a set number of iterations is reached. Because of the distance-based assignment, K-means tends to produce sharp-edged transitions between clusters. In this

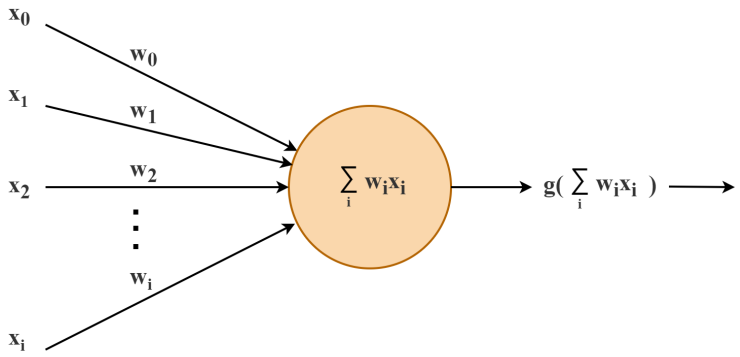


Figure 2.2: Structure of a perceptron.

context, one might define a function f (shown in Equation 2.5) that assigns each data point to one of K clusters.

$$f : \mathbb{R}^d \to \{1, ..., K\} \tag{2.5}$$

2.1.3 Deep Learning

To understand the concept of deep learning, *artificial neural networks* (ANNs) have to be introduced first. ANNs are systems inspired by the structure of the brain. By looking at examples, these systems are able to learn and solve tasks without being programmed with task-specific rules (cf. Section 2.1). They consist of a collection of interconnected nodes, called *artificial neurons* or *perceptrons*, which mimic the neurons in a brain. Each connection, like a synapse, can pass signals to other perceptrons. Here, a signal is a real number and the output of a perceptron is calculated by a non-linear function of the sum of the input signals. Edges have a certain weight, which is adjusted in the learning process. Figure 2.2 shows the structure of a perceptron.

The perceptron shown receives the input signals x_0 to x_i with their weights w_0 to w_i and calculates the output signal by summing the weighted signals, which are passed on to a so-called *activation function* g, the output signal is calculated. The output y of a perceptron can be described by the following Equation 2.6:

$$y = g\left(\sum_i w_i x_i\right) \tag{2.6}$$

Activation functions are functions that decide whether an artificial neuron "fires", i.e. is activated, or not. They are also useful to limit the output signal

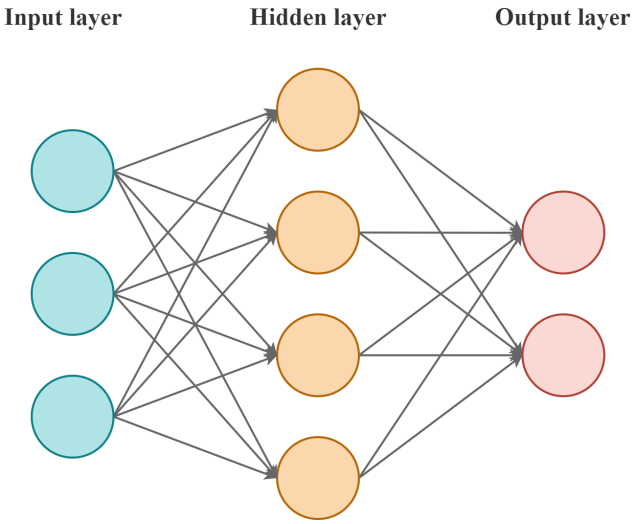


Figure 2.3: Artificial neural network.

of a perceptron to a certain range of values. Depending on the application, different activation functions can be used, but they must meet certain requirements. An activation function should be non-linear, piece-wise linear, or a step function. Normally, activation functions are also monotonically increasing. Frequently used functions are the jump function, sigmoid, tanh, or the Rectified Linear Unit (ReLU).

Connecting multiple artificial neurons creates an ANN, as shown in Figure 2.3. Here, neurons that are to receive the same input signals are grouped into layers. Importantly, all neurons in a layer must have a connection with all neurons in the layer following it. Input signals move, starting at the input layer, through a hidden layer, to the output layer, where the output of the ANN is generated.

If an ANN consists of multiple hidden layers, it is called a *deep neural network* (DNN), as illustrated in Figure 2.4. In this case, we talk about *deep learning*. The layers can have a different number of perceptrons. More perceptrons and layers allow for a higher number of hierarchical abstractions that can be learned from input data. This enables the learning of higher-order functions for more complex domains.

Learning typically occurs through training on datasets, where a cost or loss function quantifies the discrepancy between the expected and actual outputs for each dataset. The output of the cost function indicates how good the result of the ANN was and is used to subsequently adjust the weights across the network. By iteratively processing a large number of labeled data the ANN

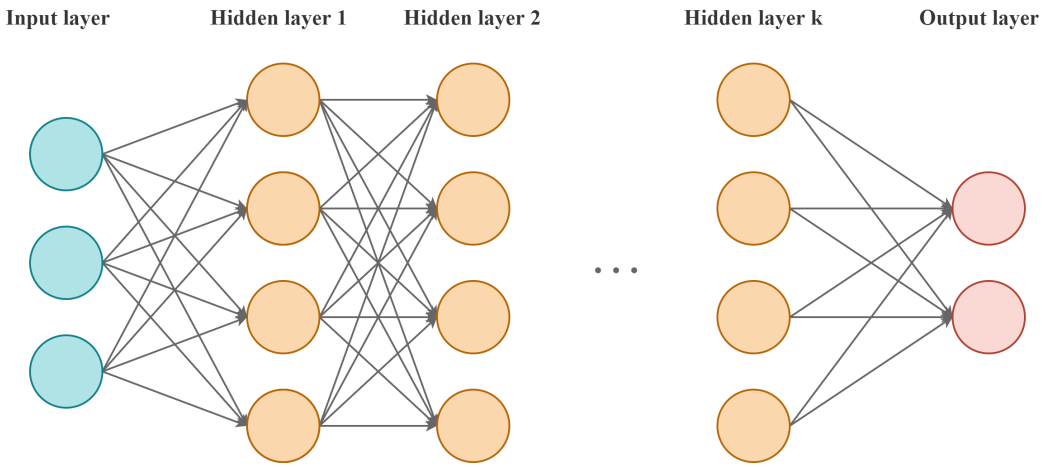


Figure 2.4: Deep neural network.

learns and as such, enhances its predictive capabilities.

2.2 Reinforcement Learning

Alongside supervised learning and unsupervised learning, reinforcement learning (RL) is the third major paradigm of machine learning. RL addresses scenarios where an agent continuously interacts with an environment by taking actions. The environment responds to the actions taken by the agent by presenting the agent with an updated state and a reward represented by a numerical value. The objective of the agent is to interact with the environment and derive an *optimal policy* so that the actions taken maximize the cumulative reward gained over time. RL applications range from simple game-like environments where an agent has to move across a grid with obstacles to more complex scenarios in industrial control problems [46, 26].

2.2.1 Reinforcement Learning in General

Typically, the behavior of an agent interacting with an environment through actions, in its most basic form, is modeled as a *Markov decision process* (MDP), a formalization of a decision process in which sequential decisions are made in discrete time steps t and each action taken influences not only the immediate rewards but also the states that follow. This kind of modeling allows for an application of RL techniques via dynamic programming [30, 50]. A MDP is a 4-tuple (S, A, P, R), where S is a set of states, A is a set of actions,

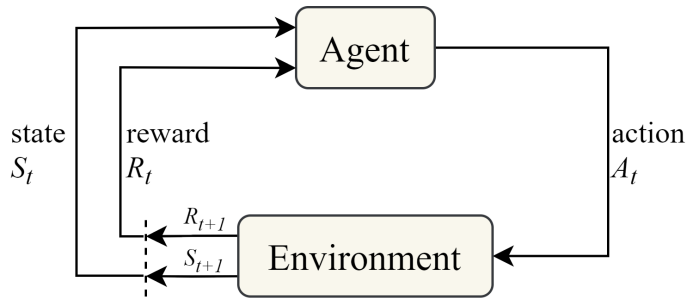


Figure 2.5: The interaction of an agent with an environment in a Markov decision process (adapted from [46]).

$P(s_{t+1}|s_t, a_t) \in [0, 1]$ is the probability of reaching state $s_{t+1} \in S$ when taking action $a_t \in A$ in state $s_t \in S$ and $R(s_t, a_t) \in \mathbb{R}$ is a reward function. A policy $\pi : S \to A$ that may also be stochastic such that $\pi(a_t|s_t) \in [0, 1]$ is then a representation of the behavioral strategy of an agent. The agent-environment interaction described and construed as an MDP is visualized in Figure 2.5.

To formalize the objective of maximizing the cumulative reward, the notion of the *expected discounted return* G is introduced in Equation 2.7. The discount factor $\gamma \in [0, 1]$, which is usually close to 1, defines the value of future rewards at the time step t, with $\gamma = 0$ meaning that only immediate rewards are taken into account.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.7}$$

Further, the *state-value function* of a state s under a certain policy π introduced in equation 2.8 represents the expected return when starting from state s and then following policy π.

$$V^{\pi}(s) = \mathbb{E}\left[G_t|S_t = s\right], \forall s \in S \tag{2.8}$$

A policy that from any given state selects the action that maximizes the state-value function is called *optimal policy* π^*.

In addition to the state-value function, the *action-value function* Q is defined in equation 2.9. This function denotes the expected return when starting from state s, taking action a, and following policy π afterward, thus learning this function is a crucial part of RL. Often, the action-value function is also referred to as the *Q-function*, and the result of this function is referred to as the *Q-value* of the state-action pair (s, a).

$$Q^{\pi}(s,a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right] \tag{2.9}$$

Q-Learning One approach to learning the Q-function of an RL environment is the *Q-learning algorithm*. This algorithm is a form of *Temporal Difference (TD) learning*, where the Q-values are updated using the difference between the current Q-value and the estimated future Q-value. For each time step t, in which the agent takes an action a_t, the Q-value is updated by the update rule stated in equation 2.10.

$$Q(s_t,a_t) = (1-\alpha)\cdot Q(s_t,a_t) + \alpha \cdot \Big[R_t + \gamma \cdot \max_a Q(s_{t+1},a) - Q(s_t,a_t)\Big] \tag{2.10}$$

In the update rule, the parameter $\alpha \in (0,1]$, the *learning rate* is introduced. This parameter controls the extent to which the new Q-value estimate is adopted to update the current Q-value. To ensure both exploration of the environment and exploitation of known Q-values, strategies like the ϵ-greedy method are employed. This involves the agent occasionally taking a random action (exploration) and otherwise selecting the action with the highest Q-value (exploitation). In its simplest form, the result of the Q-learning algorithm is a Q-table, a table that stores the Q-values for each possible state-action pair. The optimal policy π^* is then derived by always selecting the action a that maximizes $Q(s,a)$ for each state s.

2.2.2 Characteristics of Reinforcement Learning Settings

In order to understand the terminus used in the RL jargon, some key characteristics relevant to this thesis are described in this section.

Episodic and Continuing Tasks Episodic tasks involve the agent interacting with the environment through a sequence of episodes. In each episode, the agent starts at a particular state and then takes actions until some terminal state, or in some cases, a predefined maximum amount of time steps the agent is allowed to take, is reached, at which point the episode terminates. An example of episodic tasks is games, where usually the objective is to attain some kind of goal. Upon reaching this goal, the game concludes, signifying the end of the episode.

In continuing tasks, on the other hand, an agent starts from a given state and then continuously interacts with the environment, without there

being a terminal state. It is basically a never-ending process where the agent constantly interacts with the environment. Consider the example of a heat control system. Unlike games where there is a defined end, this system continuously collects and processes data to update the energy input required for maintaining or reaching a certain temperature.

Discrete and Continuous Spaces In RL problems, the environment and the way an agent interacts with the environment can be divided into discrete and continuous spaces. The action space, as well as the state space, can be either discrete or continuous. In discrete spaces, the possible states, actions, or both are countable and distinct. An advantage of discrete spaces is that it is relatively simple to find optimal policies using methods such as dynamic programming described in section 2.2.3.

In contrast, continuous spaces consist of spaces that are an uncountable set of states or actions. Thus, the number of actions or states is infinite. The resulting higher complexity is often tackled by employing techniques such as discretization or deep learning approaches like the neural network-based approach described in section 2.2.4.

Online and Offline Learning In an online RL setting, the agent learns a policy through direct interactions with the environment. As it navigates, it collects experiences i.e. states, actions, rewards, and state transitions, thus facilitating a dynamic learning process, allowing the agent to adapt and refine its policy based on immediate feedback.

Conversely, offline RL operates on a more static dataset. Experiences are first sampled from an environment, and this collection process doesn't necessarily involve an RL agent. For instance, the data might come from human experts or other sources. Once gathered, this data serves as the foundation to train an RL agent, enabling it to learn without direct environment interaction.

On-Policy and Off-Policy Learning In an on-policy setting, the action-value function $Q^{\pi}(s, a)$ is learned by taking actions recommended by the current policy π. As the agent interacts with the environment, this policy is steadily updated based on the feedback received, ensuring that the learning is directly aligned with the agent's current strategy.

For the off-policy setting, on the other hand, this function is learned by taking actions suggested by a different policy. This could be a random policy, a previously learned policy, or any other policy. This approach allows for greater flexibility, as the agent can learn from experiences generated by various strategies. An example of an off-policy learning

algorithm is the Q-learning algorithm introduced in chapter 2.2.1, as it updates its Q-values based on the optimal action, regardless of the exploration policy in use.

Model-Based and Model-Free Approaches In a model-based approach, the agent seeks to understand and internalize the dynamics of the environment. This involves creating an internal model that predicts state transitions and rewards. As the agent interacts with the environment, this model is refined and updated, allowing the agent to plan ahead and make decisions based on its predictions.

Conversely, in a model-free approach, the agent doesn't attempt to understand the underlying dynamics of the environment. Instead, it focuses on learning the value or policy directly from its interactions, without relying on an internal model of the environment. An example of a model-free method is the Q-learning algorithm mentioned in chapter 2.2.1, which learns action values directly from experiences without constructing a model of the environment.

Deterministic and Stochastic Environments In deterministic RL environments, given a certain state and an action, the resulting next state and reward are always the same for the given state-action pair. This predictability allows agents to plan actions with certainty, knowing the exact outcomes of their decisions.

In contrast, stochastic RL environments introduce an element of randomness. Even when an agent takes the same action from a particular state, the resulting next state or reward might differ due to inherent uncertainties of the environment. An example of this can be seen in many real-world scenarios, like stock market predictions or autonomous vehicles, where despite similar initial conditions, outcomes can vary due to multiple unpredictable factors.

2.2.3 Dynamic Programming Approaches

In general, dynamic programming (DP) refers to algorithmic methods that solve optimization problems by breaking them down into simpler sub-problems [8]. Given a perfect model of the environment as an MDP, DP can be used to derive optimal policies in RL [46]. A typical DP algorithm for RL revolves around two primary steps: policy evaluation and policy improvement. These steps are iteratively executed in succession until convergence to an optimal policy is achieved.

In the first step, the current policy π is evaluated by computing the state-value function V^{π}. For the policy improvement step, the question of whether there is a better action $a! = \pi(s)$ for a given state s is asked. To answer this question, $Q^{\pi}(s, a)$ is calculated. If there is an action a for a state s, for which $Q^{\pi}(s, a) > V^{\pi}(s)$, it is better to change the action to a.

The foundation for this iterative process is the *policy improvement theorem* [8]. According to this theorem, a policy π' is deemed superior to policy π and thus replaces it if and only if the condition in Equation 2.11 holds true for all states s in the state space S.

$$\pi' \geq \pi \quad \Longleftrightarrow \quad V^{\pi'}(s) \geq V^{\pi}(s) \quad \forall s \in S \tag{2.11}$$

The DP approach described is also often referred to as *policy iteration*. Another DP approach is the so-called *value iteration*. Unlike policy iteration, value iteration combines the two steps of policy evaluation and policy improvement into a single update. First, the value function V gets initialized arbitrarily, and then, in each iteration, the value of V gets updated for each state using the *Bellman optimality equation*, displayed as Equation 2.12.

$$V(s) = \max_{a} \sum_{s'} P(s'|s, a) \left[R(s, a, s') + \gamma V(s')\right] \tag{2.12}$$

Where $P(s'|s, a)$ is the probability of transitioning to state s' while in state s using action a. The update iterations are continued until the value function change between the iterations has reached a defined threshold.

Given the iterative methodology of DP and its requirement to evaluate every potential state, classical DP frequently becomes impractical for environments with large or continuous state spaces, leading to a substantial increase in computational complexity [52].

2.2.4 Neural Network-based Value Function Approximation

In many real-world problems, the state or action space is too large or continuous, making it impractical to store and update values for each possible state-action pair. This necessitates the use of function approximation to generalize across states and actions. Due to their ability to approximate complex, non-linear functions, ANNs have emerged as a popular choice for this task.

When using an ANN for function approximation in RL, the network takes in a state, or state-action pairs on the input layer and outputs the estimated

value of that state or state-action pair. One advantage of using an ANN with regard to tasks with high dimensional state spaces is that an ANN is able to generalize from seen to unseen states. This capability not only enables them to estimate values for states outside the training set but also makes them suitable for continuous state spaces.

Using ANNs for RL also presents challenges, such as instability arising from the non-stationary nature of data and correlations between samples. Additionally, there's no guarantee that they will converge to the optimal value functions. [31]

Neural Fitted Q Iteration The *Neural Fitted Q Iteration* (NFQ) algorithm, introduced by Martin Riedmiller in 2005 [44], leverages an ANN to approximate the Q-function. As a model-free and off-policy method, NFQ operates on a fixed set of experiences, employing offline learning to iteratively refine its Q-value estimates. An experience in this context includes a state s, an action a, a reward R, and a subsequent state s'. The algorithm's iterative process involves predicting Q-values through a forward pass and then adjusting the ANN weights in a backward pass to minimize the difference between the predicted and target Q-values. The target Q-value is computed as shown in Equation 2.13:

$$target = R(s, a, s') + \gamma \max_{b} Q(s, b) \tag{2.13}$$

NFQ's methodology, despite its simplicity, has proven effective in managing challenges posed by high-dimensional state spaces and continuous action domains [29].

2.3 Explainable AI

Understanding why an AI model makes certain decisions is not a trivial task. The discipline of Explainable AI (XAI) focuses on the task of making the decisions of AI models comprehensible and explaining why an AI model makes certain decisions in a way that humans can understand. In the following sections, the concept of XAI will be outlined, some specific ways of achieving explainability in AI models will be pointed out, and the special case of experience-based XAI will be introduced.

2.3.1 Introduction to Explainable AI

As AI methods are achieving higher and higher levels of performance in solving increasingly complex tasks, their usage in critical contexts, such as medical diagnosis, security, or finance rises [48]. Whilst earlier implementations of AI models, such as decision trees, were easy to understand, more modern approaches like the deep learning methods introduced in section 2.1.3, consisting of vast amounts of layers with millions of parameters, are considered as black-box models [9]. Various reasons underscore the importance of making black-box model decisions explainable, some of which are:

- **Trustworthiness:** To gain more acceptance for the usage of modern AI technologies in critical contexts, trust has to be established amongst stakeholders. Trustworthiness in the context of XAI can be described as the confidence in the ability of an AI model to work as intended [5].
- **Fairness:** With an increasing demand for ethical AI, XAI can offer insights into topics like the bias of an AI model towards certain social groups. With these insights, fairness towards all individuals can be improved [33].
- **Compliance:** New state regulations are put into place to ensure trustworthy and explainable AI. In a proposal by the European Parliament, it is stated that *"High-risk AI systems shall be designed and developed in such a way to ensure that their operation is sufficiently transparent to enable users to interpret the system's output and use it appropriately."* [19].

Additional reasons outlined by Arietta et al. [5] are causality, transferability, informativeness, confidence, accessibility, interactivity, and privacy awareness.

While the importance of XAI has been underlined by the reasons above, the difficulties of XAI also have to be taken into account. One factor is the *Performance / Interpretability trade-off*. Simple models like decision trees are easier to understand because of their simple underlying mathematical structure but they also tend to perform worse on complex tasks than black-box models [5]. While deep learning methods have shown good performance, the ability to interpret results is low. Rule-based systems on the other hand have comparatively bad performance but their decision process is completely comprehensible. Figure 2.6 visualizes this trade-off.

As opposed to glass-box models like rule-based models that have an intrinsic explainability, in order to explain the predictions made by black-box models, post hoc explainability methods have to be applied. Some of which will be outlined in the following section 2.3.2.

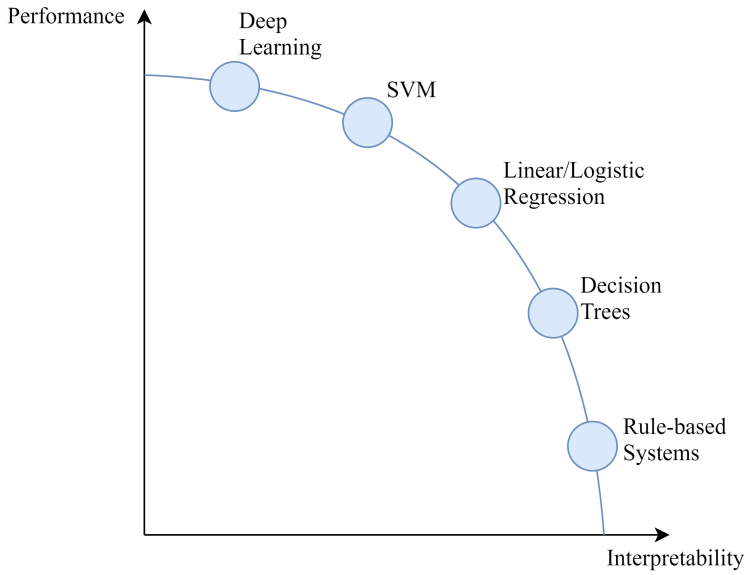


Figure 2.6: Visualization of the Performance / Interpretability trade-off (adapted from [5]).

2.3.2 Methods to Achieve Explainability

Over the years, several techniques have been developed to achieve explainability in AI models. Adadi and Berrada [2] classify XAI methods based on two criteria. Firstly, depending on the timing of information extraction, they can be either intrinsic or post-hoc. Secondly, in terms of scope, they can be categorized as either global or local. Figure 2.7 illustrates these relationships.

Global and Local Explainability When referring to the scope of an explanation, a distinction between global and local explanations has to be made. A global explanations goal is to explain a model's behavior as a whole, providing insights into its general decision-making process, by inspecting the structure of the model. Local explanations aim at explaining individual decisions made by the model rather than a model's overall behavior, like "Why did the model make a certain prediction/decision for an instance/for a group of instances?" [43]. Another goal of local explanations is to identify which of the input features are most relevant for the outputs made [18].

Intrinsic and Post-hoc Explainability The timing of explanation application differentiates intrinsic from post-hoc explainability methods. As illustrated with the example of decision tree models in section 2.3.1,

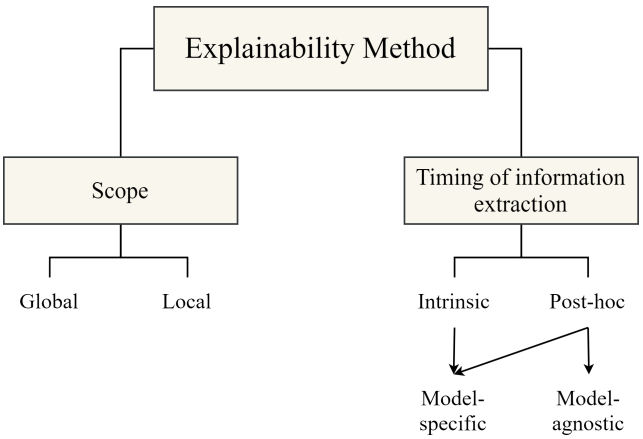


Figure 2.7: Overview of distinction factors for XAI methods (adapted from [2]).

intrinsic explanations are embedded within the model's design, making the model itself interpretable by nature. On the flip side, post-hoc methods come into play after a model has been trained, aiming to elucidate the already established model's decisions. These techniques dissect the model's predictions without going into its internal mechanics, offering insights into its decision-making process after training [42].

Model-Specific and Model-Agnostic Explainability Techniques limited to a specific model or model class are called model-specific. For instance, the explanation of a decision tree model is model-specific, because the explanation of intrinsically explainable models is always model-specific [42]. Similarly, methods that exclusively work for the explanation of an ANN are model-specific. In contrast, model-agnostic methods can be applied to any model and are applied post hoc. They typically work by assessing the correlation between feature inputs and their resulting outputs [42]. These methods are not able to access internal features of the model such as the weights.

In the following, a selection of some widely used explainability methods are presented.

LIME - Local Interpretable Model-Agnostic Explanations LIME is a local model-agnostic algorithm. It works by approximating a complex black-box model with a simpler interpretable model, like a decision tree, for individual predictions. LIME achieves this by perturbing the input data, observing

the changes in predictions, and then fitting a simple model to explain those changes. The goal is to identify which features in the input data are most influential for a particular prediction. This provides a local explanation, helping to understand why a specific decision was made by the model.

SHAP - SHapley Additive exPlanations SHAP is a method to explain the output of any machine learning model by attributing the prediction to its input features. It is based on Shapley values [45], a concept from cooperative game theory, which ensures a fair distribution of contributions among features. For each prediction, SHAP values indicate how much each feature contributed, either positively or negatively, relative to a baseline value. The sum of all SHAP values for a given prediction equals the difference between the model's output and the baseline. This allows for consistent and locally accurate feature importance explanations, helping users understand the model's decision-making process. A formalization of the calculation of the SHAP value of the i^{th} feature of a machine learning model f and an instance x with M features is provided in Equation 2.14:

$$\phi_i(f) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(M - |S| - 1)!}{M!} \left[f(S \cup \{i\}) - f(S) \right] \tag{2.14}$$

Where N is the set of all features, S is a subset of N that does not include feature i, $f(S \cup \{i\})$ is the prediction of the model when only the features in S are used and $f(S)$ is the prediction of the model when only features in S are used. In other terms, the SHAP value for a feature is the weighted average of the difference in model predictions with and without that feature, over all possible subsets of features [42].

Kernel SHAP Kernel SHAP is an extension of the original SHAP method, designed to provide an efficient approximation of Shapley values for any machine learning model. While the original SHAP method requires evaluating all possible combinations of features, which is computationally expensive, Kernel SHAP uses a kernel function to weigh the importance of each subset of features, thereby reducing the computational effort.

Kernel SHAP solves a weighted linear regression problem to approximate the Shapley values. The weights are determined by the kernel function, which considers the size of each coalition of features and the total number of features. The formula for Kernel SHAP can be represented as follows:

$$\phi_i(f) \approx \sum_{S \subseteq N} w(S) \left[f(S \cup \{i\}) - f(S) \right] \tag{2.15}$$

Where $w(S)$ is the weight assigned by the kernel function to the subset S, N is the set of all features, and $f(S \cup \{i\})$ and $f(S)$ are the model's predictions when the features in S are used, with and without feature i, respectively.

This approach allows Kernel SHAP to offer a computationally efficient yet accurate approximation of the Shapley values, making it suitable for explaining the decisions of complex machine learning models [42].

The examples of XAI methods provided here focus on explanations of the level of features of the input samples. When one wants to know which samples of the training data were decisive for a certain prediction or decision of the model, different methods have to be applied, some of which will be elucidated in the following section 2.3.3.

2.3.3 Experience-Based Explainable AI

Experience-based XAI, also referred to as *example-based explanations* aims at selecting certain data samples that have been used to train a model, so-called *experiences*, in order to explain the behavior of a machine learning model [42].

This type of explanation resonates with human understanding because it mirrors the decision-making processes inherent in the human brain and thus can enhance the acceptance of AI models. Here is an example to illustrate this: A physician encounters a patient with unusual symptoms. The patient's symptoms remind her of another patient with similar symptoms she had some time ago. She suspects that her current patient could have the same disease and proceeds to take a blood sample to test for this specific disease [1]. In this example, the physician makes a decision based on her past experience. When approached by another person and asked for the reason for this decision, the acceptance for this decision would be much higher when justifying it with the past experience the physician made. Examples like this can be found all over in everyday life, like not touching the hot stove plate because we had a bad experience in our childhood or taking a different route to work because the recommended route caused us to be late because of traffic jams in the past.

In the following, some experience-based methods are presented.

Counterfactual Explanations Counterfactual explanations provide insights into a model's decision by suggesting the smallest change to the input that would have changed the model's decision. Wachter et al. [51] proposed a method for generating counterfactual explanations through the formulation of an optimization problem, with the objective of identifying the nearest instance that alters the model's decision.

Influential instances Influential instances or influential experiences are the training data points that are the most influential for the parameters of a

prediction model or the predictions themselves, meaning that the more influential an experience is, the more the removal of that particular experience affects the outcome of the prediction of the model. One approach for finding influential experiences is *deletion diagnostics*. Deletion diagnostics works by repeatedly retraining a machine learning model, each time omitting individual experiences [42]. The influence of an experience i for the prediction of instance j can then be defined by a simplified form of the *Cook's distance* [10] as depicted in Equation 2.16:

$$Influence_j^{(-i)} = |\hat{y}_j - \hat{y}_j^{(-i)}| \tag{2.16}$$

Where $\hat{y}$ is the output of the model.

Prototype and Critic Explanations Prototype explanations provide representative examples from a specific class, while critic explanations highlight borderline cases that help differentiate between classes. The MMD-critic approach is one method to identify both Prototypes and Critics. This method evaluates the data distributions of the entire dataset against selected prototypes. Prototypes are chosen based on their ability to minimize the distance between these distributions. Subsequently, critics are identified through a greedy search, pinpointing data points that maximize the distance from other points, emphasizing their distinctiveness. [34]

Chapter 3

Related Work

In this chapter, the state-of-the-art research related to the topics covered in this thesis is presented. First, in section 3.1 recent advances in general strategies to make RL more interpretable will be presented. After that, in section 3.2 the focus will be shifted from general strategies to approaches that emphasize the use of training experiences to explain RL policies. Finally, in Section 3.3, the unique contributions of this thesis will be outlined, highlighting how it diverges from and extends the existing literature.

3.1 Explainable AI for Reinforcement Learning

As use cases for RL increase, the interest in explaining RL models and the decisions of RL agents increases as well. In recent years, many attempts have been made to increase explainability in the RL domain [40].

Hein et al. [28] propose a way to train interpretable RL policies by using genetic programming applied post-hoc. The approach works by using genetic programming in order to learn a policy from state-action trajectory samples generated by an already trained policy. The resulting policy equation generated by genetic programming can be more easily understood by humans than for example a DNN-based policy. Like in this thesis, besides other experiments, Hein et al. also conduct experiments on the industrial benchmark environment to show that their approach works in complex industrial control settings. Another way of creating intrinsically interpretable policies is proposed by Landajuela et al. [36], who present an intrinsically interpretable method using a neural-guided policy generator to create mathematical expressions as control policies, which are easily understandable and are iteratively refined through simulation.

Other methodologies prioritize the generation of human-interpretable explanations of policies [40]. These approaches mostly focus on visual, game-like environments [6, 25, 32, 47]. In a publication by Atrey et al. [6] saliency maps are used to evaluate the behavior of RL agents by highlighting the importance of different regions of visual inputs, making this approach focus on feature importance. However, the authors argue that these maps are best viewed as exploratory tools rather than explanatory ones, as the explanations derived from saliency maps can often be subjective and unfalsifiable. Another approach that focuses on creating explanations that are understandable by humans is by Amir and Amir [4]. The authors introduce *HIGHLIGHTS*, an algorithm that is designed to automatically generate summaries of an agent's behavior by extracting important trajectories from simulations of the agent's actions. In this case, trajectories are chosen based on their importance, which is calculated using Q-values. The importance of a state is defined as the difference between the maximum and minimum Q-values for that state.

How will an agent react when encountering similar states? This question can be answered by an approach by Topin and Veloso [49] in which the authors introduce Abstract Policy Graphs (APGs). APGs are Markov chains of abstract states that summarize a policy, allowing individual decisions to be explained in the context of expected future transitions. In this framework, states in which the agent behaves similarly lead the agent to states where it will also behave similarly, effectively predicting the agent's transitions between abstract states.

Madumal et al. [39] present a method that uses causal models to explain the actions of RL agents. They answer "why" and "why not" questions by analyzing these models. Leveraging an action influence graph, which is a structural causal model that encodes causal action-based relationships between state features of interest, they are able to provide detailed explanations for the agent's actions. These explanations help users differentiate between causal relationships and mere correlations, enhancing their understanding of the agent's capabilities.

The approaches discussed in this section primarily concentrate on generating interpretable policies or emphasizing feature importance. However, they do not address the significance of training experiences in the learning process. Upcoming Section 3.2 will delve into recent advancements in this area.

3.2 Experience-Based Explainable AI for Reinforcement Learning

Experience-based methods for XAI in RL focus on identifying training experiences that are most influential for particular decisions made by the agent. As discussed in section 2.3.3, the influence of a training experience can in this case be seen as the change in a metric, like the Q-value, of a certain state-action pair, when removing this particular training experience from the training process [24].

Gottesman et al. [24] explore this idea by introducing the concept of *total influence* and *individual influence* for transitions. The total influence of a transition is defined as the change in the value estimate if that transition is removed from the dataset. Mathematically represented as $I_j = \hat{v}_{-j} - \hat{v}$, where $\hat{v}_{-j}$ is the value estimate using the same dataset after the removal of the j^{th} transition. On the other hand, the individual influence is the change in the estimated value of $q(s^{(i)}, a^{(i)})$ as a result of removing a transition j. It is denoted as $I_{i,j} = \hat{q}_{-j}(s^{(i)}, a^{(i)}) - \hat{q}(s^{(i)}, a^{(i)})$.

The paper employs Fitted Q-Evaluation to model the Q-function of the evaluation policy π_e. This approach is similar to Fitted Q-Iteration but is performed on offline data. The individual influence $I_{i,j}$ represents the change in the estimated Q-value $q(s^{(i)}, a^{(i)})$ due to the removal of a specific transition j. Formally, $I_{i,j} = \Delta q_i - r^{(j)} + \gamma q_j$.

These influence metrics serve as tools for assessing the reliability of Off-Policy Evaluation (OPE) in RL. If no transitions have an influence above a certain threshold, the OPE is considered to be reliable. If any of these transitions are flagged as unrepresentative or artifacts, the OPE is considered to be unreliable. This approach allows for a more interpretable and robust evaluation process. Gottesman et al. validated their approach on medical simulations, as well as on real-world intensive care unit data.

Another approach that leverages training experiences for insights is by Dao et al. [11]. They introduce DRL-Monitor, a framework that captures snapshots of significant moments during Deep Reinforcement Learning (DRL) training. These snapshots are selected using Sparse Bayesian Reinforcement Learning and consist of raw images, actions, and weight distributions.

The DRL-Monitor has two main modules: a DRL module that produces Q-values and a Monitor that extracts feature vectors, called *perceptions*, from the final layer before Q estimation. Snapshots can be visualized to understand the agent's learning process and are particularly useful for interpreting complex input states and actions.

This approach offers a sparse and interpretable set of snapshots, making it easier to understand what and how the DRL agent has learned and is applied to

a visual maze problem and Atari games to observe recorded snapshot images.

3.3 Novel Contributions and Distinctions

While the works discussed in Section 3.1 primarily focus on generating interpretable policies or emphasizing feature importance, they do not delve into the role of individual training experiences. This thesis addresses this gap by applying a *deletion diagnostics* methodology to identify influential training experiences in both a general classification model and RL models.

In comparison to the work presented in Section 3.2, this thesis offers several novel contributions. Gottesman et al. introduce the concept of *total influence* and *individual influence* for transitions, focusing primarily on the reliability of OPE in RL. While the concepts of the paper can be seen as related to the deletion diagnostics approach in this thesis, the deletion diagnostics approach uses the...

Their approach is limited to the influence of transitions on value estimates and is validated mainly in medical simulations and intensive care unit data. Additionally, while Dao et al. capture snapshots of significant moments during training, this thesis aims to quantitatively measure the influence of these experiences on decision-making, thereby offering a more granular level of explainability.

Furthermore, the methodology in this thesis is designed to adapt to complex, real-world industrial scenarios, like the Industrial Benchmark. This allows for a more comprehensive understanding of how specific training experiences can influence decision-making in machine learning models, thereby advancing the state of the art in Experience-based Explainable AI.

Chapter 4

Methodology

This chapter explains how the research in this thesis was carried out. First, the research questions, which guide the rest of the study, are listed in Section 4.1. Next, in Section 4.2, an overview of the general approach to answering the research questions is given. Finally, in Section 4.3 each conducted experiment is explained in more detail.

4.1 Research Question

The central research question that guides this thesis is:

> *How can deletion diagnostics methodology be effectively applied to various machine learning and reinforcement learning settings to identify influential subsets of training data on model outcomes?*

To address this overarching question, several sub-questions are formulated:

1. *How does the deletion diagnostics approach influence the performance of a DNN in a cluster-based classification problem?*
2. *In an RL grid world environment, which subsets of training data have the most significant impact on the Q-values guiding the agent's actions?*
3. *How can deletion diagnostics be adapted to a complex RL environment like the Industrial Benchmark, and what are the implications of different influential training subsets on the agent's policy in such a setting?*

These research questions aim to provide a comprehensive understanding of the utility and limitations of the deletion diagnostics methodology across different ML paradigms, from simple classification tasks to complex, real-world-inspired RL environments.

4.2 General Approach

This thesis is organized into three distinct phases, each increasing in complexity to provide a comprehensive understanding of the research problem. The overarching aim is to apply a deletion diagnostics methodology, detailed in Section 2.3.3, to identify experiences that exert a significant influence on specific outcomes.

In the first phase, the research employs a cluster-based classification technique. Points in a two-dimensional space are clustered into several classes by their location, and a DNN is trained to classify new points. The primary objective here is to identify subsets of training data that most significantly influence the prediction scores for the correct class of specific test points, as determined by the model trained on the full data. This phase serves as an initial validation for the feasibility of the deletion diagnostics approach on a simplified problem.

The second phase introduces an RL grid world environment, adding layers of complexity by incorporating not only a goal state but also a stochastic teleportation field with a configurable probability of transporting the agent to the goal state. The focus in this phase is on determining which subsets of training data have the most substantial impact on the Q-values of the action guiding the agent to the teleportation field instead of directly to the goal state.

The third and final phase engages with the Industrial Benchmark (IB) [27], a sophisticated RL environment designed to emulate real-world industrial control scenarios. It features high-dimensional continuous state and action spaces, stochastic dynamics, and a multi-objective reward signal. Here, the analysis aims to identify influential subsets of training data that affect the Q-values associated with an agent's actions in this complex environment.

The general deletion diagnostics methodology employed in this thesis is illustrated in Figure 4.1. The process unfolds as follows:

1. **Training Data Generation / Collection:** Initially, training data is either generated or collected by an agent in the case of an RL setting.

2. **Data Partitioning:** The collected training data is then segmented into smaller subsets based on a specific criterion. Multiple training datasets are formulated, including one that contains the complete data and others where each lacks a specific partition of the original data.

3. **Training:** ML models are trained using each of these distinct datasets. In settings with DNNs where the resulting outputs of the DNN might fluctuate, even when trained on the same data, multiple trainings are carried out, resulting in multiple models or policies per training dataset. This is to eliminate the influence of outliers in the evaluation process.

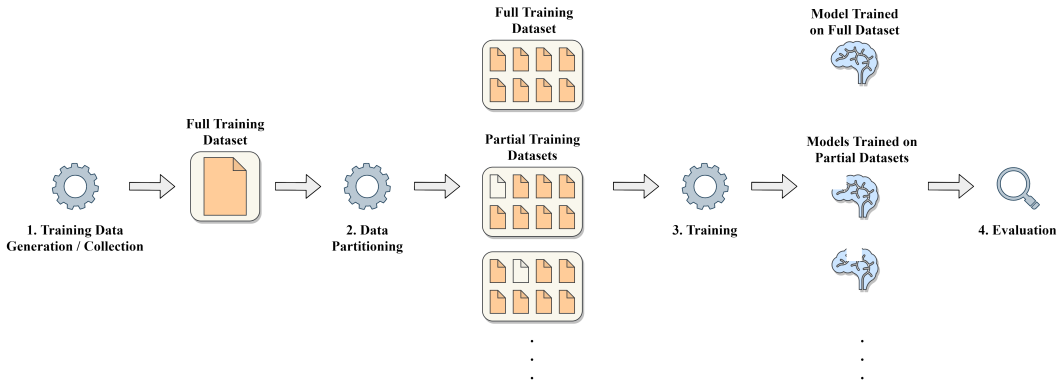


Figure 4.1: The general deletion diagnostics workflow applied in this thesis.

4. **Evaluation:** Finally, each trained model is evaluated for general correctness and later to assess the impact of the deleted data partitions.

In the subsequent Section 4.3, a comprehensive elaboration of the three mentioned phases will be provided. This will include detailed descriptions of the RL environments, as well as specific training methodologies utilized.

4.3 Experiment Specifics

As introduced in Section 4.2, there are three phases to this thesis. This section offers a detailed exploration of each phase, explaining both the methodologies used and the reasoning behind their usage. Each sub-section will describe one phase.

4.3.1 Cluster-Based Classification

The cluster-based classification experiment is supposed to be the first step in researching the feasibility of a deletion diagnostics approach in an ML setting and should give an idea of what can be expected from this type of approach and what is needed for an elaborate evaluation of the results.

In the experimental setup, data points are generated in a two-dimensional space. These points are then grouped into clusters using a clustering algorithm, each cluster being assigned a unique class label. A DNN is subsequently trained on these clustered data points to enable the classification of new, previously unseen points.

To implement the deletion diagnostics technique, the spatial region containing the training points is segmented into a grid, consisting of multiple square

(a) Generated training data points.

(b) Clustered training data points.

(c) Clustered training data points with a grid.

Figure 4.2: Process of generating training data points (4.2(a)), clustering them (4.2(b)), and applying a grid segmentation to the space (4.2(c)).

cells. For each grid cell, a specialized DNN is trained, intentionally excluding the data points within that cell. This results in two sets of DNN models: one trained on the complete dataset and another set where each model is trained without the data points from a specific grid cell.

The process of generating the training data points, clustering them, and applying a grid to the two-dimensional space is depicted in Figure 4.2.

For evaluation, test points not included in the original training set are selected. These points are classified by each of the trained DNN models. The classification score for the correct class, as determined by the DNN trained on the full dataset, serves as the ground truth. The scores from the other specialized models are then compared to this baseline. The influence of the training samples within a particular grid cell is calculated by the difference between the ground truth score and the score from the model that excluded those specific samples, in accordance with Equation 2.16.

An underlying hypothesis to be empirically evaluated by experiments is that training data points in close proximity to a specific test point, or grid cells containing training data from multiple classes that aid the DNN in class differentiation, have a greater influence on the classification of that point compared to other data points.

4.3.2 Grid World

The second phase of this thesis focuses on exploring a grid world environment, which represents an escalation in complexity relative to the cluster-based classification scenario. Additionally, this environment serves as an intermediary

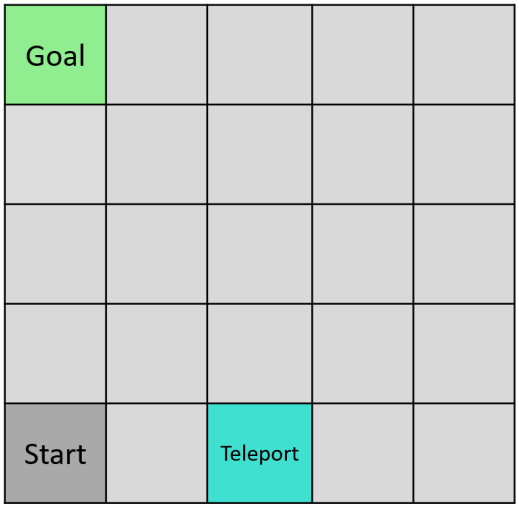


Figure 4.3: An example for the grid world environment.

stage, bridging the gap between the simpler classification example and the more intricate industrial benchmark setting.

4.3.2.1 Environment

The grid world environment implemented for this thesis is a two-dimensional grid with various types of cells, each with its own characteristics:

- **Start Field:** The initial position of the agent.
- **Standard Field:** A standard cell where the agent can move freely.
- **Goal State:** The target cell that the agent aims to reach.
- **Teleportation Field:** A special cell that has a stochastic behavior.

An example for this grid world is displayed in Figure 4.3. The example environment is of size 5x5 and features a start field located at the bottom-left corner, a goal state at the top-left corner, and a teleportation field situated at the bottom-center of the grid. All other fields are considered standard fields.

Actions and Transitions The agent has the ability to move in four cardinal directions: up, right, down, and left. Each action leads to deterministic transitions, except when the agent lands in a teleportation field. In this special cell, the agent has a pre-defined probability of being teleported directly to the

goal state. In the case of teleportation, the subsequent state in a transition is not the teleportation field, but the goal state.

State Space and Observations The state space is a discrete set of cells within the grid. The agent receives observations that encode its current position, allowing it to make informed decisions based on its location within the grid.

Reward Structure The agent receives rewards based on the type of cell it moves into. The rewards serve as signals to guide the agent toward the goal state while avoiding less favorable cells and punishing the distance the agent travels to reach the goal state. The reward structure is as follows:

- **Start and Normal Field:** Negative reward to encourage minimal travel distance.
- **Goal State:** Positive reward to signal successful completion.
- **Teleportation Field:** Negative reward to indicate risk. Essentially treated as a normal field if teleportation does not occur.

Stochastic Elements The environment introduces stochasticity through the use of a teleportation field. Upon entering this field, the agent encounters uncertainty due to the probabilistic chance of being teleported to the goal state. This integration of stochastic elements serves several research aims. Firstly, it injects a degree of unpredictability that better mirrors real-world scenarios, enhancing the study's validity. Secondly, the uncertain nature of the teleportation field offers a rigorous testing ground for the models, compelling the agent to adapt its strategy to accommodate these uncertainties. This added layer of complexity acts as a transitional bridge between the more straightforward cluster-based classification and the highly complex Industrial Benchmark settings, thereby facilitating a more nuanced assessment of the deletion diagnostics methodology.

4.3.2.2 Training

For the training process, multiple methodologies have been employed to generate the training data, and various algorithms have been utilized to optimize the agent's policy effectively.

Training Data Generation The training data for the grid world environment is generated through two distinct modes: *Episodic Mode* and *Variable Teleport Chance Mode*.

In the Episodic Mode, the agent navigates the environment using a stochastic policy, where the agent chooses actions in a completely random manner for a predetermined number of episodes. The teleportation chance remains constant across all episodes, ensuring a consistent environment. Each episode is capped at a maximum number of steps to avoid infinite exploration loops. The agent's experiences, captured as state-action-reward-next state tuples, are aggregated into a dataset for offline training.

Unlike the Episodic Mode, the Variable Teleport Chance Mode introduces variability in the teleportation chance across episodes. For each specified teleportation chance, the agent explores the environment for a set number of episodes, again, choosing actions in a completely random manner. This results in a diverse dataset that captures the agent's experiences under different environmental dynamics.

The compiled datasets from both modes serve as the foundation for offline training of RL agents, enabling a comprehensive evaluation of the policy under varying conditions.

Dynamic Programming Training The agent's policy in the grid world environment is trained using a DP approach, specifically a form of value iteration. This approach is adapted from standard methodologies in the field of RL [46]. Among the primary advantages of utilizing this DP technique are its computational efficiency and the assurance of convergence to an optimal policy under specific conditions. These attributes make it well-suited for environments characterized by a limited state and action space, such as the grid world. Furthermore, the guaranteed convergence ensures consistent outcomes when the agent is trained on identical data sets, thereby contributing to the stability and reliability of the evaluation results.

Neural Fitted Q-Iteration Training Additionally, an NFQ training approach is employed as a more advanced method for training the agent's policy in the grid world environment. This method is particularly advantageous for several reasons. First, it employs a more sophisticated DNN-based training procedure, which serves as verification of the deletion diagnostics methodology for DNN-based RL agents. Second, it serves as an effective transitional step toward the more complex Industrial Benchmark environment, especially given its demonstrated efficacy in that context [29]. One of the standout benefits of the NFQ approach is its proficiency in efficiently managing complex

state-action spaces, thereby making it a good preparatory step for subsequent transition to the Industrial Benchmark setting.

Application of Deletion Diagnostics The application of deletion diagnostics in the grid world environment is versatile and can serve various analytical purposes, such as evaluating the Q-value of actions directing the agent toward the teleportation field as opposed to directly to the goal state. As previously outlined, two distinct methodologies for generating training data exist: The episodic mode and the variable teleport chance mode.

In both modes, the initial step is the same as in the cluster-based classification approach: a policy is trained using the full dataset.

In the episodic mode, deletion diagnostics is implemented by training separate policies that each exclude a specific episode from the training dataset. For example, if the training data comprises 50 episodes of random exploration, 50 distinct policies can be trained, each omitting one particular episode. Subsequent evaluation may reveal that certain episodes exert a greater influence on specific Q-values.

In the variable teleport chance mode, the application of deletion diagnostics follows a similar logic. However, in this case, episodes are excluded based on their teleportation probabilities. For instance, if the training data includes episodes with teleportation chances of 0%, 10%, ... 100%, separate policies are trained that exclude episodes with each of these probabilities. This allows for a nuanced analysis of how varying teleportation probabilities influence the trained policies.

Application of SHAP In addition to the basic deletion diagnostics procedure, a more advanced technique employing Shapley values has been tested in this setting. Specifically, a modified version of the Kernel SHAP approach, as described in Section 2.3.2.

Unlike the standard SHAP method, which focuses on the contributions of individual input features to the model's prediction, this modified approach focuses on the contributions of partial training subsets. In the context of the episodic data generation mode, each episode is treated as a partition to be left out during the Kernel SHAP analysis. In the variable teleport chance mode, one teleportation probability, including multiple episodes would be treated as one partition.

The modified Kernel SHAP formula can be represented as:

$$\phi_i(f) \approx \sum_{S \subseteq N} w(S) \left[f(S \cup \{i\}) - f(S) \right] \tag{4.1}$$

Where $w(S)$ is the weight assigned by the kernel function to the subset S, N is the set of all data partitions, so the full dataset. $f(S \cup \{i\})$ and $f(S)$ are the model's predictions when the partitions in S are used, with and without data partition i, respectively.

This approach allows for a better understanding of how each data partition, influences certain Q-values, providing a way to approximate the contributions of each data partition.

To enhance the interpretability of these findings, the episodes or teleport probabilities including episodes are subsequently classified into five distinct categories based on their influence on Q-values: *strong_positive*, *positive*, *neutral*, *negative*, and *strong_negative*. Episodes in the *strong_positive* category are those that significantly contribute to elevating the Q-value for the action being analyzed, while those in the *strong_negative* category have the contrary effect.

Extracting Single Influential Experiences To further elucidate the differences between data partitions that exert high versus low influence on Q-values, an approach based on the Kullback-Leibler (KL) divergence [35] is utilized. The KL divergence quantifies the discrepancy between two probability distributions $P(x)$ and $Q(x)$, and is formally defined as:

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{4.2}$$

In this specific context, the formula is adapted to compare individual experiences within different data partitions. Instead of using full probability distributions, the focus is on individual probabilities. Specifically, the ratio p_x of a particular experience to all other experiences in a partial dataset is compared to the ratio q_x of this experience in the full dataset. This leads to a simplified equation:

$$divergence = p_x \log \left(\frac{p_x}{q_x} \right) \tag{4.3}$$

A positive divergence score implies that the frequency of the particular experience is higher in the data partition than in the full dataset. Conversely, a negative score indicates that the experience is less frequent in the data partition relative to the full dataset.

Distinction between the Dynamic Programming and the Neural Fitted Q-Iteration Approach While the Dynamic Programming approach undergoes a comprehensive evaluation using all the mentioned methods, the

NFQ approach serves the role of a transitional step toward the Industrial Benchmark setting. As such, only the basic deletion diagnostics method is applied to the NFQ approach, to assess the capabilities of deletion diagnostics in a more complex DNN-based environment, as opposed to the simpler table-based methods used in Dynamic Programming. Given the computational demands of the SHAP approach, which requires training an extensive number of models, its application to NFQ training is considered beyond the scope of this thesis.

4.3.3 Industrial Benchmark

The third phase of this thesis delves into the complexities of the Industrial Benchmark (IB) [27] environment, an RL environment that aims to realistically simulate the complexities commonly encountered in real-world industrial settings.

4.3.3.1 Environment

The IB environment features a high-dimensional, partially observable state space and a continuous action space. The actions are composed of three continuous elements that influence three different control variables. The environment also incorporates stochastic elements and delayed outcomes. The objective is multi-faceted, with two reward components that are inversely related to the actions. The system's dynamics are influenced by latent variables and are heteroscedastic, meaning the variability of outcomes changes with the state. Additionally, the system is affected by an external factor that is beyond the control of the actions. The optimal policy in this environment will not converge to a static set of control variables. All these features are inspired by challenges commonly faced in industrial applications.

Actions and Transitions The agent can interact with the environment through actions a_t that are three-dimensional vectors in $[-1, 1]^3$. These actions can be viewed as suggested modifications to three observable steering variables: velocity v, gain g, and shift h. Each of these steering variables is constrained to the interval $[0, 100]$, leading to the following equations:

$$a_t = (\Delta v_t, \Delta g_t, \Delta h_t) \tag{4.4}$$

$$v_{t+1} = \max(0, \min(100, v_t + d_v \Delta v_t)) \tag{4.5}$$

$$g_{t+1} = \max(0, \min(100, g_t + d_g \Delta g_t)) \tag{4.6}$$

$$h_{t+1} = \max(0, \min(100, h_t + d_h \Delta s_t)) \tag{4.7}$$

Here, the scaling factors d_v, d_g, and d_h are set to 1, 10, and 5.75, respectively.

State Space and Observations Taking an action a_t, the system transitions to a new internal state s_{t+1} at the next time step $t+1$. These internal states serve as the Markovian representations of the environment. An external predefined load p_t, referred to as *setpoint*, is applied to the system, affecting both consumption c_t and fatigue f_t. The setpoint cannot be changed by any actions and has a significant influence on the dynamics and stochasticity of the environment. Two operational modes are possible for the setpoint. It can either be fixed at a constant value, essentially serving as a hyperparameter, or it can operate as a time-varying external driver, leading to highly non-stationary dynamics. In this thesis, exclusively the first mode, where the setpoint is fixed, is adopted.

The environment updates these variables to c_{t+1} and f_{t+1} as part of the new internal state s_{t+1}. The observed state o_t is a subset of s_t and includes:

1. Current control variables: velocity v_t, gain g_t, and shift h_t.
2. External influencing factor: setpoint p_t.
3. Reward-related metrics: consumption c_t and fatigue f_t.

Additionally, the agent is allowed to use previous observation vectors and actions to estimate the Markovian state s_t. This feature adds a layer of complexity and realism to the environment, as it mimics the conditions often found in industrial applications.

Reward Structure The reward r_t is exclusively based on the new internal state s_{t+1} and is calculated as:

$$r_t = -c_{t+1} - 3f_{t+1} \tag{4.8}$$

In the industrial scenarios that inspired this environment, the reward function is usually explicitly defined. Hence, it is assumed that both consumption c_t and fatigue f_t are observable, while the rest of the Markov state remains hidden.

The agent's reward is also influenced by its ability to oscillate the shift h around an *effective shift* h^e, which is itself influenced by the setpoint p. The formula for calculating h^e is:

$$h^e = \max\left(-1.5, \min\left(1.5, \frac{h}{20} - \frac{p}{50} - 1.5\right)\right) \tag{4.9}$$

By setting h^e to zero and solving for h, one can find a value for h, called *safe zone* that yields relatively good results without requiring the agent to learn how to handle h. This simplifies the action-space to two dimensions. The equation for h when $h^e = 0$ is:

$$h = \frac{p}{2.5} + 30 \tag{4.10}$$

This simplification not only reduces the computational effort during training but also facilitates a more straightforward interpretation of evaluation results.

4.3.3.2 Training

This section describes the methodologies and approaches employed for generating training data, applying the NFQ algorithm for policy training, and utilizing deletion diagnostics to evaluate the trained models for the IB environment.

Training Data Generation The generation of training data for the IB environment follows the same principles as in the previous phases. The agent navigates the environment using a stochastic policy, where actions are selected at random for a predetermined number of episodes each having a fixed number of steps. At the start of each episode, the agent begins from default settings [29]. This process is executed for environments with diverse setpoint values (p), with the setpoint remaining constant throughout each episode. This approach serves as a simulation for collecting training data under varying conditions, which then forms the foundation for offline RL training across different environmental settings.

Neural Fitted Q-Iteration Training The NFQ training approach has been previously validated by Hein et al. [29]. Their work provides empirical evidence for the effectiveness and suitability of NFQ in training RL policies for the IB environment. One of the fundamental prerequisites for employing NFQ is the necessity for a discrete action-space. This is a critical consideration because many real-world environments, including the IB environment, naturally have a continuous action-space. To adapt NFQ to such settings, the continuous action-space must be discretized into a finite set of actions. Additionally, NFQ is designed to operate on offline data, which aligns well with the deletion diagnostics approach employed in this thesis.

Application of Deletion Diagnostics In the case of the IB environment, the criterion for excluding episodes is based on the setpoint values of the en-

vironment from which their transition data has been collected. For example, if the full training dataset includes episodes with setpoint values $p \in \{10, 20, \ldots, 100\}$, multiple policies are trained in isolation, each excluding episodes corresponding to a specific setpoint value from the full dataset. For instance, one policy might be trained on data that excludes all episodes with a setpoint of $p = 10$, another with $p = 20$, and so on. With this setting an evaluation of the influence of datasets generated in environments with different p values on the resulting policy can be conducted. As previously mentioned in Section 4.3.2.2, the computational demands of employing the Shapley value methodology on sophisticated DNN-based models make it beyond the scope of this thesis. Consequently, the evaluation process for the IB environment is based on the utilization of Q-values, assessed through the basic deletion diagnostics approach. This limitation also serves as a point of consideration for future research.

Consequently, a policy trained on the full dataset is selected as base policy for comparison. From the trajectory generated by this policy agent, a specific state is isolated for analysis. The Q-values and chosen actions at the chosen state are then compared across the various policies trained on the different datasets to assess their relative output.

In addition to Q-values, this experiment also evaluates advantage values. For each trained policy, the advantage of the action selected by the policy is compared to the action chosen by the base policy. The advantage is calculated using the following formula:

$$\text{Advantage}(s, a) = Q(s, a) - V(s) \tag{4.11}$$

where $Q(s, a)$ is the Q-value of taking action a in state s, and $V(s)$ is the value of the state s.

A noteworthy observation would be the divergence in Q-values between policies trained on different datasets, while Q-values trained on the same dataset are similar. These differences could then be examined further. This remains to be investigated in the experiments.

Chapter 5

Implementation

5.1 Programming Setup

The implementation of the methodologies described in this thesis relies on a well-defined programming setup to ensure both computational efficiency and reproducibility. The primary programming language used is Python 3.10 [22], chosen for its readability, extensive libraries, and strong community support, particularly in the realm of ML and data science.

For numerical computations, the NumPy library [14] is employed. It provides a robust set of functions to perform mathematical operations efficiently, thereby facilitating the manipulation of large datasets and complex calculations.

The ML models are trained using PyTorch [16]. This open-source ML library offers a wide range of functionalities for tensor computations, automatic differentiation, and GPU acceleration, making it a good choice for the creation of ANNs and the training-related tasks in this thesis.

For the purpose of data visualization, Matplotlib [13] is used in conjunction with the Seaborn library [17]. These libraries offer a high degree of customization and are capable of producing a wide variety of visualizations.

To maintain a comprehensive record of experiments and their outcomes, a logging system is implemented by using the Python logging library [20]. This not only helps in debugging but also provides a transparent history of the research process, which is important for reproducibility.

Configuration settings for the experiments are managed using YAML files. This approach ensures that the experiments can be easily replicated, as all the hyperparameters and settings are stored in a structured, human-readable format.

Lastly, the trained machine learning models are serialized and saved using the Pickle library [21]. This allows for the storage of large data structures, in

a way that they can be reloaded and used in future experiments or for further analysis.

5.2 Cluster-Based Classification

As described in detail in Section 4.3.1, cluster-based classification is an experimental setup designed to explore the influence of parts of the training data on the performance of DNNs in a classification task. The experiment leverages a grid-based approach to segment the training data into distinct subsets, or grid cells, each of which is used to train multiple instances of a classification model. This setup allows for an evaluation of how each data partition contributes to the model's prediction scores. As a first step, Table 5.1 outlines the hyperparameters used in the experiment, each of which can be tuned to explore different aspects of the model's behavior.

Parameter	Description
`n_samples`	Total number of data points generated for training.
`n_clusters`	Number of clusters the clustering algorithm should produce.
`test_size`	Ratio of the dataset reserved for validation of the classification model.
`learning_rate`	Learning rate for the classification model training.
`patience`	Number of steps without improvement before halting training.
`n_models`	Number of models to train for each grid cell.
`n_test_points`	Number of points that should be generated for testing the influence of training data partitions on them.
`n_grid_cells_per_dimension`	Number of grid cells each dimension should be separated into.

Table 5.1: Parameters for the Cluster-Based Classification experiment.

5.2.1 Training Data Generation

The data for training is generated using the `n_samples` hyperparameter, which specifies the total number of data points to be created. For better visualiza-

tion and interpretation of the clustering results, the data points are initially generated in a manner that already loosely groups them into clusters. This is accomplished using the `make_blobs` function from the scikit-learn Python library [37]. The function generates two-dimensional sample points around multiple centers, following an isotropic Gaussian distribution. The number of these centers is determined by the `n_clusters` hyperparameter.

To avoid the complexity of overlapping clusters in the training data, a K-means algorithm is employed for further clustering, again using `n_clusters` as an indicator for the number of clusters to create. As highlighted in Section 2.1.2, K-means yields clusters with distinct, sharp boundaries. These newly labeled data points then form the training set for subsequent classification model training.

Additionally, to facilitate the application of the deletion diagnostics technique, the training data is segmented into distinct subsets. This is in accordance with the methodology outlined in Section 4.3.1. The hyperparameter `n_grid_cells_per_dimension` sets the number of grid cells each dimension of the two-dimensional space should be partitioned into. Grid cells that do not contain any data points are omitted since the resulting dataset would just be the full dataset.

5.2.2 Model Training

The classification model uses a simple DNN with two input neurons for the x- and y-coordinates of the data points. The network has two hidden layers of 64 neurons each, followed by a 32-neuron layer. All layers use a ReLU activation function. The output layer matches the number of clusters, denoted by `n_clusters`.

The model is trained using the Adam optimizer, with the learning rate set by the `learning_rate` hyperparameter. Cross-Entropy Loss is used as the objective function to minimize during training. The model undergoes multiple epochs, where each epoch consists of a forward and backward pass to update the model parameters. Early stopping is employed to halt the training process if the validation loss does not improve for a consecutive number of epochs, as specified by the `patience` hyperparameter. The trained model outputs a score for each cluster label, and a data point is classified based on the label with the highest score.

The training process is executed for each of the previously generated datasets and is repeated multiple times as specified by the `n_models` hyperparameter, resulting in `n_models` number of classification models per training dataset. As previously stated in Section 4.2, this serves to mitigate the influence of outliers during evaluation.

5.2.3 Evaluation

A total of `n_test_points` test points are uniformly distributed within the region where the training data was generated. For each test point, the full models predict both the scores and the class labels, and these scores are subsequently averaged. Next, the partial models also predict scores for the class identified by the full models, and these scores are averaged within each grid cell. The influence of a specific grid cell on a test point's prediction is then determined by calculating the difference between the average score for the correct class from the full model and the corresponding average score from the partial model.

5.3 Grid World

Hyperparameters different depending on training algorithm

5.3.1 Environment

The grid world environment is implemented using the OpenAI Gym library [15]. The environment is initialized with a 2D grid encoded as a list that consists of various types of fields: the start field ('S'), standard field ('F'), goal state ('G'), and teleportation field ('T'). Each cell in the grid is uniquely identified by an integer, starting from 0 at the top-left corner and incrementing sequentially towards the bottom-right. Some functions are implemented to specifically fulfill the interface structure established by OpenAI Gym.

The transition dynamics of the environment are established through the `initialize_transitions` function. This function constructs a transition table that captures the probabilities of moving from one state to another based on the agent's actions. For most states and actions, the transition probability is deterministic and set to one. Exceptions are made for the teleportation field, where the transition probability is determined by a predefined hyperparameter. If the agent attempts to move against a wall, it remains in the current field, effectively moving into that field from the field itself.

The `reset` function serves multiple roles in setting up the environment for a new episode. It re-positions the agent at the start field, sets the reward to zero, initializes the `done` flag to `False`, and re-initializes the transition table. The function then returns the agent's initial state, represented as an integer value.

Agent-environment interactions are governed by the `step` function. This function takes an action as input and performs several tasks. It identifies feasible transitions and their associated probabilities through the transition table based on the agent's current state and the chosen action. The agent's

Parameter	Description
`n_episodes`	Total number of episodes to generate training data for.
`max_steps_per_episode`	Maximum number of steps the agent takes per episode for training data generation.
`experiment_mode`	Either `episodic` or `variable_teleport_chances`. Corresponding to the training data generation modes described in Section 4.3.2.2.
`teleport_chances`	If `experiment_mode` is `variable_teleport_chances`, contains a list of teleportation probabilities used for data generation environments. If `experiment_mode` is `episodic`, contains a single teleportation probability.
`state_to_check`	State for which the Q-value should be examined in this experiment.
`action_to_check`	Action for which the Q-value should be examined in this experiment.
`training_algorithm`	Either `dp` or `nfq` depending on the training algorithm that should be used.
`discount_factor`	Discount factor γ for Q-value calculation in training.

Table 5.2: General parameters for the Grid World experiment.

position is then updated according to a probabilistically selected transition. Concurrently, the function updates the reward based on the new state and sets the `done` flag to `True` if the agent reaches the goal state. Finally, the function returns a tuple containing the new state, the received reward, and the `done` flag.

5.3.2 Training Data Generation

Data generation is executed by a random agent. This agent initiates an episode at the start field and proceeds to randomly select actions. An episode concludes when either the agent reaches the goal field or the number of steps taken

Parameter	Description
n_models	Number of models to train for each dataset partition.
learning_rate	Learning rate for policy training.
patience	Number of epochs without improvement before halting the NFQ iteration.
n_iterations	Number of training iterations.
max_epochs_per_iteration	Maximum number of training epochs per iteration.
n_evaluation_episodes	Number of episodes the current policy is evaluated for per NFQ iteration for policy selection.
n_max_steps_per _evaluation_episode	Maximum number of steps per evaluation episode.

Table 5.3: Parameters specific to the NFQ training for the Grid World experiment.

reaches the limit specified by the max_steps_per_episode parameter. Upon the conclusion of each episode, the reset function of the environment is called.

In the episodic data generation mode, the agent operates through a predetermined number of episodes, as set by the n_episodes parameter, within an environment where the teleportation probability is defined by a single value in teleport_chances.

Conversely, in the variable_teleport_chances mode, the agent completes n_episodes for each teleportation probability listed in teleport_chances, each time initializing a new environment with the corresponding teleportation probability.

5.3.3 Policy Training

For the Dynamic Programming training, it is sufficient to train only one model per teleport probability defined in teleport_chances. Since NFQ is a DNN-based approach the resulting models and their outputs can slightly differ as explained in Section 4.2. As such, multiple models have to be trained for each dataset partition, in order to make a meaningful evaluation. The n_models parameter defines how many models will be trained per data partition.

5.3.3.1 Dynamic Programming

The first approach in the training of the agent's policy in the grid world environment employs a Dynamic Programming (DP) algorithm, specifically a Value Iteration algorithm. This approach is inspired by the methodology presented in Sutton and Barto's RL book [46].

- **Initialization:** The Q-table is initialized with zeros, representing the initial estimated value of taking an action a in state s.
- **Transition Probabilities:** For each state-action pair (s, a) present in the training dataset, the transition probabilities to all possible next states s' and rewards r are computed based on the collected dataset. These probabilities are stored in a four-dimensional array, where the dimensions correspond to the current state, action, next state, and reward.
- **Value Iteration:** The algorithm iteratively updates the Q-values until convergence. For each state-action pair (s, a), the Q-value is updated as follows:

$$Q(s,a) = \sum_{s',r} P(s', r|s, a) \left[r + \gamma \max_{a'} Q(s', a') \right] \tag{5.1}$$

  where $P(s', r|s, a)$ is the transition probability, r is the reward, and γ is the discount factor.
- **Convergence Check:** The algorithm checks for convergence by comparing the Q-table before and after the update. If the maximum absolute difference between the corresponding entries is below a predefined threshold, the algorithm terminates.
- **Final Policy:** Once the Q-table has converged, the final policy is extracted. For each state s, the action a with the highest Q-value is selected. If there are multiple actions with the same highest Q-value, all are included in the policy for that state.
- **Unvisited State-Action Pairs:** Any state-action pairs that were not visited during data collection are assigned a Q-value of -1 to indicate their unexplored status.

The resulting policy and Q-table serve as the basis for the agent's behavior in the Grid World environment.

5.3.3.2 Neural Fitted Q-Iteration

The NFQ training approach is partially employed as a preparatory measure for the IB environment, where the ability to produce well-performing policies has already been tested by Hein et al. [29]. The implementation for the grid world environment is based on Hein et al.'s work, with minor adjustments to tailor it to the specific requirements of the grid world setting.

The foundation for the training is established by an ANN, referred to as QNet. The network consists of an input layer with three neurons, designed to accept the state s encoded as x- and y-coordinates on the grid along with an action a. This is followed by a hidden layer comprising 20 neurons activated by a ReLU function. The output layer comprises a single neuron, which represents the Q-value $Q(s, a)$ for the given state-action pair. The weights of the QNet are initialized to random values in the interval $[-0.1, 0.1]$.

The training process consists of the following steps:

- **Data Partitioning:** The dataset, sourced from collected experiences, is divided into training and testing subsets. Specifically, 80% of the data is allocated for training, while the remaining 20% is reserved for validation purposes.
- **Initialization for Training:** The neural network is trained using stochastic gradient descent, with the learning rate specified by `learning_rate`. The objective function for optimization is the mean squared error between the predicted and target Q-values.
- **Training Iterations:** The training process runs for a maximum of `n_iterations` iterations. At the start of each iteration, target values are calculated. Each iteration consists of up to `max_epochs_per_iteration` epochs. An iteration is considered complete either when the maximum number of epochs is reached or when the validation loss fails to improve for a number of epochs specified by `patience`.
- **Policy Performance Evaluation:** At the end of each training iteration, the current policy is evaluated. This evaluation is conducted over `n_evaluation_episodes` episodes and employs varying teleportation probabilities, as defined in `teleportation_chances`. The average reward per episode is then computed as the evaluation metric.
- **Optimal Policy Retention:** Given the observed tendency of the NFQ algorithm to experience performance degradation after reaching a peak performance [29], it is important to continually save the best-performing

policy. If a policy evaluation score surpasses the previous best, the corresponding neural network weights are saved. When the training is finished, the policy associated with the highest evaluation score is selected as the final policy.

5.3.4 Evaluation

The parameters `state_to_check` and `action_to_check` specify the state-action pair under investigation in the experiment. The objective is to analyze the variations in Q-values corresponding to this particular pair.

For the calculation of Shapley values, the SHAP Python library [7] is used. Specifically, the `KernelExplainer` class, which is an implementation of the kernel SHAP method. The `KernelExplainer` class is initialized by passing two arguments:

1. A reference to a training function, which is used to train the models.
2. A NumPy object that mimics the shape of the dataset intended for the analysis. This serves as a background dataset for the explainer.

Once the `KernelExplainer` object is initialized, the Shapley values are computed by invoking the `shap_values()` method. This method takes a list of data partitions on which the Shapley values are to be evaluated. The method returns the Shapley values, which quantify the contribution of each dataset partition to the Q-value of a specified state-action pair.

Here is the corresponding Python code snippet for the relevant calls to the SHAP library:

```
1 explainer = shap.KernelExplainer(train, np.zeros_like(
    ↪ dataset_full_episodic))
2 shap_values = explainer.shap_values(dataset_full_episodic)
```

The `train` method must be designed to return the specific value for which the Shapley values are to be computed. In the context of the experiments conducted in this thesis, this value corresponds to the Q-value associated with the state-action pair defined by `state_to_check` and `action_to_check`. For easier interpretation, the Q-values are normalized and presented as probabilities. Specifically, the Q-value for the examined state-action pair is divided by the sum of the Q-values for all possible actions in that state.

After obtaining the Shapley values for all data partitions, the data partitions are grouped based on their influence on the examined state-action pair. The grouping follows the following logic:

Let max_val = max (|shapley_values|)

$$\text{group\_assignment} = \begin{cases} \text{"strong\_positive"} & \text{if value} \geq \text{max\_val} \cdot \left(1 - \frac{2}{5}\right), \\ \text{"positive"} & \text{if value} \geq \text{max\_val} \cdot \left(1 - \frac{4}{5}\right), \\ \text{"neutral"} & \text{if value} \geq -\text{max\_val} \cdot \left(1 - \frac{4}{5}\right), \\ \text{"negative"} & \text{if value} \geq -\text{max\_val} \cdot \left(1 - \frac{2}{5}\right), \\ \text{"strong\_negative"} & \text{otherwise.} \end{cases}$$

Subsequently, for each group, a collection of the included experiences is obtained. For each of these experiences, the modified KL divergence is computed, as described in Section 4.3.2.2.

5.4 Industrial Benchmark

5.4.1 Environment

The environment for the IB experiment is pre-implemented and accessible through a Gym Wrapper.

The code snippet below shows the initialization of the IB environment. The variable `setpoint` corresponds to the setpoint currently under evaluation, which could either be one from the `setpoints` parameter or the one specified in `setpoint_for_evaluation_environment`. The `observation_type` is configured as `include_past` to incorporate observations from `n_past_timesteps` previous timesteps into the current observation. All other parameters of the IB environment are left at their default value.

```
1 env = IBGym(setpoint=setpoint,
2              observation_type="include_past",
3              n_past_timesteps=n_past_timesteps)
```

5.4.2 Training Data Generation

As in the grid world experiment, data generation is executed by a random agent. Following the data collection methodology proposed by Hein et al. [27], the agent accumulates transition data for each setpoint listed in `setpoints`. Specifically, the agent performs `n_episodes` episodes of transition collection, each consisting of `steps_per_episode` steps. Given the environment's continuing nature, episodes will not stop earlier than the defined number of steps.

At the start of each episode, the steerings velocity v and gain g are set back to their default values of 50. shift h is initialized to be within the safe zone, in accordance with Equation 4.10. For h to stay in the safe zone, the portion of the actions corresponding to the change in h is always set to zero.

5.4.3 Training

The implementation of the NFQ algorithm for IB follows the same principles as the Implementation in the grid world experiment and is based on the implementation by Hein et al. [29].

In the IB environment, the action space is inherently continuous, covering the interval $[-1, 1]$ for each steering variable. To facilitate the application of the NFQ algorithm, this continuous action space is discretized. Specifically, for the scope of this thesis, each steering variable is discretized into the set $\{-1.0, -0.9, \ldots, 1.0\}$, thereby converting the continuous action space into a discrete set suitable for NFQ.

The architecture of the QNet comprises an input layer with a dimensionality of $2+5\cdot$`n_past_timesteps`. The term 2 accounts for the action space dimensions, excluding h, which remains constant at zero and is therefore not learned. The term 5 represents the dimension of the observation space, again excluding the observation for h. This observation space dimension is then scaled by the number of past timesteps included, denoted by `n_past_timesteps`. Following the input layer, a sigmoid activation function is applied, followed by a hidden layer containing $20\cdot$`n_past_timesteps` neurons. The output layer consists of a single neuron corresponding to the resulting Q-value of the input state-action pair.

The training procedure largely mirrors the approach used for the grid world environment, described in Section 5.3.3.2. However, a small modification is introduced during the Policy Performance Evaluation step. In this context, the evaluation of the IB policy is as follows: For each setpoint specified in `setpoints`, a distinct evaluation environment is instantiated. The agent, governed by the current policy, then engages with this environment to accumulate rewards over `n_training_evaluation_episodes` episodes, each lasting `n_steps_per_training_evaluation_episode` steps. The metric used for evaluation is the average reward per episode.

5.4.4 Evaluation

An evaluation environment is initialized using the `setpoint_for_evaluation_environment` as the setpoint. The first policy in the list of those trained on the complete dataset is designated as the base policy for comparison. This environment is then navigated by the base policy agent for a predetermined number of steps, specified by `step_for_evaluation`. The state reached at the end of these steps is used as the evaluation state for subsequent analyses. For each trained policy, the action selected at this evaluation state is identified. Additionally, a Q-value heat map is generated to examine the behavior of Q-values across all

possible actions at this particular state.

Parameter	Description
n_episodes	Total number of episodes to generate training data for per setpoint.
steps_per_episode	Number of steps the agent takes per episode for training data generation.
setpoints	List of setpoints used for data generation environments.
n_past_timesteps	Number of past timesteps used in the observation space for the environment.
n_models	Number of models to train for each dataset partition
discount_factor	Discount factor γ for Q-value calculation in training.
learning_rate	Learning rate for policy training.
patience	Number of epochs without improvement before halting the NFQ iteration.
n_iterations	Number of training iterations.
max_epochs_per_iteration	Maximum number of training epochs per iteration.
n_training_evaluation_episodes	Number of episodes the current policy is evaluated for per NFQ iteration for policy selection.
n_steps_per_training_evaluation_episode	Number of steps per evaluation episode.
setpoint_for_evaluation_environment	The setpoints used for the evaluation environment.
step_for_evaluation	The step at which a state is extracted for Q-value evaluation.

Table 5.4: Parameters for the Industrial Benchmark experiment.

Chapter 6

Experiments and Results

6.1 Metrics

6.2 Results

6.2.1 Cluster-Based Classification

In this section, the setup for the conducted cluster-based classification will be described in detail and the results will be presented.

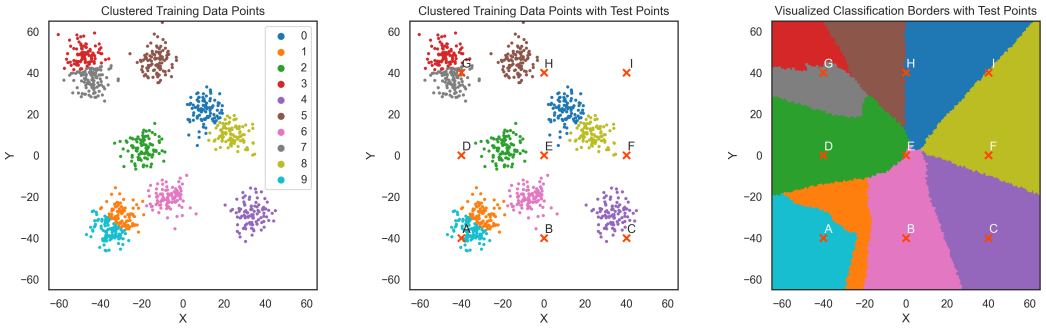
The parameter values for the experiments can be seen in Table 6.1. Besides `grid_cell_size`, all parameters stay the same for all experiments.

Parameter	Value
`n_samples`	1000
`n_clusters`	10
`test_size`	0.2
`learning_rate`	0.001
`patience`	50
`n_models`	10
`n_test_points`	9
`n_grid_cells_per_dimension`	Varying per experiment

Table 6.1: Parameters values for the cluster-based classification experiment.

The experimental setup is organized as follows:

1. Cluster centers are generated within a region that spans from -50 to 50 along both the x- and y-axes. These points are subsequently grouped into



(a) 1,000 generated training data points clustered into 10 clusters.

(b) Training data points with test points.

(c) Classification regions as determined by a trained classification model.

Figure 6.1: Classification experiment setup with clustered training data points (6.1(a)), test points (6.1(b)), and classification regions determined by a trained classification model (6.1(c))

10 distinct clusters, which correspond to the classes $\{0, 1, \ldots, 9\}$. This distribution is illustrated in Figure 6.1(a). The arrangement of these clusters varies, with some isolated and others being right next to each other. To accommodate all points, the coordinate axes are extended to range from -65 to 65 for both x and y.

2. Nine test points, labeled A through I, are evenly distributed across the region. These points are depicted in Figure 6.1(b) and will later serve as the basis for evaluating the influence of specific training data regions on the classification scores of these points.

3. After training the models, their performance is assessed by classifying 500,000 uniformly distributed points within the same region where the training data was generated. This allows for a visual inspection of the classification boundaries, as shown in Figure 6.1(c). The figure reveals how each test point is classified by the model trained on the complete dataset. It also highlights points that are centrally located within classification regions as well as those that border multiple regions, offering intriguing avenues for further evaluation.

The base model, trained on the full dataset, classifies the test points as described in Table 6.2.

As defined by the `n_models` parameter, a total of 10 models are trained for each dataset, including both the full and partial datasets. Figure 6.2 illustrates the classification scores for each test point, as determined by the models

Test Point	Class (Color)	Position
A	9 (turquoise)	central
B	6 (pink)	central
C	4 (violet)	central
D	2 (green)	central
E	6 (pink)	near multiple
F	8 (olive)	central
G	7 (gray)	near red
H	0 (blue)	near brown
I	8 (olive)	near blue

Table 6.2: Classification of test points by the base model.

trained on the full dataset. The inherent variability in DNN models, as discussed in Section 4.2, is visible here. Despite being trained on identical data and evaluating the same test points, the models yield slightly different scores. However, the variations observed are within an acceptable range for this study, as evidenced by the relatively small standard error indicated in red.

In the upcoming sections, three distinct experiments are presented, each varying the grid cell sizes used for partitioning the training data. These experiments aim to demonstrate the impact of the size of omitted data segments on classification scores.

6.2.1.1 Large-Scale Data Partitions

In the first cluster-based classification experiment, the parameter `n_grid_cells_per_dimension` is set to 3, resulting in a total of 9 grid cells that partition the training data space, as illustrated in Figure 6.3. Notably, the top-right cell contains no data, eliminating the need for a specialized model for this cell, as it would just be a model trained on the full dataset, that already exists. Therefore, to implement the deletion diagnostics approach in this experiment, 80 partial models (8 grid cells times 10 models per cell) and 10 models trained on the full dataset are required.

In the application of deletion diagnostics, the impact of each grid cell on the classification score for the correct class, as determined by the model trained on the full dataset, is assessed. Figure 6.4 highlights the three most influential grid cells for each test point.

For the majority of test points, besides point G, the top-left and bottom-left cells have the highest influence. These cells are densely populated with data points, potentially explaining their significant impact on classification scores.

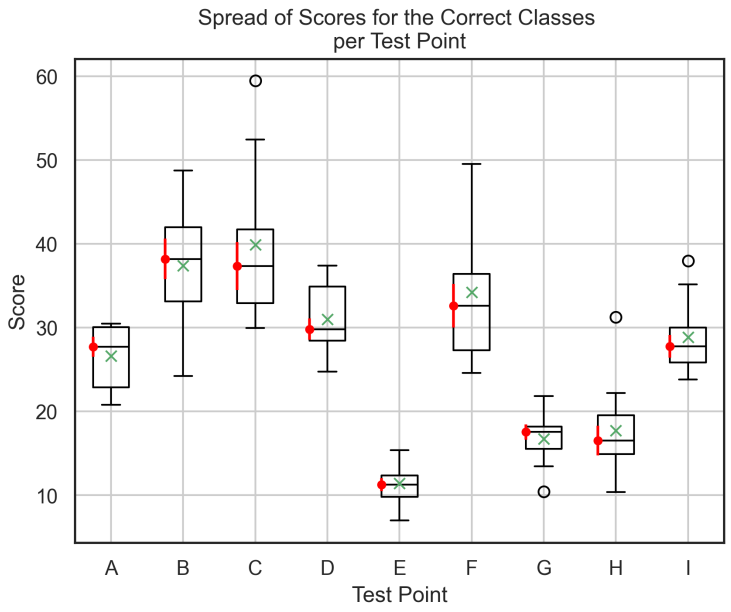


Figure 6.2: Boxplot showing the scores for the correct class per test point classified by the models trained on full training data. Outliers are represented by the "o" markers. Mean values are denoted by green "X" markers, while the standard error is highlighted in red.

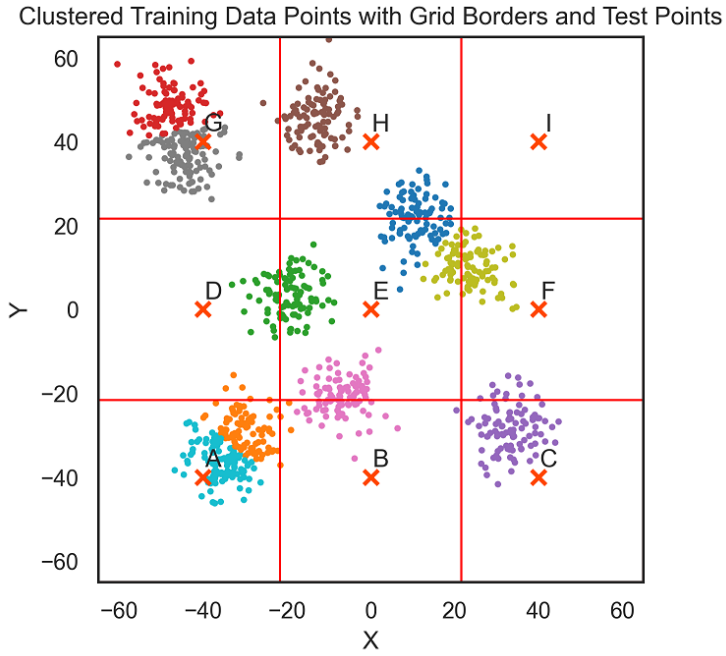


Figure 6.3: Training data space partitioned into 9 grid cells.

Interestingly, the top-left cell is not among the top three influence cells for the test point located within it.

The top-center cell ranks among the top influence cells for six out of the nine test points. For the remaining three points, the center cell takes its place among the top three. The center cell is not only densely populated but also contains a diverse set of classes, possibly aiding the model in differentiation.

Grid cells that are highly influential tend to be densely populated, while those with less influence generally contain fewer data points. This observation suggests that the current grid cell size may be too large. It appears logical that omitting cells with many data points would have a substantial effect on the training outcome.

To validate that the influence scores are not just driven by outliers in the classification scores, a box plot for each test point is generated, as shown in Figure 6.5. This plot illustrates the classification scores for the correct class, as identified by the model trained on the full dataset. If the scores from models trained on data with omitted grid cells closely overlapped with those from the full-data model, the influence scores would lose their significance. However, as evident from the figure, certain grid cells, when excluded, yield classification scores that either do not overlap or only slightly overlap with the

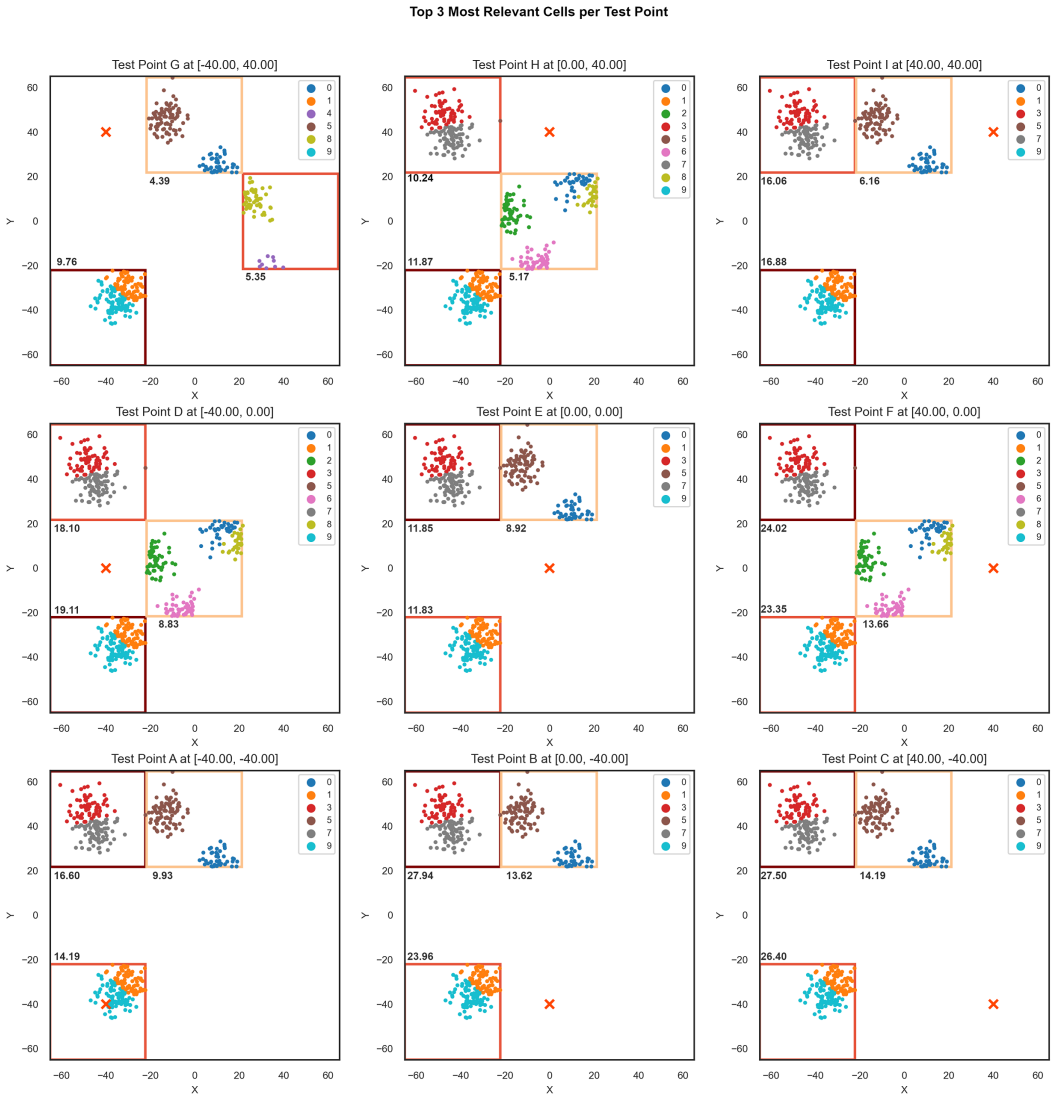


Figure 6.4: Top three influential grid cells for each of the test points with the training data space separated into 9 grid cells. Each grid cell has its influence score associated with it.

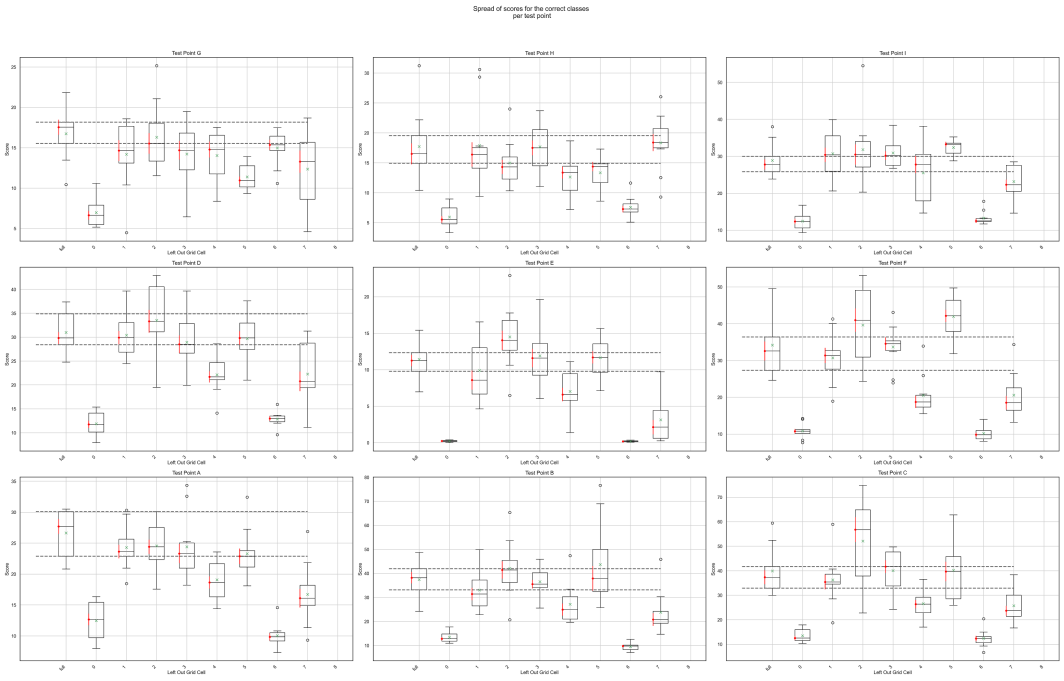


Figure 6.5: Box plots illustrating the classification scores for the correct classes of various test points, as determined by the model trained on the full dataset, for the partition of the training space into 9 grid cells. The numbering of the omitted grid cells progresses from the bottom left to the top right cells. Dashed lines indicate the range of the box for the full data model.

scores generated by the models trained on the complete dataset.

6.2.1.2 Medium-Scale Data Partitions

For the second cluster-based classification experiment, the parameter `n_grid_cells_per_dimensio` is set to 6, resulting in a total of 36 grid cells. The partitioning is shown in Figure 6.6. Out of the 36 cells, only 22 contain data points and thus require models to be trained.

The result from the application of deletion diagnostics can be seen in Figure 6.7, again highlighting the top three most influential training data cells for the test points.

For seven out of the nine test points, the grid cell located near the bottom left, which contains groups 1 and 9 (orange and turquoise), ranks among the most influential. Given the grid configuration shown in Figure 6.6, this particular cell contains nearly two complete clusters, justifying its significant influence on the classification.

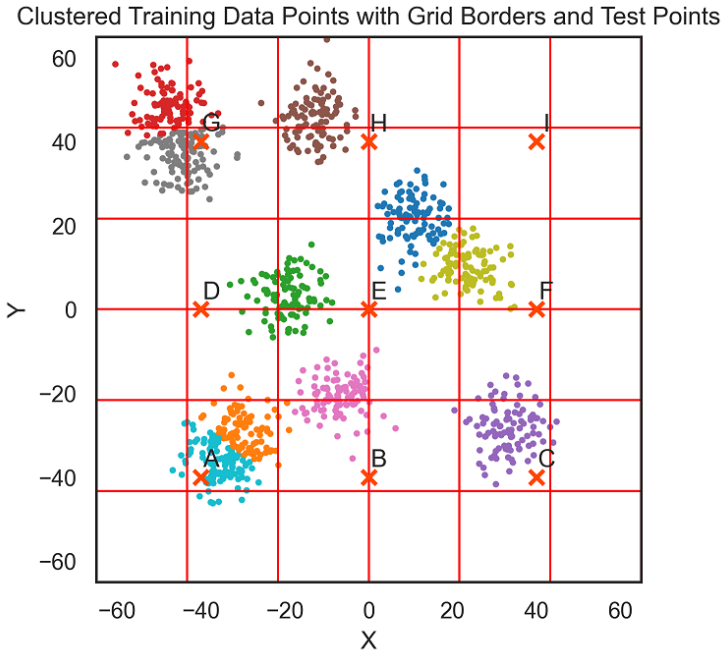


Figure 6.6: Training data space partitioned into 36 grid cells.

Similarly, for seven out of the nine test points, cells in the center containing class 2 (green) are among the top three most influential cells.

Notable peculiarities for each test point are as follows:

- **A:** Classified as turquoise (9). Interestingly, none of the influential cells contain data for this class. This is one of only two test points where the orange/turquoise clusters are not among the most influential.
- **B:** Classified as pink. The adjacent cell containing pink data is influential.
- **C:** Classified as purple. Influential cells include those with green and orange/turquoise data.
- **D:** Classified as green. Contains a significant amount of green data in the influential cells.
- **E:** Classified as pink. Influential cells include two with green and pink data, which might help differentiate between classes.
- **F:** Classified as olive. Influential cells include one with both olive and blue data.

- **G:** Classified as gray. A cell with red data is directly above, potentially aiding in differentiation between the red and gray classes.
- **H:** Classified as blue. Influential cells include one with both blue and olive data, potentially aiding in differentiation.
- **I:** Classified as olive. Influential cells include one with both blue and olive data, potentially aiding in differentiation.

For certain test points, it appears that the most influential cells are those that aid in distinguishing between adjacent classes.

Again, to validate the meaningfulness of the influence scores, additional boxplots are generated to display the classification scores for each test point, as illustrated in Figure 6.8. Unlike the previous experiment involving the partition into 9 cells, this setup shows instances where the scores overlap, notably for test points A and D. However, there remain grid cells where the scores mostly fall below the first quartile of those generated by models trained on the full dataset. For instance, while cells 10, 23, and 36 for test point A are not entirely below the first quartile, a significant portion of their data points do fall below it. Although this lowers the meaningfulness of the influence score, it provides valuable insights by revealing that reducing the size of the omitted training data partitions diminishes their impact on the classification score.

6.2.1.3 Small-Scale Data Partitions

Further reducing the size of the grid cells is experiment three in the cluster-based classification setup. `n_grid_cells_per_dimension` is now set to 10. Dividing the training data space into 100 cells as shown in Figure 6.9. Out of the 100 cells, only 49 contain data points, requiring models to be trained.

Figure 6.10 presents the five most influential training data cells for each test point, as determined by deletion diagnostics. The focus is shifted to the top five influential cells, given the relatively small size of the data partitions.

For every test point, a cell located near the bottom left, which contains some data points from class 9 (turquoise), consistently is among the most influential.

Five out of the nine test points feature a cell near the top left as highly influential. This cell contains data points from classes 3 and 7 (red and gray) and is populated very densely.

Certain influential cells contain only very few data points, a phenomenon that could be attributed to training variability. The absence of noteworthy findings among the identified influential partitions may be a consequence of choosing too small partition sizes, a notion further confirmed by the boxplots presented in Figure 6.11.

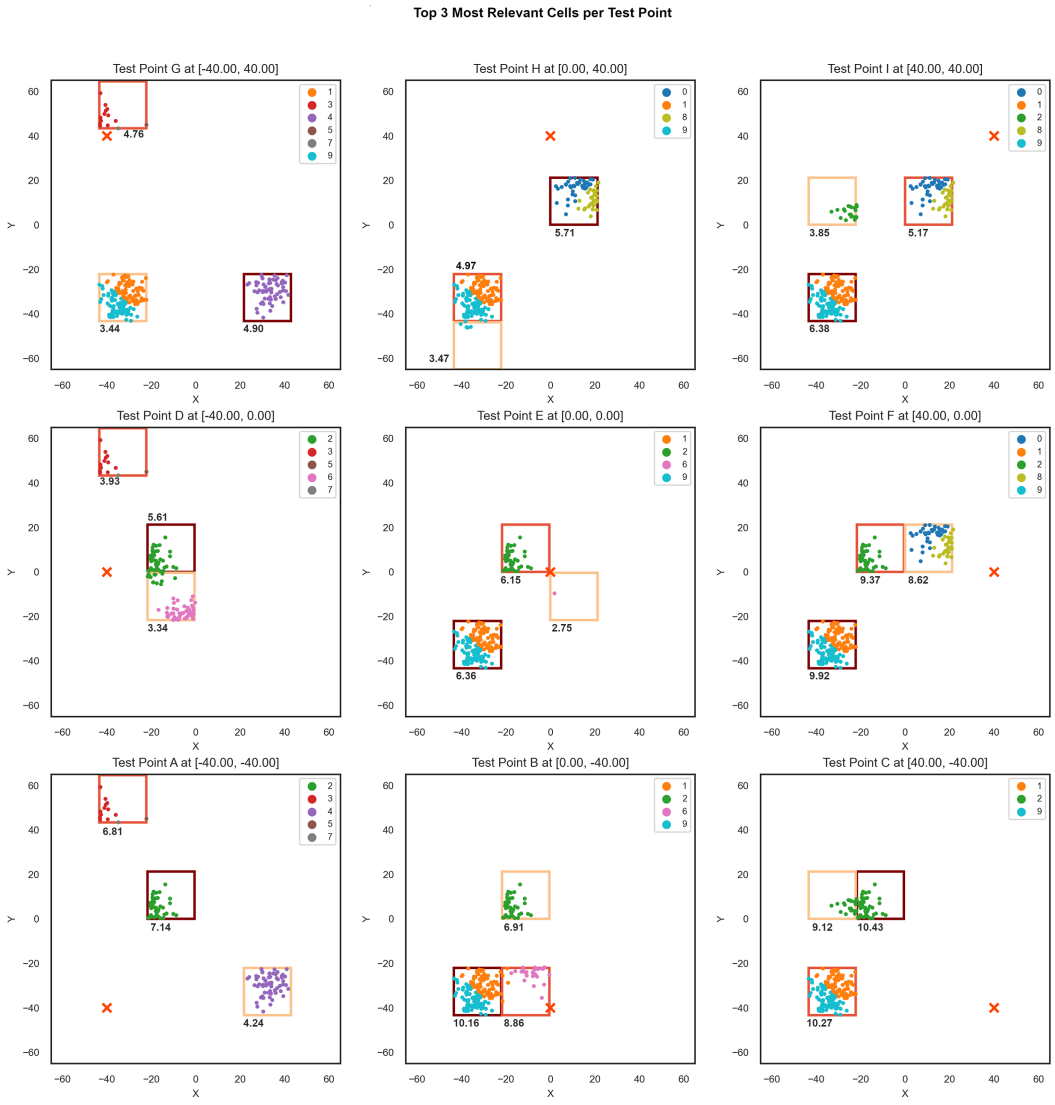


Figure 6.7: Top three influential grid cells for each of the test points with the training data space separated into 36 grid cells. Each grid cell has its influence score associated with it.

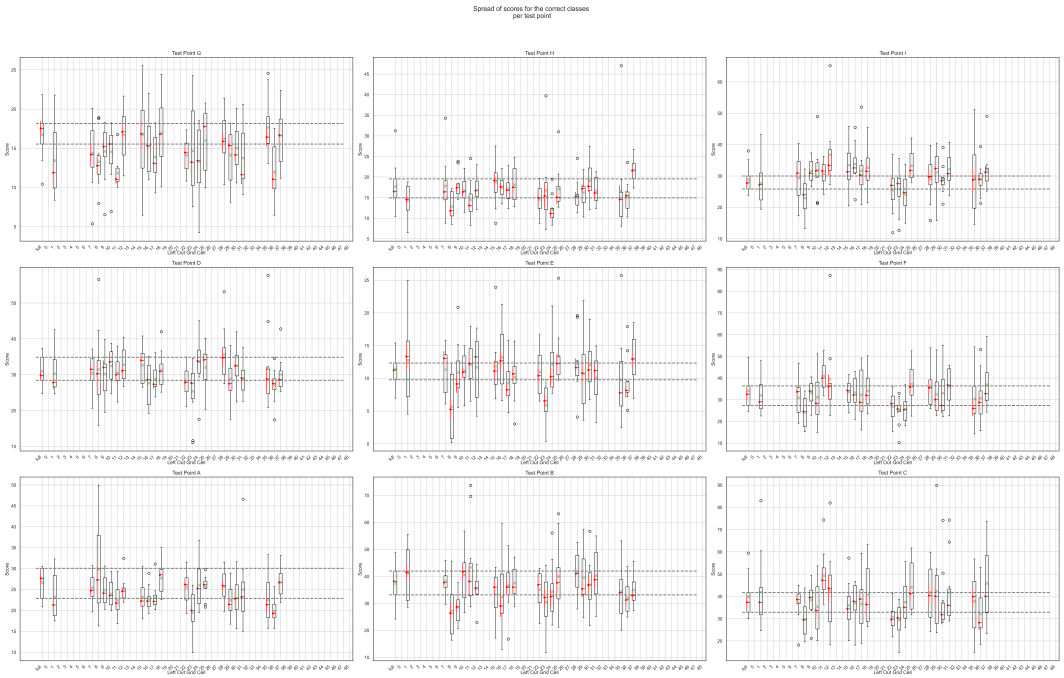


Figure 6.8: Box plots illustrating the classification scores for the correct classes of various test points, as determined by the model trained on the full dataset, for the partition of the training space into 36 grid cells. The numbering of the omitted grid cells progresses from the bottom left to the top right cells. Dashed lines indicate the range of the box for the full data model.

Figure 6.9: Training data space partitioned into 100 grid cells.

The boxplots depicted in Figure 6.11 indicate high variations and increased standard errors among models trained on identical datasets. There is considerable overlap between scores from full-data and partial-data models. Such overlap suggests that overly fine-grained partitioning of the data could compromise the reliability of the influence scores.

6.2.2 Grid World

The experiments in the grid world environment are designed to explain the rationale behind the agent's directional choices based on its training data experiences.

Figure 6.12 illustrates the grid world layout used for the experiments. The agent starts each episode at field 20, located at the bottom left corner, and is faced with the following decision: it can either go up towards the goal state at field 0, or go right towards the teleportation field at 22. The latter offers a chance to be transported to the goal state based on a predefined probability.

Opting for the upward path to the goal state is a longer but risk-free route. Conversely, the rightward path to the teleportation field is shorter but comes with the risk of failed teleportation attempts.

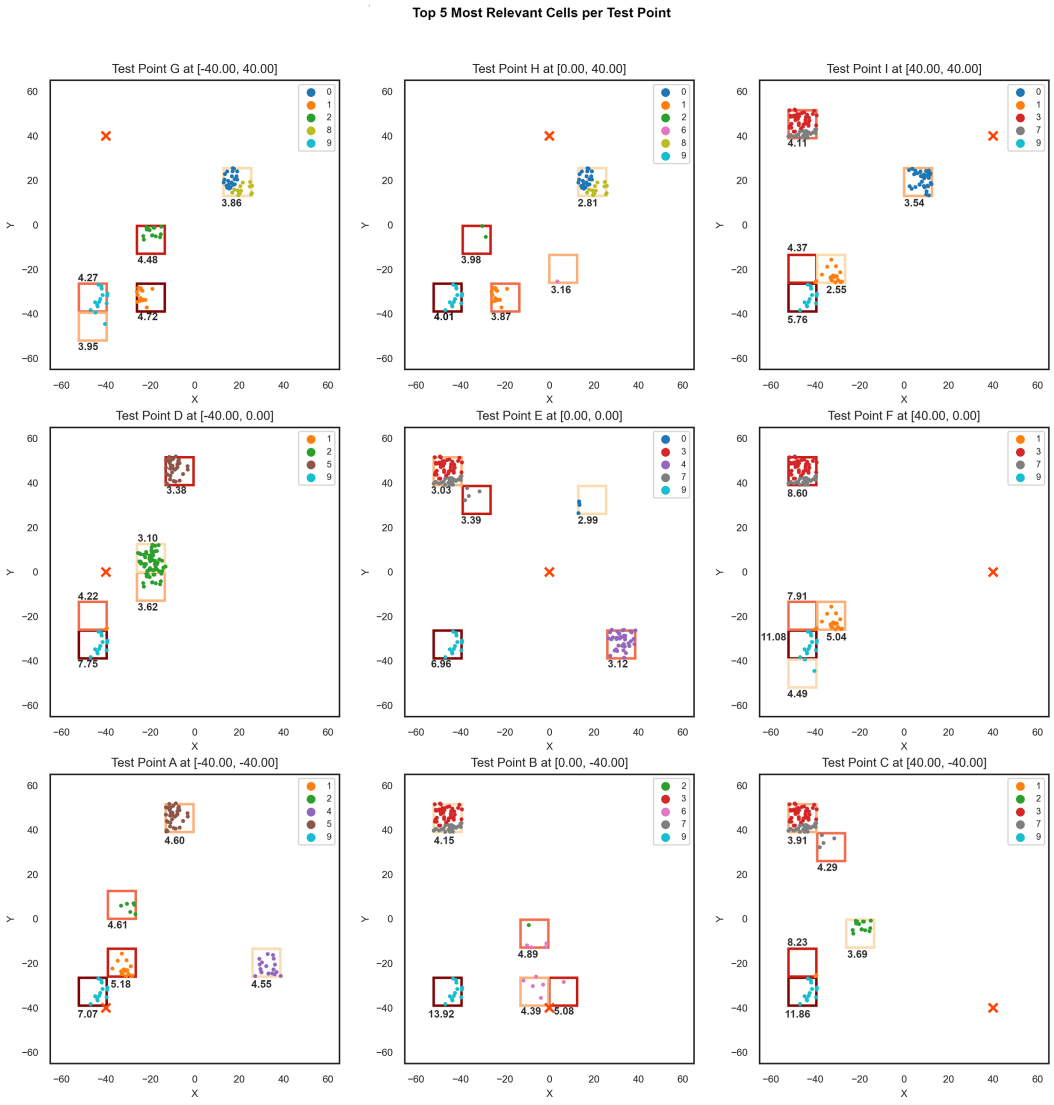


Figure 6.10: Top five influential grid cells for each of the test points with the training data space separated into 100 grid cells. Each grid cell has its influence score associated with it.

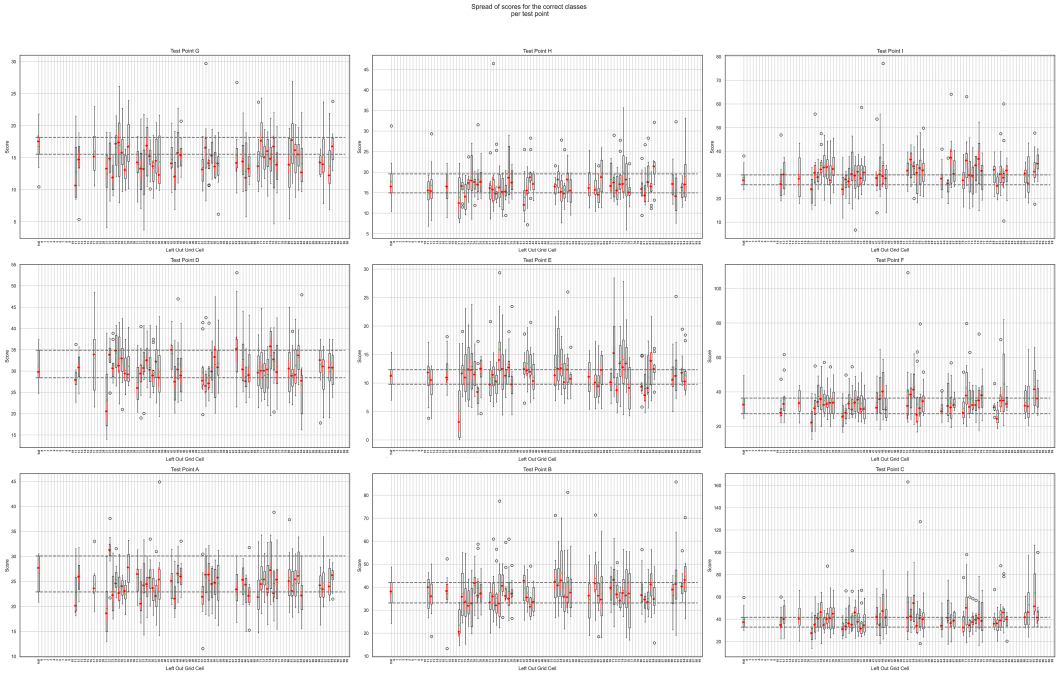


Figure 6.11: Box plots illustrating the classification scores for the correct classes of various test points, as determined by the model trained on the full dataset, for the partition of the training space into 100 grid cells. The numbering of the omitted grid cells progresses from the bottom left to the top right cells. Dashed lines indicate the range of the box for the full data model.

Figure 6.12: A 5×5 grid world for the conducted experiments. The agent starts at the start field 20. 0 is the goal state and 22 is a teleportation field that teleports the agent to the goal state with a specified teleportation probability.

This experimental setup enables the investigation of how different training data sets influence the agent's decision-making, specifically whether it opts for the riskier teleportation route or the safer, direct path to the goal state.

Table 6.3 outlines the general parameters configured for the grid world experiments. The parameters `state_to_check` and `action_to_check` specify that the focus of the experiments is to examine the Q-values associated with the agent's decision to move right towards the teleportation field.

Data generation for training is executed for 50 episodes for each teleportation probability specified in `teleport_chances`, with each episode capped at a maximum of 50 steps.

6.2.2.1 Dynamic Programming

The grid world experiments primarily focus on the integration of the dynamic programming approach with Shapley values for deletion diagnostics and modified KL divergence for assessing the relevance of individual training experiences. As detailed in Section 4.3.2.2, DP training adopts a table-based strategy and needs to train only one model per dataset, because of the stability of the training results. For these DP trainings, the `training_algorithm` is configured as `dp`, and the `discount_factor` is set to 0.95.

Parameter	Value
n_episodes	50
max_steps_per_episode	50
experiment_mode	Varying per experiment.
teleport_chances	Varying per experiment.
state_to_check	20
action_to_check	1 (go right)
training_algorithm	Varying per experiment.
discount_factor	Varying per experiment.

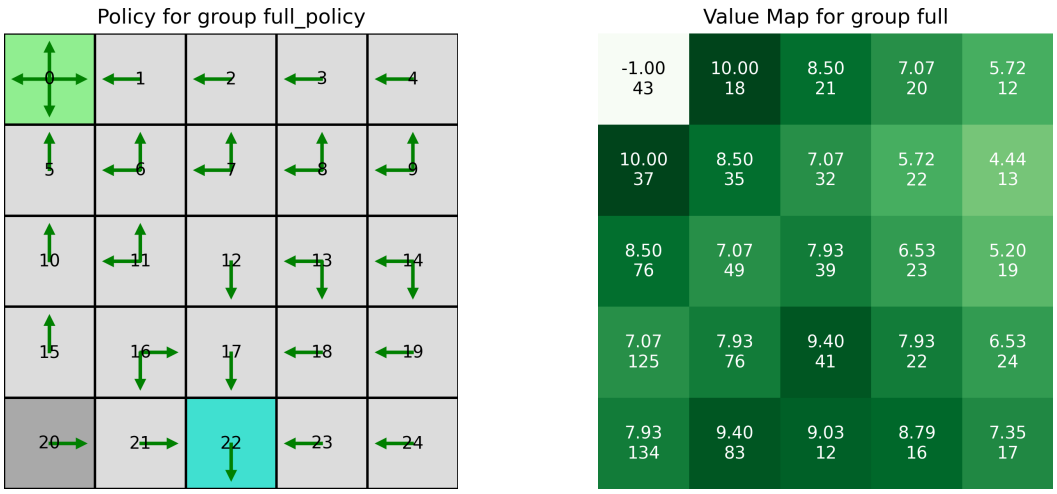
Table 6.3: General parameter values for the grid world experiments.

Episodic Data Generation For experiments utilizing the episodic data generation, `experiment_mode` is set to `episodic` and `teleport_chances` is the single value 0.6, meaning a 60% chance of being teleported to the goal state upon reaching the teleportation field.

The teleportation chance is set at 60% to create a decision-making scenario. The expected reward for going upwards is $3 \cdot (-1) + 10 = 7$. For going rightwards, the expected reward after reaching the teleporter is $\frac{1}{0.6} \cdot (0.6 \cdot 10 + 0.4 \cdot (-1)) - 1 \approx 8.3$. Statistically, the agent should opt for the rightward path to the teleporter, as it offers a higher expected reward of 8.3 compared to 7 for going upwards.

The policy derived from training the agent on the full dataset is illustrated in Figure 6.20(a). Notably, the agent chooses to move right toward the teleporter when starting from field 20. However, when the agent is within a 3-step distance from the goal state 0, it opts for the more secure path, proceeding directly to the goal. The corresponding value map in Figure **??** depicts the Q-values for each field and the frequency with which the agent visited each field during training data generation.

The application of deletion diagnostics using Shapley values is visualized in Figure 6.14 through a SHAP force plot. As outlined in Section 5.3.4, the Q-values are normalized. The force plot indicates that training on the full dataset yields a normalized Q-value of 0.65 for moving rightwards from the starting field. Episodes contributing positively to this Q-value are shown in red to the left of the 0.65 value, while those exerting a negative influence are displayed in blue to the right. The proximity of these episodes to the central value of 0.65 signifies their level of influence. For example, the training data from episode 31 exerts a strong positive influence on the decision to move right from the starting field, whereas the data from episode 32 has a significant negative impact on that same decision.



(a) (b)

Figure 6.13: Policy generated by dynamic programming algorithm in an environment with a 60% teleportation probability in the episodic experimental setup (6.20(a)) and the resulting Q-values of the fields with the number of times the agent landed in the respective field in training data (6.20(b))

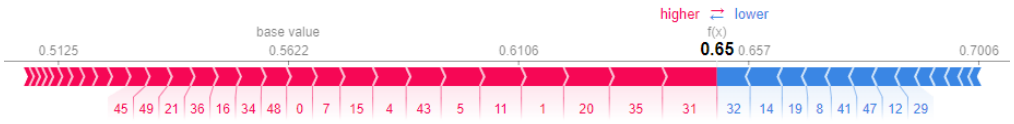


Figure 6.14: Shapley value force plot of the influence on choosing action 1 (moving right) in the starting field in the episodic experiment mode.

For further analysis, the episodes are grouped based on their Shapley value as described in Section 5.3.4. The grouping can be seen in Table 6.4

Influence Group	**Episodes**
`strong_positive`	1, 4, 5, 11, 20, 31, 35, 43
`positive`	0, 7, 15, 16, 18, 21, 26, 27, 34, 36, 37, 45, 48, 49
`neutral`	2, 3, 6, 9, 10, 13, 17, 24, 28, 30, 33, 38, 39, 40, 42, 44, 46
`negative`	9, 12, 14, 19, 22, 23, 25, 29, 41, 47
`strong_negative`	32

Table 6.4: Influence groups determined by Shapley values in episodic experiment setup

For each group, the most relevant experiences are identified based on their modified KL divergences. The figures for each group are organized as follows: On the top, experiences are displayed along with their relevance scores. The experiences on the x-axis are formatted as `(field, action, reward, next_field, is_done)` $\times$ `number_of_occurrences`. The y-axis represents the modified KL divergence, denoted as *Experience-Relevance*. On the bottom-left, for easier comprehension, the nine most relevant experiences are visually represented within the grid. Curved arrows leading to the teleportation field and subsequently to the goal state signify experiences where the agent entered the teleportation field and was successfully teleported. The bottom center shows a policy that is trained exclusively on the data present in the group, and on the bottom right, the value map for that policy with the number of cell visits in each cell is shown.

In the `strong_positive` group, shown in Figure 6.15 every episode features the action of moving from the left into the teleportation field and successfully teleporting, without any instances of teleportation failure. Additionally, the clustering of experiences in the bottom-left corner suggests that the agent consistently reaches the teleportation field with minimal detour. Notably, the experience of moving from the left into the teleportation field has a score of 1.0, signifying that this particular experience is unique to this group.

In the `positive` group, the relevance scores, as shown in Figure 6.16, peak at about 0.06. This low score indicates a lack of experiences that are even moderately unique to this group.

In the `neutral` group, Figure 6.17 shows that the most relevant experiences are `(7, 3, -1, 6, False)` (moving left from field 7) and `(7, 0, -1, 2, False)` (moving upward from field 7). The majority of this group's relevant experiences are located in the grid's upper-right region, distant from both the teleportation field and the goal state. However, a few are closer to the goal

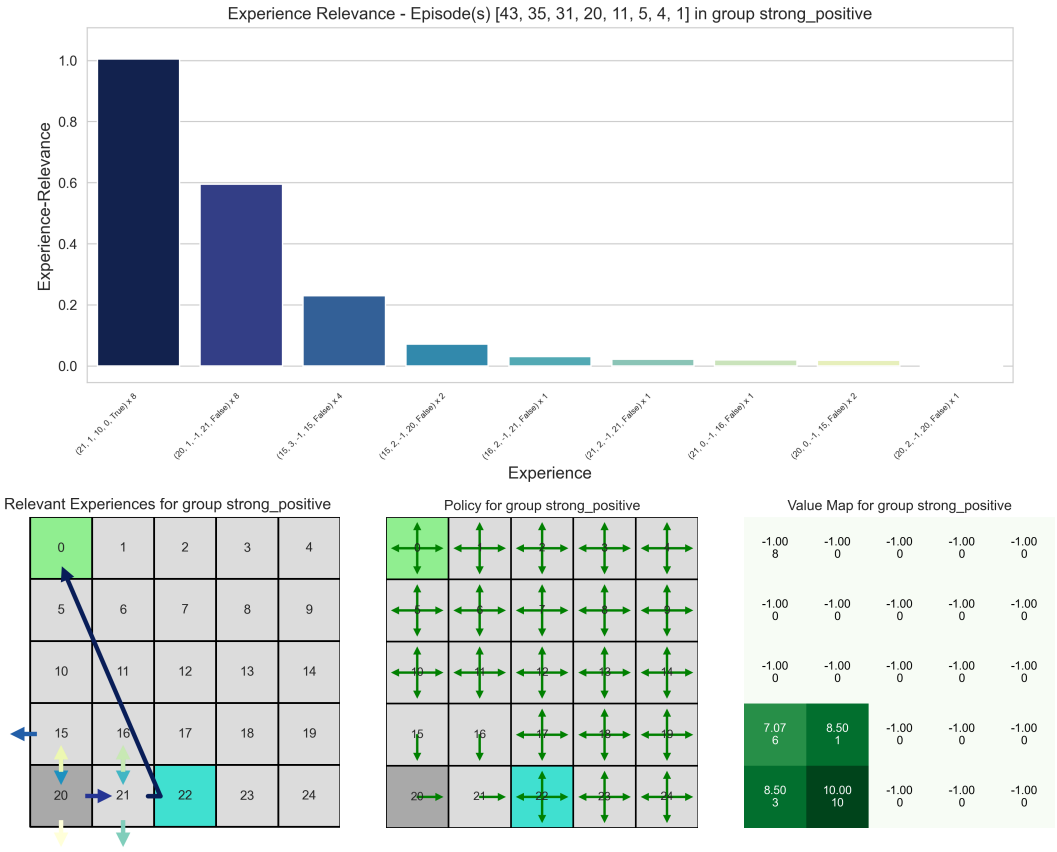


Figure 6.15: Experience relevance for group `strong_positive` in the episodic grid world experiment.

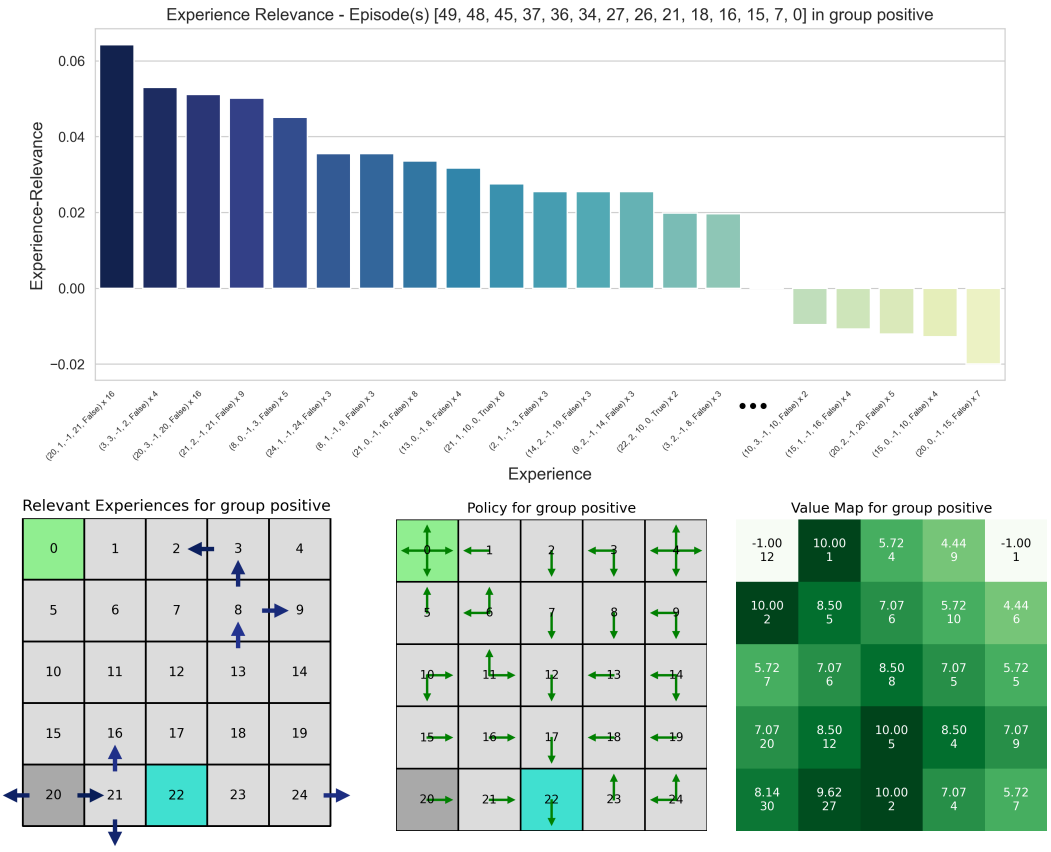


Figure 6.16: Experience relevance for group `positive` in the episodic grid world experiment.

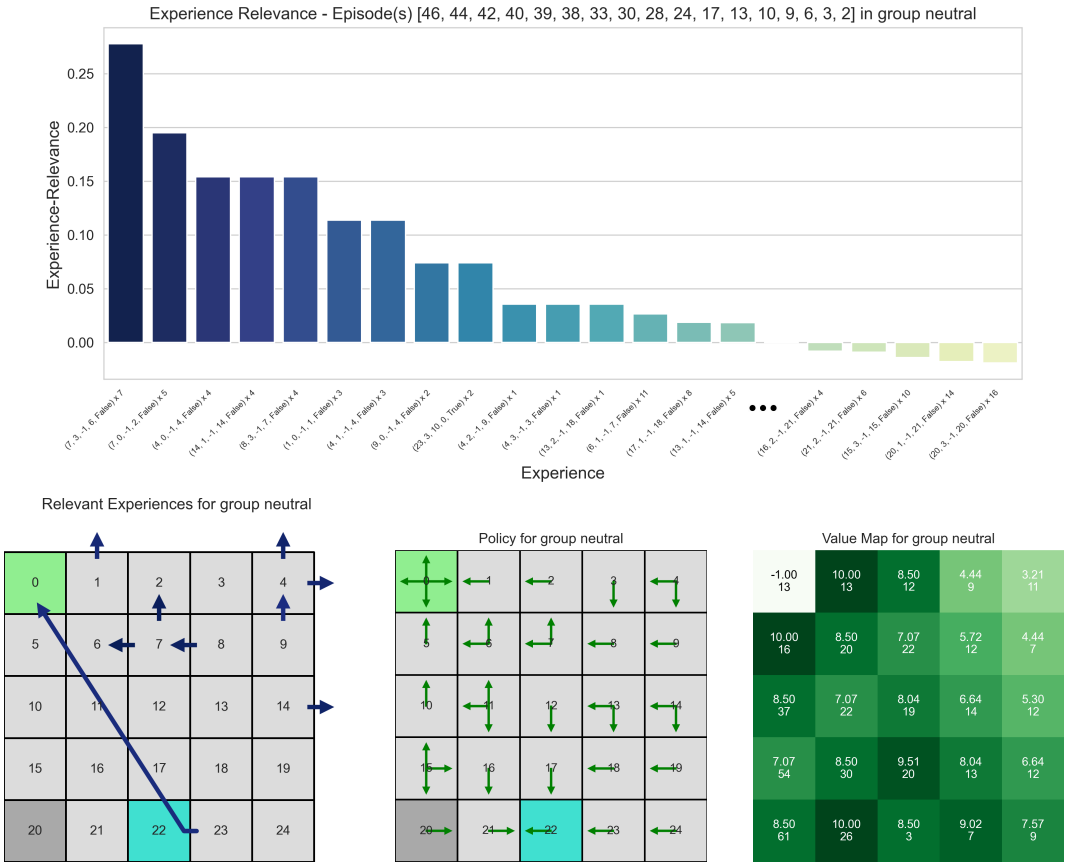


Figure 6.17: Experience relevance for group `neutral` in the episodic grid world experiment.

state, and one experience, occurring twice, involves moving from field 23 into the teleportation field and successfully teleporting.

In the `negative` group, Figure 6.18 shows that the relevance scores are again relatively low. Most of the relevant experiences are concentrated in the bottom-left region of the grid, indicating that the agent frequently moves in this area without successfully reaching and teleporting from the teleporter. Additionally, the experience `(5, 0, 10, 0, True)` appears five times, suggesting that in half of this group's episodes, the agent directly moved into the goal state from the field immediately below it.

In the `strong_negative` group, as illustrated in Figure 6.19, the tendency to move in the bottom-left region without successful teleportation is more pronounced. This group, consisting of only a single episode, features the agent entering the teleportation field but failing to teleport. Subsequently, the agent

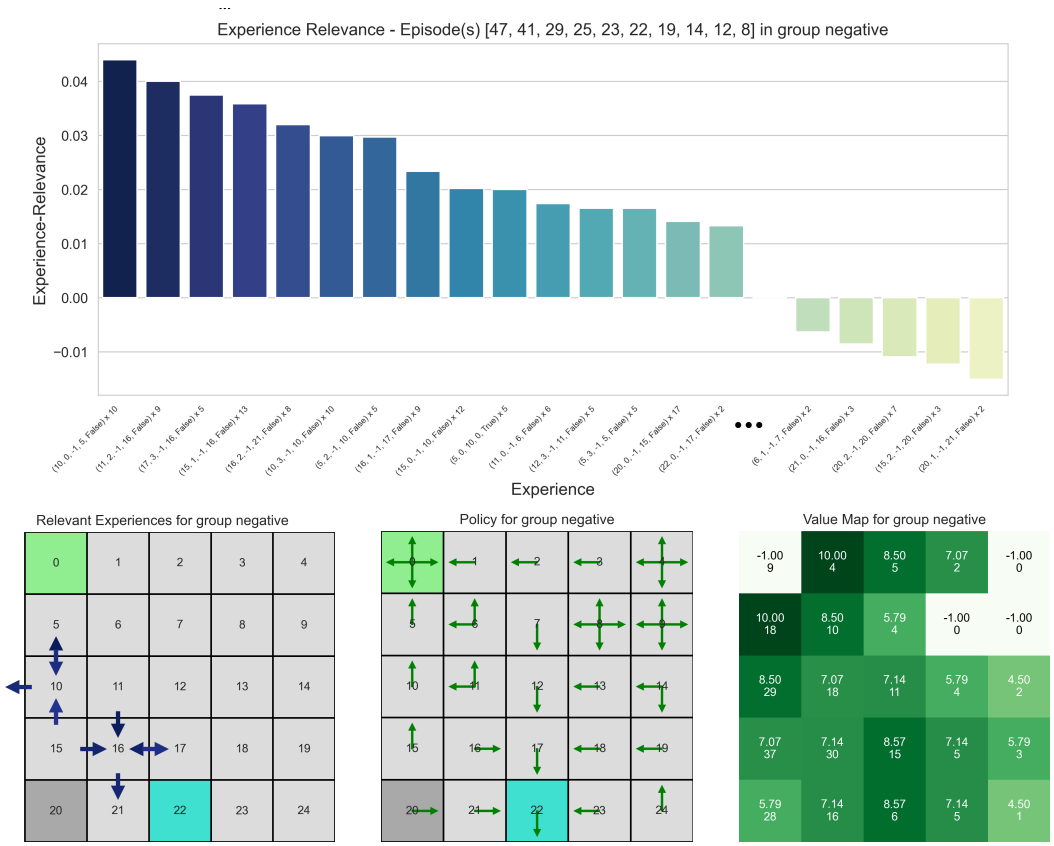


Figure 6.18: Experience relevance for group `negative` in the episodic grid world experiment.

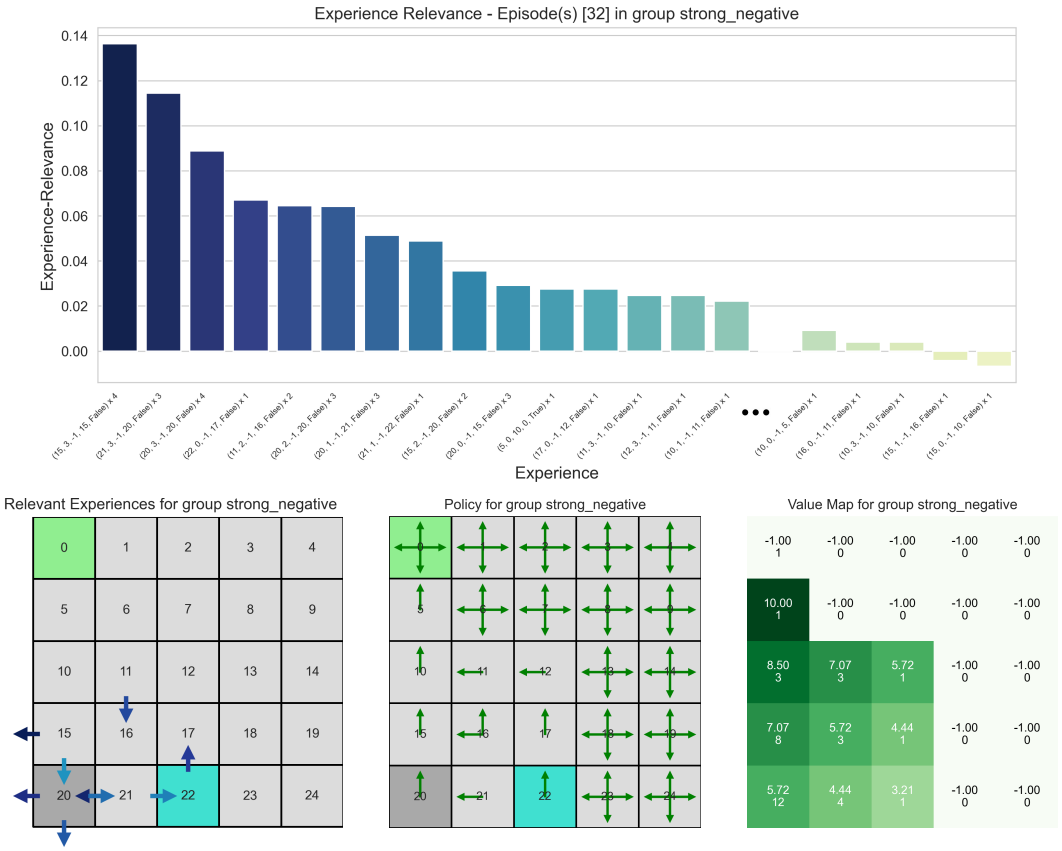
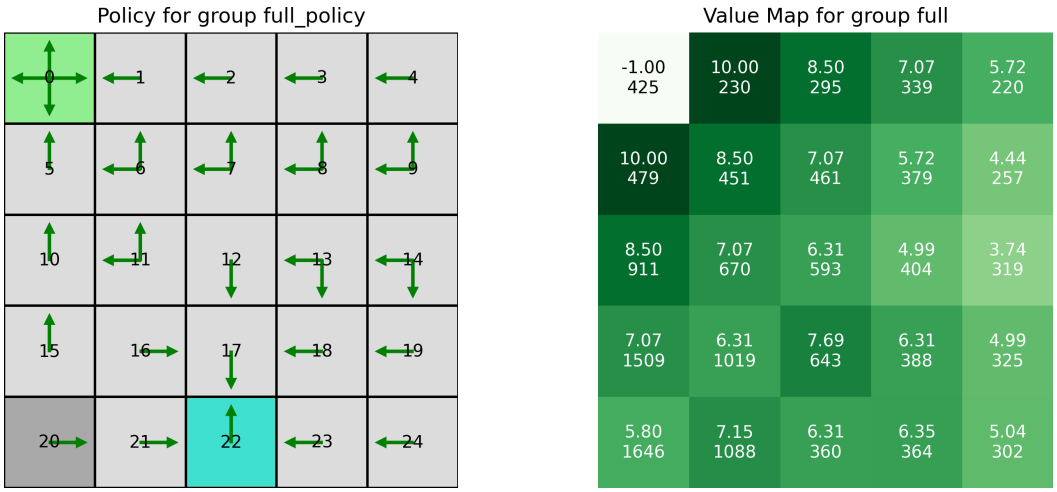


Figure 6.19: Experience relevance for group `strong_negative` in the episodic grid world experiment.

directly moves into the goal state.

Variable Teleport Chances Data Generation In the experiments that employ variable teleport probabilities, the `experiment_mode` is set to `variable_teleport_chances`, and `teleport_chances` is set to $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. This configuration leads to training data generated across 11 distinct environments.

Figure 6.20 displays the policy trained on the complete dataset on the left, accompanied by the corresponding value map on the right. Given the diverse nature of the training data and an average teleportation probability of 0.5 in the full dataset, the value for moving right from the starting field slightly surpasses that of moving upward. This results in a policy close to the one generated in the episodic experiment using the full dataset.



(a) (b)

Figure 6.20: Policy generated by dynamic programming algorithm trained with data from environments with teleport chances $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ in the variable teleport chances experimental setup (6.20(a)) and the resulting Q-values of the fields with the number of times the agent landed in the respective field in training data (6.20(b))

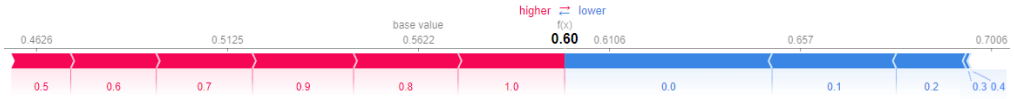


Figure 6.21: Shapley value force plot of the influence on choosing action 1 (moving right) from the starting field in the variable teleport chances experiment mode.

The SHAP force plot in Figure 6.21 presents the Shapley values obtained through the deletion diagnostics approach using SHAP. The figure shows a normalized Q-value of 0.6 for action 1 (moving right) in the starting field when trained on the full dataset. Training data generated in environments with teleport probabilities that positively influence this Q-value are displayed on the left in red. Conversely, those that negatively influence the Q-value are shown on the right in blue. Interestingly, the values are nearly perfectly sorted by their respective teleportation probabilities, ranging from 0.0 as the most negatively influential to 1.0 as the most positively influential.

Again, for more detailed analysis, teleportation probabilities are classified into groups according to their Shapley values, as detailed in Section 5.3.4. This classification is displayed in Table 6.5. Notably, the `strong_positive` group is empty in this scenario. This occurs because the groupings are determined by the highest absolute value, and the negative values in this case have a significantly higher absolute magnitude compared to the positive values.

Influence Group	Teleportation Probabilities
`strong_positive`	
`positive`	0.5, 0.6, 0.7, 0.8, 0.9, 1.0
`neutral`	0.3, 0.4
`negative`	0.1, 0.2
`strong_negative`	0.0

Table 6.5: Influence groups determined by Shapley values in the variable teleport chances experiment setup

The structure of the plots showing the most relevant experiences for each group follows the same guidelines as established for the episodic experiment.

For the `positive` group, Figure 6.22 shows that the majority of significant experiences occur in the grid's bottom-left corner. Specifically, two experiences stand out: `(21, 1, 10, 0, True)`, where the agent moves from the left into the teleportation field and successfully teleports, and `(17, 2, 10, 0, True)`,

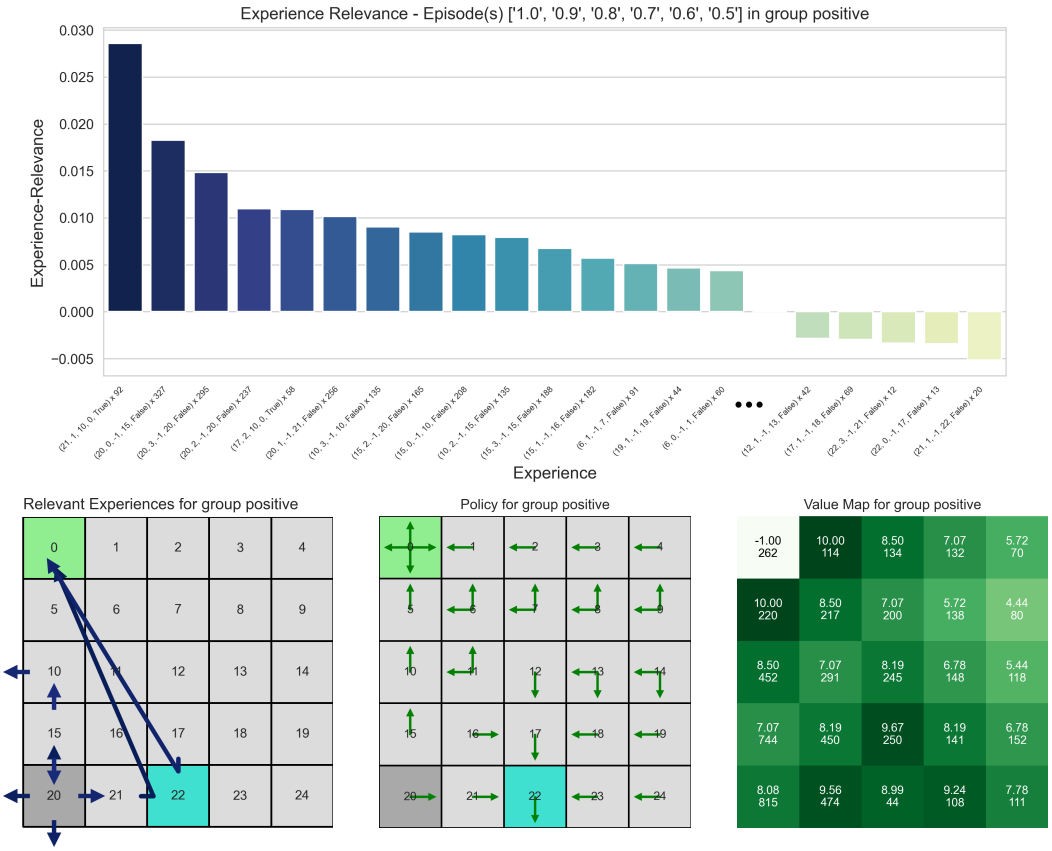


Figure 6.22: Experience relevance for group `positive` in the variable teleport chances grid world experiment.

where the agent enters the teleportation field from the top and also successfully teleports.

In the `neutral` group, Figure 6.23 displays relatively low relevance scores, indicating that the experiences in this group are not particularly unique to it. The majority of these experiences are concentrated in the grid's center. One notable exception is the experience `(22, 2, 10, 0, True)`, where the agent starts at field 22, moves against the bottom wall, essentially re-enters the same field, and successfully triggers the teleportation.

Figure 6.24 displays the relevant experiences for the `negative` group. Notably, most of these experiences are concentrated around the teleportation field. Many of these experiences involve the agent entering the teleportation field but failing to teleport, likely leading the agent to perceive the teleportation field as lacking significant positive value.

The relevant experiences for the `strong_negative` group, as depicted in

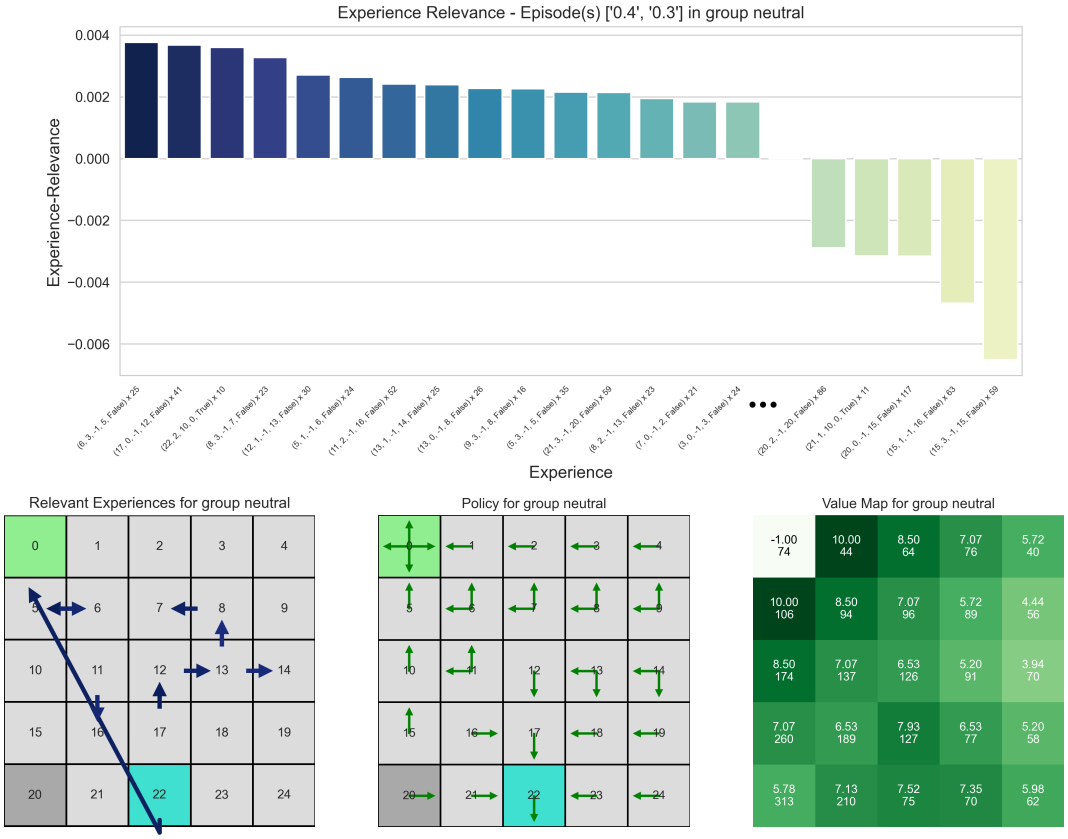


Figure 6.23: Experience relevance for group `neutral` in the variable teleport chances grid world experiment.

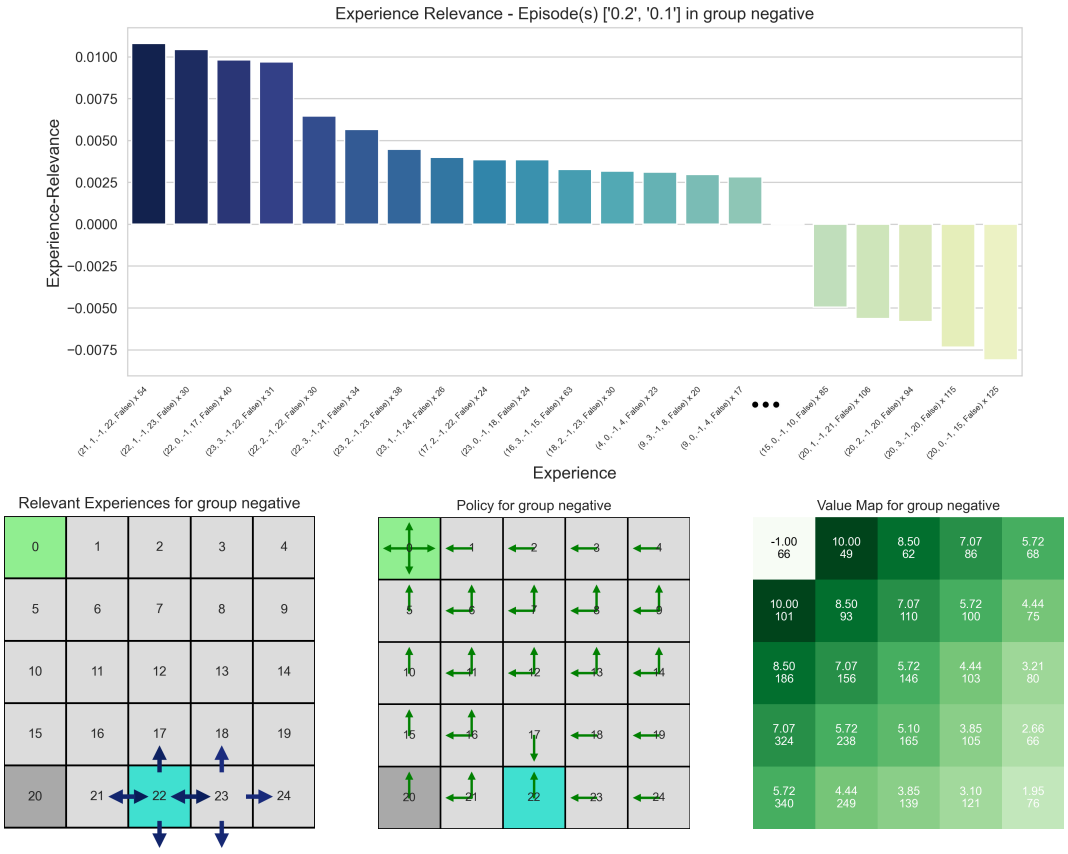


Figure 6.24: Experience relevance for group `negative` in the variable teleport chances grid world experiment.

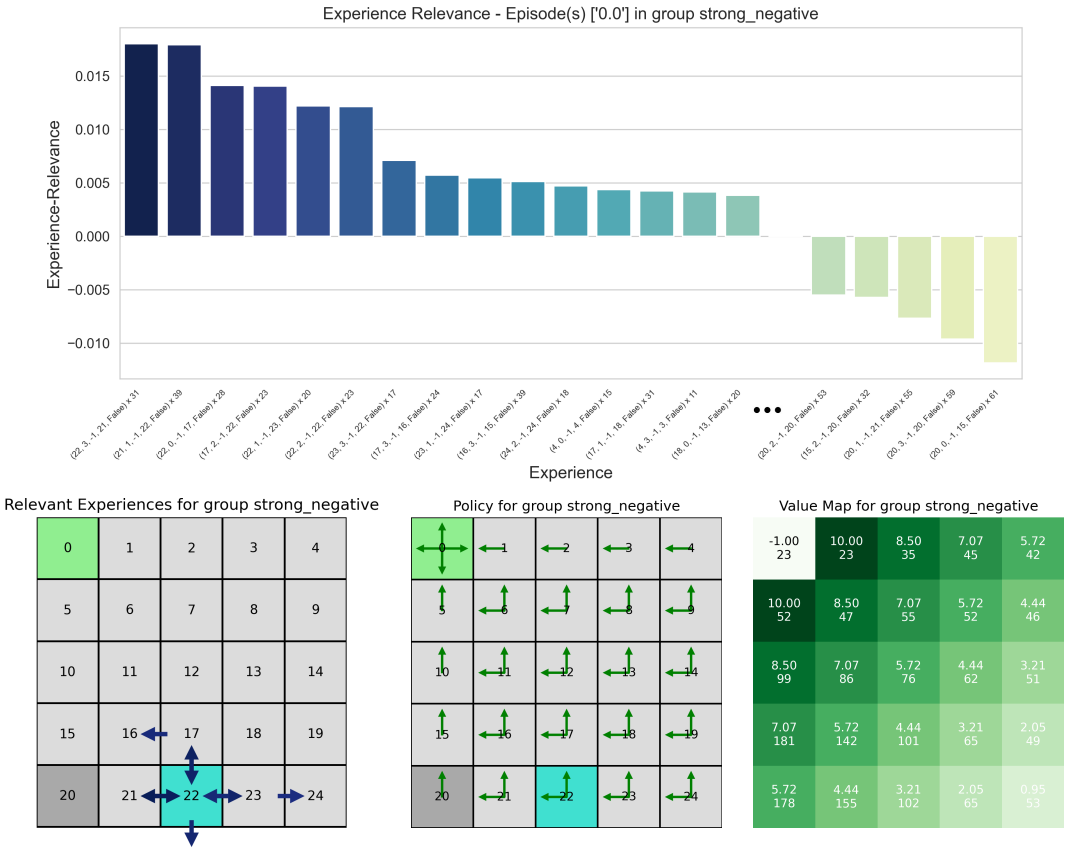


Figure 6.25: Experience relevance for group `strong_negative` in the variable teleport chances grid world experiment.

Figure 6.25, largely mirror those in the `negative` group. The majority of these experiences feature the agent entering and exiting the teleportation field without successfully teleporting.

6.2.2.2 Neural Fitted Q-Iteration

In this initial stage, serving as a foundation for the more complex industrial benchmark environment, only elementary analyses of Q-values, applying the basic deletion diagnostics approach, have been carried out. The analyses target the state-action pair where the agent moves right from the start field under different environment dynamics, i.e. training data gathered from environments with different teleportation probabilities. NFQ-specific parameters for this experiment are presented in Table 6.6.

The general experimental settings include: `experiment_mode` configured to

variable_teleport_chances, training_algorithm set to nfq, and discount_factor adjusted to 0.97.

Parameter	Value
n_models	10
learning_rate	0.1
patience	100
n_iterations	20
max_epochs_per_iteration	10,000
n_evaluation_episodes	10
n_max_steps_per_evaluation_episode	50

Table 6.6: Parameter values specific to the NFQ training for the grid world experiment.

In this section, experiments are conducted under two distinct settings, determined by the value of teleport_chances. The first setting is referred to as *Dense Data Population*, while the second is termed

Sparse Data Population . These configurations are intended to mirror the methodology employed in the cluster-based classification experiments in Section **??**, where the size of the omitted data varied between different experiments.

Dense Data Population In the Dense Data Population experiment, the teleport_chances parameter is configured with a finely-grained range of values: $\{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$. This setup provides a broad spectrum of teleportation probabilities, spanning from 0.0 to 1.0, for the data generation environments.

In this setting, deletion diagnostics yield the Q-values for the examined state-action pair, as illustrated in Figure 6.26. For these DNN-based trainings, the Q-values exhibit significant fluctuations even within models trained on the same dataset. Moreover, the differences between models trained on different datasets are relatively minor. However, a subtle downward trend in the Q-value is observable as the teleportation probability of the omitted dataset increases. Specifically, the mean Q-value is approximately 5.9 when data with a 0% teleportation probability is excluded, compared to a mean Q-value of about 5.5 when data with a 100% teleportation chance is omitted.

For comparative purposes, the same basic deletion diagnostics methodology was applied using the table-based DP training approach on identical training

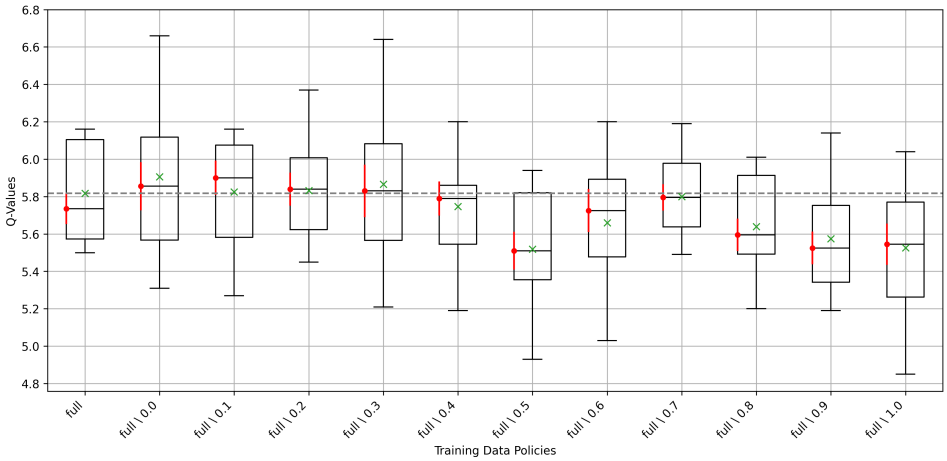


Figure 6.26: Q-values for the action of moving right from the starting field, obtained through NFQ training. The deletion diagnostics were applied to training data generated in grid world environments with varying teleportation probabilities.

datasets. The outcomes are depicted in Figure 6.27. Overall, the Q-values fall within a similar range, although the lower end is slightly higher. The mean Q-value for the model trained on the complete dataset is approximately 5.8, closely aligning with the mean results from the NFQ approach. Notably, the impact of omitting datasets with higher teleportation probabilities is more obvious here. As the teleportation probability of the excluded data increases, the Q-values show a consistent downward trend.

Sparse Data Population The second experiment conducted for the NFQ approach uses less datasets, with `teleport_chances` set to $\{0.0, 0.5, 1.0\}$.

The findings from the NFQ training with deletion diagnostics are displayed in Figure 6.28. The figure highlights that the exclusion of the dataset with a 0% teleportation probability shows a substantial impact on the evaluated Q-value. In contrast, omitting datasets with 50% and 100% teleportation probabilities produces just a slight decrease in the Q-values. Additionally, variations in Q-values are again observed within trainings using identical datasets.

For comparison, again, the experiment was replicated using the DP training approach. The resulting Q-values closely align with the mean values from the NFQ training. This supports the hypothesis that an overly fine-grained data space for deletion diagnostics minimizes the observable impact on the resulting Q-values, whereas a more coarse-grained space is more effective for observing differences.

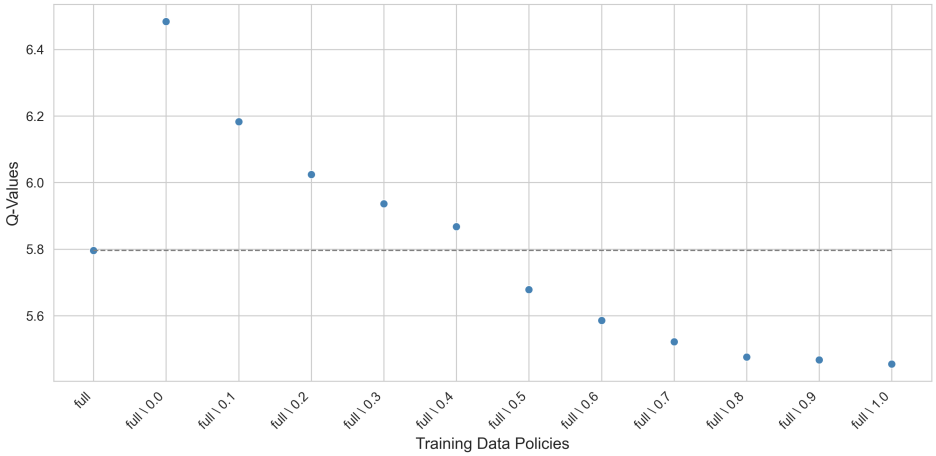


Figure 6.27: Q-values for the action of moving right from the starting field, obtained through DP training. The deletion diagnostics were applied to training data generated in grid world environments with varying teleportation probabilities.

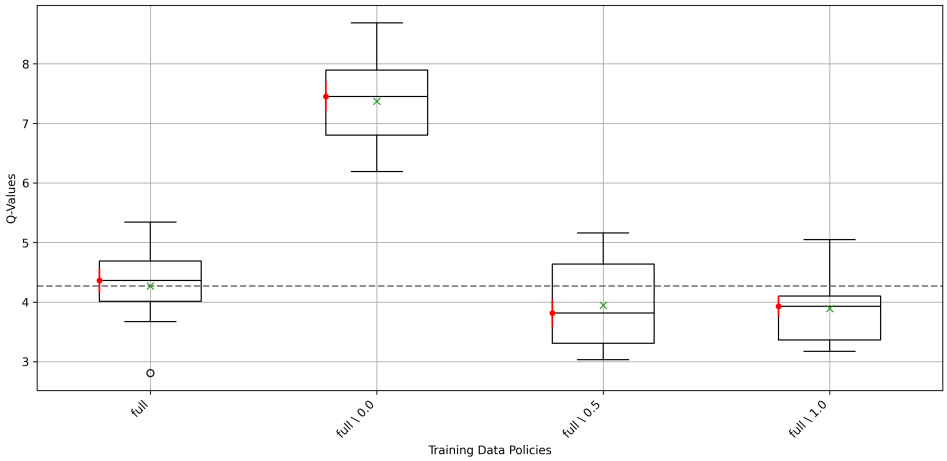


Figure 6.28: Q-values for the action of moving right from the starting field, obtained through NFQ training. The deletion diagnostics were applied to training data generated in grid world environments with varying teleportation probabilities.

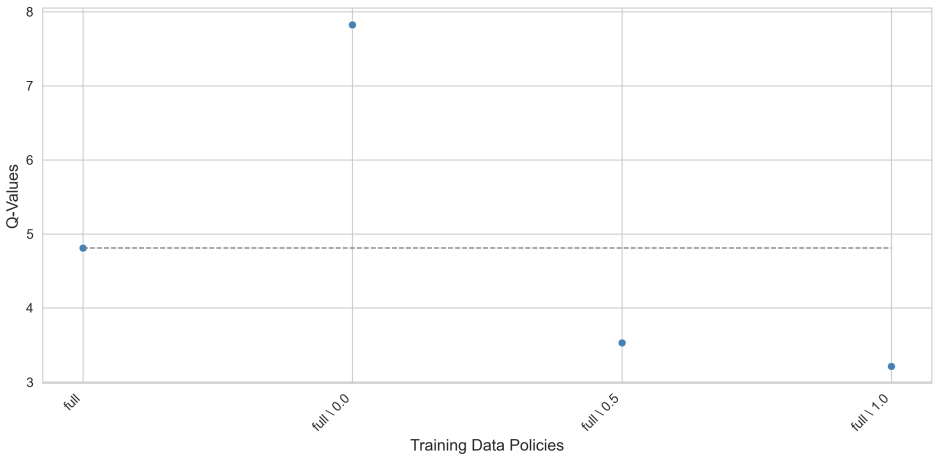


Figure 6.29: Q-values for the action of moving right from the starting field, obtained through DP training. The deletion diagnostics were applied to training data generated in grid world environments with varying teleportation probabilities.

6.2.3 Industrial Benchmark

Average reward comes to about -170 which aligns with findings in [29]

In order to see how the trained policies behave, some state and action trajectories have been created. here and appendix.

Provides a foundation for future research.

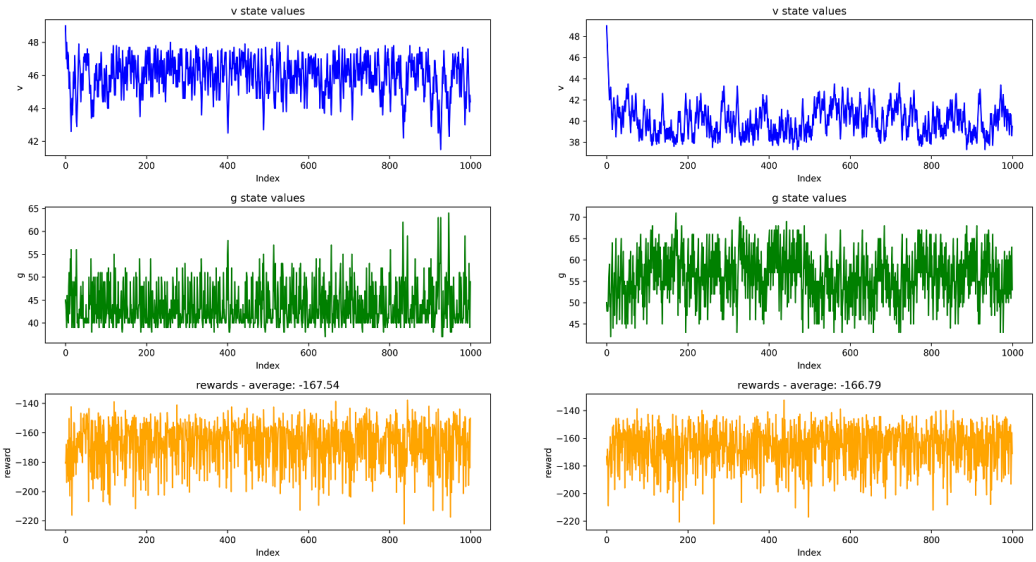
6.3 Discussion

6.3.1 Cluster-Based Classification

6.3.2 Grid World

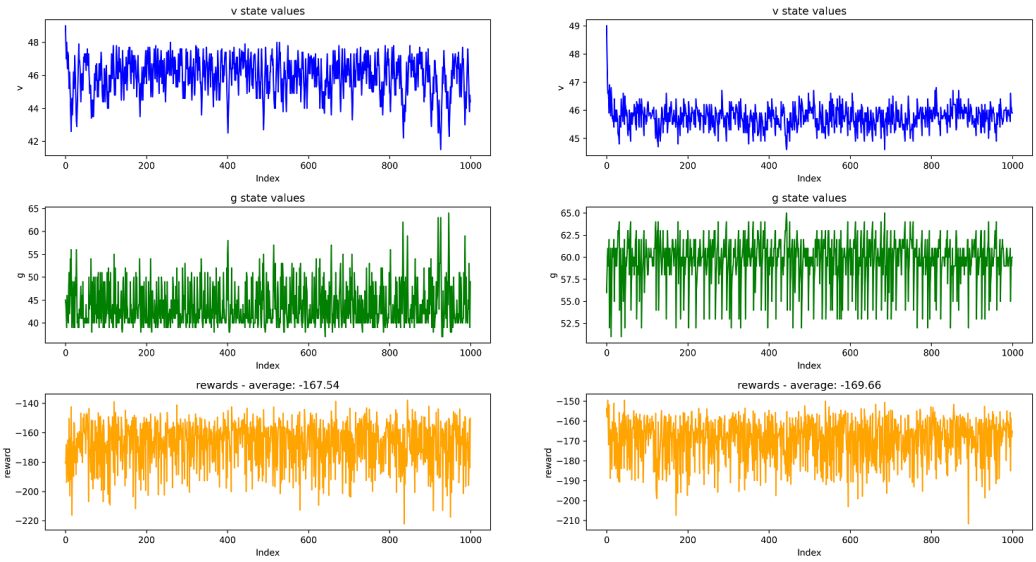
6.3.3 Industrial Benchmark

6.3.4 Limitations of the Study



(a) TODO. (b) TODO.

Figure 6.30: TODO.



(a) TODO. (b) TODO.

Figure 6.31: TODO.

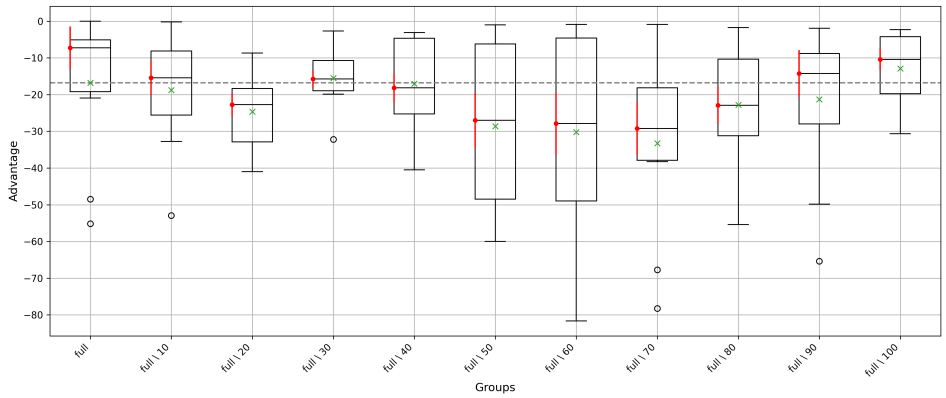


Figure 6.32: TODO.

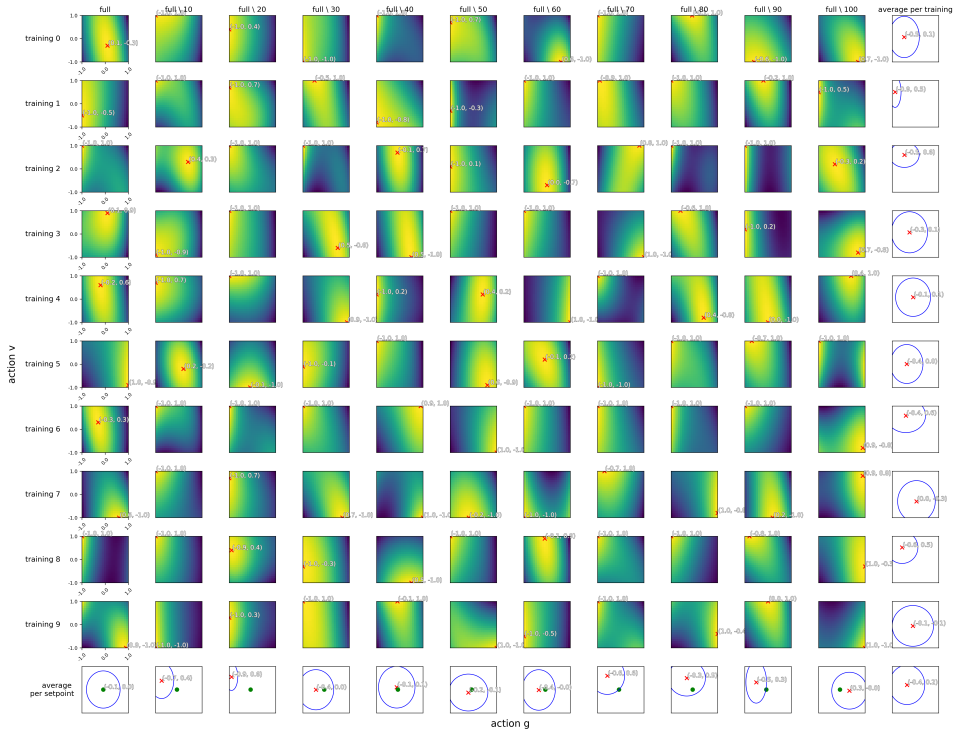


Figure 6.33: TODO.

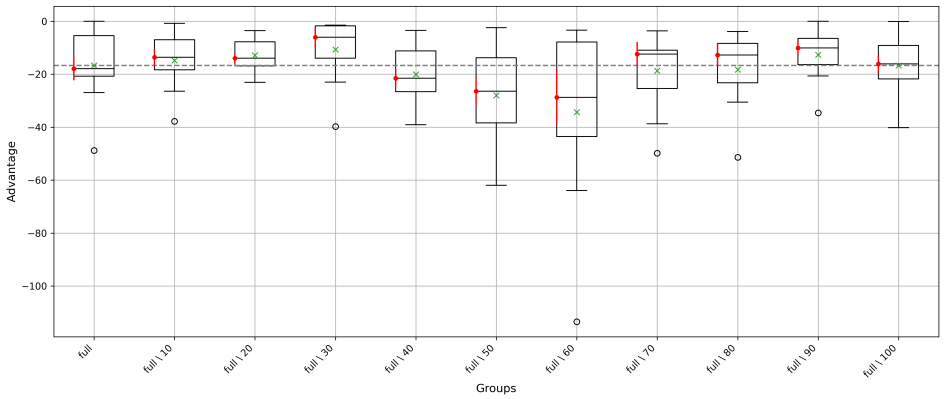


Figure 6.34: TODO.

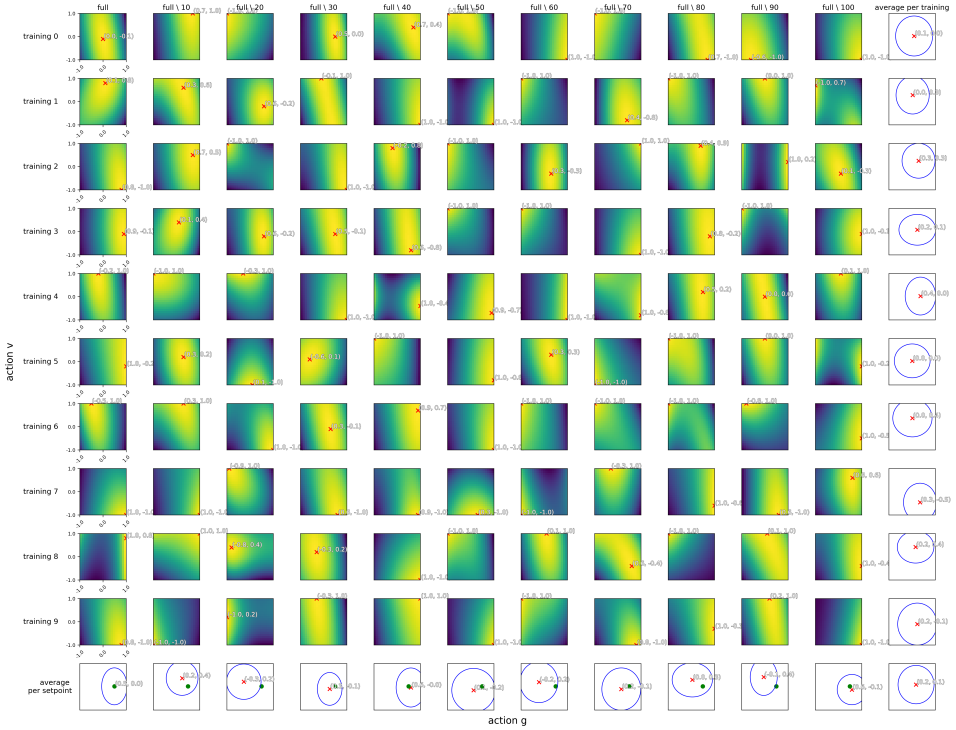


Figure 6.35: TODO.

Parameter	Value
n_episodes	10
steps_per_episode	1,000
setpoints	Varying per experiment.
n_past_timesteps	10
n_models	10
discount_factor	0.99
learning_rate	0.1
patience	100
n_iterations	100
max_epochs_per_iteration	10,000
n_training_evaluation _episodes	10
n_steps_per_training _evaluation_episode	1,000
setpoint_for_evaluation _environment	50
step_for_evaluation	Varying per experiment.

Table 6.7: Parameters for the Industrial Benchmark experiment.

Chapter 7

Conclusion and Outlook

7.1 Conclusion

7.2 Future Work

7.3 Final Remarks

Bibliography

[1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, March 1994.

[2] Amina Adadi and Mohammed Berrada. Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160, 2018.

[3] E. Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning series. MIT Press, fourth edition, 2020.

[4] Dan Amir and Ofra Amir. Highlights: Summarizing agent behavior to people. In *Adaptive Agents and Multi-Agent Systems*, 2018.

[5] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai, 2019.

[6] Akanksha Atrey, Kaleigh Clary, and David D. Jensen. Exploratory not explanatory: Counterfactual analysis of saliency maps for deep RL. *CoRR*, abs/1912.05743, 2019.

[7] Lloyd J. Ball. Shap documentation, 2023. Accessed: 2023-08-25.

[8] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957.

[9] Davide Castelvecchi. Can we open the black box of ai? *Nature*, 538:20–23, 10 2016.

[10] R. Dennis Cook. Detection of influential observation in linear regression. *Technometrics*, 19(1):15–18, 1977.

[11] Giang Dao, Indrajeet Mishra, and Minwoo Lee. Deep reinforcement learning monitor for snapshot recording. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 591–598, 2018.

[12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[13] Matplotlib Developers. Matplotlib: A 2d plotting library, 2023. Accessed: 2023-08-15.

[14] NumPy Developers. Numpy: The fundamental package for scientific computing with python, 2023. Accessed: 2023-08-15.

[15] OpenAI Developers. Gym library: Openai gym documentation, 2023. Accessed: 2023-08-18.

[16] PyTorch Developers. Pytorch: An open source machine learning framework, 2023. Accessed: 2023-08-15.

[17] Seaborn Developers. Seaborn: Statistical data visualization, 2023. Accessed: 2023-08-15.

[18] Mengnan Du, Ninghao Liu, and Xia Hu. Techniques for interpretable machine learning. *CoRR*, abs/1808.00033, 2018.

[19] European Commission, Directorate-General for Communications Networks, Content and Technology. Proposal for a regulation of the european parliament and of the council laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts, 2021.

[20] Python Software Foundation. Logging — logging facility for python, 2023. Accessed: 2023-08-15.

[21] Python Software Foundation. pickle — python object serialization, 2023. Accessed: 2023-08-15.

[22] Python Software Foundation. Python programming language – official website, 2023. Accessed: 2023-08-15.

[23] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[24] Omer Gottesman, Joseph Futoma, Yao Liu, Sonali Parbhoo, Leo Anthony Celi, Emma Brunskill, and Finale Doshi-Velez. Interpretable off-policy evaluation in reinforcement learning by highlighting influential transitions. *CoRR*, abs/2002.03478, 2020.

[25] Sam Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and understanding atari agents. *CoRR*, abs/1711.00138, 2017.

[26] Daniel Hein, Stefan Depeweg, Michel Tokic, Steffen Udluft, Alexander Hentschel, Thomas A. Runkler, and Volkmar Sterzing. A benchmark environment motivated by industrial control problems. *CoRR*, abs/1709.09480, 2017.

[27] Daniel Hein, Stefan Depeweg, Michel Tokic, Steffen Udluft, Alexander Hentschel, Thomas A. Runkler, and Volkmar Sterzing. A benchmark environment motivated by industrial control problems. *CoRR*, abs/1709.09480, 2017.

[28] Daniel Hein, Steffen Udluft, and Thomas A. Runkler. Interpretable policies for reinforcement learning by genetic programming. *CoRR*, abs/1712.04170, 2017.

[29] Daniel Hein, Steffen Udluft, Michel Tokic, Alexander Hentschel, Thomas A. Runkler, and Volkmar Sterzing. Batch reinforcement learning on the industrial benchmark: First experiences. *CoRR*, abs/1705.07262, 2017.

[30] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[31] Cô me Huré, Huyên Pham, Achref Bachouch, and Nicolas Langrené. Deep neural networks algorithms for stochastic control problems on finite horizon: Convergence analysis. *SIAM Journal on Numerical Analysis*, 59(1):525–557, jan 2021.

[32] Hidenori Itaya, Tsubasa Hirakawa, Takayoshi Yamashita, Hironobu Fujiyoshi, and Komei Sugiura. Visual explanation using attention mechanism in actor-critic-based deep reinforcement learning. *CoRR*, abs/2103.04067, 2021.

[33] Gandhi Jafta, Alta de Waal, Iena Derks, and Emma Ruttkamp-Bloem. Evaluation of xai as an enabler for fairness, accountability and transparency. 2022.

[34] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. Examples are not enough, learn to criticize! criticism for interpretability. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[35] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951.

[36] Mikel Landajuela, Brenden K Petersen, Sookyung Kim, Claudio P Santiago, Ruben Glatt, Nathan Mundhenk, Jacob F Pettit, and Daniel Faissol. Discovering symbolic policies with deep reinforcement learning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 5979–5989. PMLR, 18–24 Jul 2021.

[37] Scikit learn Developers. Scikit-learn: Machine learning in Python, 2023. Accessed: 2023-08-15.

[38] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[39] Prashan Madumal, Tim Miller, Liz Sonenberg, and Frank Vetere. Explainable reinforcement learning through a causal lens. *CoRR*, abs/1905.10958, 2019.

[40] Stephanie Milani, Nicholay Topin, Manuela Veloso, and Fei Fang. A survey of explainable reinforcement learning, 2022.

[41] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, second edition, 2018.

[42] Christoph Molnar. *Interpretable Machine Learning*. 2 edition, 2022.

[43] Erika Puiutta and Eric MSP Veith. Explainable reinforcement learning: A survey, 2020.

[44] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel B. Brazdil, Alípio Mário Jorge, and Luís Torgo, editors, *Machine Learning: ECML 2005*, pages 317–328, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[45] Lloyd S Shapley. A value for n-person games. In Harold W. Kuhn and Albert W. Tucker, editors, *Contributions to the Theory of Games II*, pages 307–317. Princeton University Press, Princeton, 1953.

[46] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[47] Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of self-interpretable agents. *CoRR*, abs/2003.08165, 2020.

[48] Erico Tjoa and Cuntai Guan. A survey on explainable artificial intelligence (XAI): Toward medical XAI. *IEEE Transactions on Neural Networks and Learning Systems*, 32(11):4793–4813, nov 2021.

[49] Nicholay Topin and Manuela Veloso. Generation of policy-level explanations for reinforcement learning. *CoRR*, abs/1905.12044, 2019.

[50] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[51] Sandra Wachter, Brent D. Mittelstadt, and Chris Russell. Counterfactual explanations without opening the black box: Automated decisions and the GDPR. *CoRR*, abs/1711.00399, 2017.

[52] Vincent Zha and Ivey Chiu. Hyperspace neighbor penetration approach to dynamic programming for model-based reinforcement learning problems with slowly changing variables in a continuous state space. In *2021 International Conference on Computing, Computational Modelling and Applications (ICCMA)*, pages 57–64, 2021.