Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Towards a Framework for Algorithm Learning: Cross-Domain Induction and the Principled Limits of Algorithmic Abstraction

Philipp Kunz

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Dr. Ilche Georgievski
Commenced:	December 4, 2024
Completed:	June 4, 2025

Abstract

Computation graphs over homogeneous algebras allow for the learning of algorithms from data that sequentially reflect individual computation steps— so-called memory traces — to be framed as a classical machine learning problem, specifically as a classification task. However, previous work revealed two central limitations: (1) Changes in the input data distribution lead to a significant drop in model accuracy, limiting transferability. (2) The concept of abstraction in the context of computation graphs over homogeneous algebras remained vague and lacked formal definition. We evaluate the effectiveness of domain generalization methods in handling shifts in input distributions. Based on a systematic literature review, we select two established approaches and compare their performance with our domain-specific method, DIBE. We conduct a case study using classical comparison-based sorting algorithms and empirically demonstrate that some of these methods improve model robustness to domain shifts. Furthermore, we extend the theory of computation graphs over homogeneous algebras by introducing the notion of a quotient algebra. Quotient algebras are used to formally define two forms of abstraction: structural abstraction, as the compression of linear computation sequences independent of the underlying carrier set, and modular abstraction, as black-box operations. Finally, we prove that modular abstraction constitutes an undecidable decision problem, indicating that this type of abstraction is partially inaccessible to algorithmic approaches.

Kurzfassung

Berechnungsgraphen über homogenen Algebren ermöglichen es, algorithmisches Verhalten aus Daten zu erlernen, indem einzelne Rechenschritte als Klassifikationsaufgabe modelliert werden. Frühere Arbeiten zeigten jedoch zwei zentrale Probleme: (1) Modelle verlieren deutlich an Genauigkeit bei Veränderungen in der Eingabeverteilung, was ihre Übertragbarkeit einschränkt. (2) Der Begriff der Abstraktion blieb theoretisch unklar und undefiniert. In dieser Arbeit schlagen wir zunächst ein formales Modell zur Generierung geeigneter Trainingsdaten vor und grenzen mögliche Verteilungsverschiebungen auf Covariate Shifts und Label Shifts ein. Theoretisch zeigen wir, dass beide nicht gleichzeitig auftreten können. Für Insertion Sort belegen wir experimentell das Auftreten von Covariate Shifts und schließen damit gleichzeitige Label Shifts aus. Auf Basis eines systematischen Literaturreviews wählen wir zwei etablierte Domain-Generalization-Methoden aus und vergleichen diese experimentell mit unserem domänenspezifischen Ansatz DIBE. Eine Fallstudie mit klassischen Vergleichs-basierten Sortieralgorithmen zeigt, dass einige dieser Methoden die Robustheit gegenüber Verteilungsshifts verbessern. Darüber hinaus erweitern wir die Theorie der Berechnungsgraphen über homogenen Algebren durch die Einführung des Konzepts der Quotientenalgebra. Quotientenalgebren dienen dazu, zwei Formen von Abstraktion formal zu definieren: strukturelle Abstraktion als Verdichtung linearer Rechensequenzen unabhängig von der Trägermenge und modulare Abstraktion als Black-Box-Operationen. Abschließend zeigen wir, dass die modulare Abstraktion ein unentscheidbares Entscheidungsproblem darstellt, was darauf hinweist, dass diese Form der Abstraktion algorithmisch nur eingeschränkt zugänglich ist.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context and Motivation

The public perception of the disruptive potential of artificial intelligence has been shaped primarily by three key events: In 1997, the chess-engine Deep Blue[CHH02], defeated then-world champion Garry Kasparov in a highly complex game with an enormous state space — a symbolic moment that for the first time demonstrated the superiority of machines in intellectual domains[Web97]. When AlphaGo[SHM+16], an agent based on reinforcement learning, defeated Go world champion Lee Sedol in 2016, it was widely seen as the moment when a computer achieved a feat previously regarded as the epitome of human intuition and strategic depth[SM16]. With the widespread adoption of ChatGPT[Ope24] and other large language models (LLMs)[ZZL+25] starting in 2022[Met22], a kind of synthesis of previously separate paradigms took place: while LLMs, like AlphaGo, rely on data-driven learning methods, they increasingly demonstrate the ability to emulate symbolic logic without relying on explicitly encoded rules.

This ability for structured symbolic manipulation forms a conceptual bridge back to rule-based systems like Deep Blue — but based entirely on statistical principles. However, this ability remains purely statistical in nature: at this point in time, it must be questioned that these models possess an abstractable understanding of the world or perform deductive reasoning in the sense of deterministic formal symbol systems[MAS+24].

In our peer-reviewed conference paper[KGA24] 'Towards a Framework for Learning of Algorithms: The Case of Learned Comparison Sorts', presented at the 33rd International Joint Conference on Artificial Intelligence, Main Track, we introduced a framework that enables the learning of algorithmic structures — that is, rule-based systems such as Deep Blue — from memory traces. In the context of algorithm learning and logic deduction, this work addresses the absence of an explicit world model in data-driven architectures along two fundamental dimensions: (1) abstraction, as the compression of structurally equivalent computations or as the ability to derive fixed logical or symbolic rules from empirical data and (2) induction, defined as the ability to generalize from observed state transitions using statistical methods.

In this framework, operational semantics are formalized using the concept of homogeneous algebras, while control flow is represented by an augmented control-flow graph called the computation graph, which can be traversed via an algebraic structure referred to as the transition algebra. The integration with classical machine learning is achieved by training a model to predict, for a given computation state, the correct next operation within predefined algebraic structures. This positions the framework as a contribution to the field of neuro-symbolic AI, combining formal structure with statistical generalization.

1.2 Research Questions

A key aspect of our previous research is the inference of algorithmic rule systems at a fixed level of abstraction — one that is primarily determined by the training data. In the course of experimental evaluation, which was based on classical comparison-based sorting algorithms, the following key challenges and limitations were identified: (1) Since different parts of an algorithm's computation graph are traversed with varying frequencies, there is a strong tendency towards imbalanced training datasets, which can lead to reduced accuracy in handling rare cases. (2) As the input distribution shifts, a significant drop in accuracy can be observed, raising concerns about the transferability of the learned algorithm to inputs sampled from distributions that differ from the training distribution. (3) Up to now, the achieved results are grounded on the following limiting assumptions: (3.1) training data allows for the differentiation between payload and state, (3.2) the reconstruction of an algorithm's control flow graph is done under the assumption of an order on the training data points, (3.3) the set of elementary operations is known in advance. In particular, points (2) and (3) directly limit the framework's capacity for abstraction and induction. Accordingly, we address these limitations through the following research questions:

1. Induction: How can we mitigate the adverse effects of distribution shifts and imbalanced training data on model accuracy?
2. Abstraction: How can abstraction in the context of rule-based systems be conceptually and formally defined in order to establish a theoretical framework for algorithmic abstraction?
3. Abstraction: How can we infer the homogeneous algebra and transition algebra directly from training data under relaxed assumptions? Specifically when
   a) state and payload are not distinguishable in training data,
   b) we only know a set of primitives that define the functions of the homogeneous algebra.

1.3 Outlook

The remainder of this thesis is structured as follows: Chapter 2 recapitulates the central definitions and theorems of the underlying framework and introduces methods for handling shifts in test data distributions. Chapter 3 situates the work within the current state of research and establishes relevant theoretical connections. Chapter 4 provides the formal foundation for analyzing distributional shifts in training and test data and presents a qualitative investigation of these changes. Building on this, Chapter 6 reviews existing approaches to handling distribution shifts with regard to their suitability for the specific problem domain — learning algorithms. Suitable methods are selected based on predefined criteria and serve as baselines for our own approach, introduced in Chapter 7: Domain-Invariant Binary Embeddings (DIBE). Chapter 5 quantitatively analyzes the distributional shifts described in Chapter 4 using established metrics. In Chapter 8, we evaluate both existing methods and our own approach — DIBE — in terms of their robustness to the distributional shifts previously analyzed both qualitatively and quantitatively. The conceptual core of the thesis is found in Chapter 9, in which we develop a general theory of abstraction for algorithmic

processes, thereby establishing the theoretical basis for learning algorithmic abstractions. Finally, Chapter 10 provides a critical reflection on the results and outline opens questions for future research.

As the applied methodologies vary depending on the research question — from experimental evaluations to formal proofs — we refrain from including a dedicated methodology chapter. Instead, we present and justify our methodological approach within the context of each respective research question and its corresponding chapters.

2 Background

This chapter provides the theoretical background necessary for understanding the core concepts and frameworks that underpin the research presented in this thesis. It explores key theories, models, and previous studies relevant to the topic, offering a foundation for the analysis and discussion in subsequent chapters. Additionally, existing theories and models are partially extended.

2.1 Model of Computation

For the application of classical machine learning methods, such as supervised learning, to the task of learning algorithms from memory traces, a suitable theoretical framework is essential. In our previous work[KGA24], we have introduced computation graphs over homogeneous algebras as a foundation for this purpose. In this section, we refrain from citing our work repeatedly. Unless stated otherwise, all definitions, theorems, and claims presented here are based on the results of our prior work [KGA24]. In the following, we provide an overview of the key components and features of this computational model.

2.1.1 Homogeneous Algebras

Computation graphs employ homogeneous algebras as abstractions of classical data structures, providing the operations necessary to express algorithms at a particular level of abstraction. The functions defined within a homogeneous algebra constitute the formal vocabulary for algorithm specification and, together with the underlying carrier set, determine the expressive scope of that specific abstraction level.

Definition 2.1.1
[KGA24] A homogeneous algebra *is a tuple $(M, (f_i)_{i\in I})$, where M is a set and $(f_i)_{i\in I}$ an indexed family of functions of finite arity on M with I being a set of indexes.*

A configuration is an element of the carrier set M and represents the processing state of the data at a specific point during the computation. As the computation progresses, one of the functions defined in the homogeneous algebra is applied, mapping a configuration $m \in M$ to a successor $m' \in M$.

Definition 2.1.2
[KGA24] A configuration is an element $m \in M$. By $M_f := \bigcup_{i\in I} \{f_i\}$ we denote the set of operations on M_f.

2.1.2 Properties, Property Spaces and Predicates

A Turing-machine operates by reading the symbol at the current position of its tape using a read-write head, and mapping this local observation to both a new symbol (to be written) and a successor state[Tur36]. Properties aim to transfer this principle of local evaluation of data structures to the setting of homogeneous algebras. From the perspective of statistical learning, such locally evaluable properties correspond to features.

Definition 2.1.3
[KGA24] A property π is a mapping $\pi : M \to W$, where $(M, (f_i)_{i \in I})$ is a homogeneous algebra and W is a non-empty set.

A property π maps a configuration $m \in M$ to an element of W, effectively extracting specific information from m. Given multiple relevant properties $\pi_1, \ldots, \pi_n$, we can impose an order on them and apply each to m to produce an ordered tuple of measurements: $(\pi_1(m), \ldots, \pi_n(m))$.

Definition 2.1.4
[KGA24] Let $(M, (f_i)_{i \in I})$ be an an algebra and $(\pi_1, ..., \pi_n)$ a tuple of properties. The mapping $\Pi_{(\pi_1,...,\pi_n)}$ is defined as follows:

$$\Pi_{(\pi_1,...,\pi_n)} : M \to W_1 \times ... \times W_n, m \mapsto (\pi_1(m), ..., \pi_n(m)) \tag{2.1}$$

If we apply this mapping to all $m \in M$, we obtain a property space, which can be informally understood as the set of all observations across all configurations.

Definition 2.1.5
[KGA24] A property space $P^M_{(\pi_1,...,\pi_n)}$ is the image of M under $\Pi_{(\pi_1,...,\pi_n)}$, that is, $P^M_{(\pi_1,...,\pi_n)} := \Pi_{(\pi_1,...,\pi_n)}(M)$.

A central aspect of the concept of algorithms is the conditional execution of instructions based on the current intermediate result. Crucial to this is the ability to formulate such conditions in the first place. In our earlier work[KGA24], we therefore introduced the concept of predicates.

Definition 2.1.6
[KGA24] A predicate is function $q : P^M_{(\pi_1,...,\pi_n)} \to \{0, 1\}$.

2.1.3 Computation Graphs over Homogeneous Algebras

With all necessary building blocks in place, we now proceed to define computation graphs. In this framework, each node in the graph is labeled with a function from the corresponding homogeneous algebra. A computation is then represented as a traversal of the graph, during which the function associated with each visited node is systematically applied. The sequence of function applications is determined by the directed edges, which define the permissible paths through the graph.

To ensure unambiguous traversal paths through the graph, each edge is associated with a predicate. These predicates are crucial in guiding the computation: for every non-terminal node, exactly one outgoing edge has a predicate that evaluates to true at any given time. This mechanism enforces a deterministic progression through the graph, enabling precise and well-defined computations.

Definition 2.1.7
[KGA24] Let $(M, (f_i)_{i\in I})$ be an algebra, $(\pi_1, ..., \pi_n)$ a tuple of properties, and Q_M a non-empty set of predicates, where f_i is of arity one $\forall i \in I$. Then a computation graph *G is a 7-tuple $G = (V, E, M, \phi, \gamma, s, t)$, where*

1. *V is a finite set of nodes with $s \in V$ being the initial node and $t \in V \setminus \{s\}$ being the terminal node,*
2. *$E \subset V^2$ is a finite set of directed edges such that $\forall v \in V : (v, s) \notin V$ and $(t, v) \notin V$, and $\forall v_1 \in V\setminus\{t\}\exists v_2 \in V : (v_1, v_2) \in E$,*
3. *$\phi : V \to M_f$ assigns each node a function such that $\phi(t) = identity$,*
4. *$\gamma : E \to Q_M$ assigns each edge a predicate such that:*
   - $\forall p \in P^M_{(\pi_1,...,\pi_n)} \forall v_1 \in V\setminus\{t\}\exists v_2 \in V :$
     $$(v_1, v_2) \in E \land \gamma(v_1, v_2)(p) = 1$$
   - $\forall p \in P^M_{(\pi_1,...,\pi_n)} \forall (v_1, v_2), (v_1, v_3) \in E :$
     $(v_1, v_2) \neq (v_1, v_3) \land \gamma(v_1, v_2)(p) = 1$
     $$\implies \gamma(v_1, v_3)(p) = 0$$

By $identity$ we refer to the identity $M \to M$. Therefore, there must be an $i \in I$ for which $f_i = identity$ holds. Given a computation graph $G = (V, E, M, \phi, \gamma, s, t)$, we refer by G' to the directed graph $G' = (V, E)$.

A computation is defined as a systematic traversal of the underlying computation graph and consists of a sequence of states $(v, m) \in V \times M$. Each transition from one state (v_i, m_i) to the next (v_{i+1}, m_{i+1}) must conform to the structure of the computation graph G. The validity of these transitions is governed by a transition relation, which specifies the necessary and sufficient conditions under which a transition is considered valid.

Definition 2.1.8
[KGA24] Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph and

$$(v_i, m_i), (v_{i+1}, m_{i+1}) \in V \times M \tag{2.2}$$

be states. A transition relation $\vdash_G$ between (v_1, m_1) and (v_2, m_2), denoted as

$$(v_i, m_i) \vdash_G (v_{i+1}, m_{i+1}), \tag{2.3}$$

is valid *if and only if the following conditions hold true:*

- $(v_i, v_{i+1}) \in E$
- $\gamma(v_i, v_{i+1})(\Pi_{(\pi_1,...,\pi_n)}(m_1)) = 1$
- $m_2 = \phi(v_i)(m_1)$

A computation step from (v_i, m_i) to (v_{i+1}, m_{i+1}) is valid with respect to $\vdash_G$ if the following conditions are satisfied: (1) the nodes v_i and v_{i+1} are connected by an edge in G; (2) the predicate associated with the edge (v_i, v_{i+1}) evaluates to 1 (true) when applied to m_i; and (3) the configuration m_{i+1} results from applying the function associated with v_i in the homogeneous algebra to m_i.

The following theorems establish two properties of the transition relation $\vdash_G$, which fundamentally characterize the nature of computation in a computation graph: A non-terminal state (v_1, m_1) has exactly one successor with respect to $\vdash_G$, whereas a terminal state (t, m_1) has none, thereby consistently representing the end of a computation.

Theorem 2.1
[KGA24] Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph.

Then it holds:

(2.4) $\forall (v_1, m_1) \in (V \backslash \{t\} \times M)\, \exists! (v_2, m_2) \in V \times M : (v_1, m_1) \vdash_G (v_2, m_2)$

Theorem 2.2
[KGA24] Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph.

Then it holds:

(2.5) $\forall (v_1, m_1) \in V \times M : (v_2, m_2) \in \{t\} \times M \implies (v_2, m_2) \nvdash_G (v_1, m_1).$

A computation is defined as a sequence of systematic state transitions $(v_i, m_i) \vdash_G (v_{i+1}, m_{i+1})$ governed by the transition relation $\vdash_G$. This notion encompasses both terminating computations, where a final state of the form (v_1, t) is eventually reached, and non-terminating computations that proceed indefinitely. Definition 2.1.9 and Corollary 2.3 provide a formal foundation for this concept[KGA24].

Definition 2.1.9
[KGA24] Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph and $m_1 \in M$ an initial configuration. A computation *is a sequence*

(2.6) $(s_1, m_1), (s_2, m_2), ... \in V \times M$

such that the following holds:

(2.7) $(s_1, m_1) \vdash_G (s_2, m_2) \vdash_G \ldots$

Corollary 2.3
[KGA24] Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph, $m_1 \in M$ an initial configuration, and $\vdash_G^+$ a transitive hull of $\vdash_G$.

Then it holds:

(2.8) $s_1, m_1) \vdash_G (s_2, m_2) \vdash_G ...is finite \iff \exists m_2 \in M : (s, m_1) \vdash_G^+ (t, m_2)$

The effectiveness of computational graphs as a unifying framework for learning algorithms from memory traces is fundamentally underpinned by their Turing completeness. Under the assumption of the Church-Turing thesis[Tur36], every intuitively computable function can be represented by an equivalent computational graph defined over a homogeneous algebra.

Theorem 2.4
[KGA24] For a given deterministic Turing machine

(2.9) $T := (Z, \Gamma, \Sigma, \delta, z_0, b, t_T)$

there exist a homogeneous algebra

(2.10) $(M, (f_i)_{i \in I})$

such that a corresponding equivalent computation graph

(2.11) $G = (V, E, M, \phi, \gamma, s, t_G)$

can be constructed, where for any initial tape configuration $\alpha z_0 \beta$, there exist an initial configuration $m_1 \in M$ such that

(2.12) $\alpha z_0 \beta \vdash_T^+ \alpha' t_T \beta' \iff (s, m_1) \vdash_G^+ (t_G, m_2)$

The presented model thus establishes the formal foundations for a purely algebraic consideration of computations, which we will introduce in the next section.

2.1.4 Transition Algebras and Actuators

Representing computations in a fully algebraic manner allows them to be directly framed as a classification problem. This approach demands careful modeling of state transitions within the computation graph, a requirement that has driven the development of transition algebras in prior work.

Definition 2.1.10
[KGA24] Let $G = (V, E)$ be a directed graph. Then $suc : V \to V$ is defined as follows

(2.13) $suc(v_i) := |\{v_j \in V | (v_i, v_j) \in E\}|$

and V_{deg} is defined as:

(2.14) $V_{deg} = \{v_2 \in V | \exists v_1 \in V : (v_1, v_2) \in E \wedge suc(v_1) > 1\}$

Definition 2.1.10 establishes the theoretical foundation for transition algebras, as formally defined in Definition 2.1.11. For a given node v in a computation graph G, the function $suc(v)$ returns the number of successor nodes of v in G. The set V_{deg} comprises all nodes in G that have at least one predecessor with multiple successors.

Whereas homogeneous algebra operates on the elements of its carrier set, a transition algebra is designed to enable traversal within a computation graph. In this context, the carrier set corresponds to the set of nodes V, and the associated functions are induced by the topology of the underlying directed graph.

Definition 2.1.11
[KGA24] Let $G = (V, E)$ be a directed graph. A transition algebra T_G induced by G *is a tuple*

(2.15) $T_G = (V, (transition_i)_{i \in I})$,

where for all:

(2.16) $i \in I := V_{deg} \cup \{\epsilon\}$

$$transition_\epsilon(v_i) = \begin{cases} v_j & if\ suc(v_i) = 1 \wedge (v_i, v_j) \in E \\ v_i & else \end{cases}$$

$$transition_{v_j}(v_i) = \begin{cases} v_j & if\ (v_i, v_j) \in E \\ v_i & else \end{cases}$$

Given a node v with exactly one successor in G, the function $transition_\epsilon$ maps v to its successor; otherwise, it acts as the identity. Cases where a node v_j has multiple predecessors — that is, $v_j \in V_{deg}$ — are handled by $transition_{v_j}$, which enables the computation to transition to v_j from any node v_i such that $(v_i, v_j) \in E$. Based on this definition, we say that G induces the transition algebra T_G. We denote by $T^G_{\text{transition}}$ the set of functions over T_G, defined as:

(2.17) $T^G_{\text{transition}} := \bigcup_{i \in I} \{\text{transition}_i\}$

Definition 2.1.12
[KGA24] $connected^n_G(v_i, v_j) :\Longleftrightarrow$

$$\exists (v_i, ..., v_j) \in V^n \forall 1 \leq i \ \leq n-1 : (v_i, v_{i+1}) \in E.$$

For transition algebra to be practically applicable in traversing a computation graph, it is essential that every function composition corresponds to a valid path in the graph, and vice versa—that every path can be expressed as a composition of functions. The following two theorems formally guarantee this correspondence.

Theorem 2.5
[KGA24] Let $T_G = (V, (transition_i)_{i \in I})$ be a transition algebra induced by a directed graph G.

Then, for t_1, ..., $t_{n-1} \in T^G_{transition}$, it holds that:

$$\forall n \geq 2 : \forall (v_1, ..., v_n) \in V^n : connected^n_G(v_1, v_n) \implies v_n = t_{n-1} \circ ... \circ t_1(v_1)$$

Theorem 2.6
[KGA24] Let $G = (V, E)$ be a directed graph and $T_G = (V, (transition_i)_{i \in I})$ an induced transition algebra. Then, for all $v_i, v_j \in V$, $v_i \neq v_j$ it holds that:

$$\forall n \geq 0 \forall (v_1, ..., v_{n+2}) \in \{v_i\} \times V^n \times \{v_j\}$$

$$\forall k \geq 1 : (t_1, ..., t_k) \in T_G^k \implies v_j \neq t_k \circ ... \circ t_1(v_i).$$

In the context of viewing computations algebraically as compositions of functions, a central question is how to represent an individual computation step as a function application. To address this, we previously introduced the notion of an actuator. An actuator integrates a function from the homogeneous algebra with one from the transition algebra, enabling it to map a computational state $(v_1, m_1) \in V \times M$ to a successor state $(v_2, m_2) \in V \times M$.

Definition 2.1.13
[KGA24]. Let $f_t \in T^G_{transition}$ and $f_m \in M_f$. An actuator *is a function*

(2.18) $a_{f_t f_m} : V \times M \to V \times M$

such that

(2.19) $(v, m) \mapsto (f_t(v), f_m(m))$.

For a given $T^G_{transition}$ and M_f, the set of all actuators A_G is defined as follows:

(2.20) $A_G := \{a_{f_t f_m} | (f_t, f_m) \in T_G \times M_f\}$

In summary, we have demonstrated that arbitrary computations originating from an initial state $(v, m) \in V \times M$ can be represented as compositions of actuators.

In line with our objective of representing computations as concatenations of actuators, it remains to be clarified how a state sequence $(v_1, m_1), \ldots, (v_n, m_n)$ can be mapped to a corresponding actuator sequence $a_1, \ldots, a_{n-1}$ such that $(v_n, m_n) = a_{n-1} \circ \cdots \circ a_1(v_1, m_1)$.

With this in mind, we have introduced the mapping γ_G, which assigns the correct actuator to a state (v, s).

Definition 2.1.14
[KGA24] An actuator-state mapping *γ_G is a function*

(2.21) $\gamma_G : V \backslash \{t\} \times M \to M_f \times T_G$

such that

(2.22) $(v, m) \mapsto (f_t, \phi(v))$,

where

(2.23) $f_t \in A := \{t \in T_G | \phi(v, t(v))(m) = 1\} \wedge |A| \leq 1$

2.2 Machine Learning

Up to this point, we have used the term learning informally, leaving its meaning vague in the context of the problem. In this section, we introduce the mathematical and formal foundations to enable a precise definition of the term. In line with our previous work, we adopt a statistical perspective.

2.2.1 Empirical Risk Minimization (ERM)

We deliberately refrain from a broad, informal discussion of the concept of machine learning and instead focus on the principle that is fundamental to our problem context: empirical risk minimization (ERM)[HTF09].

Consider random variables X and Y with a joint distribution $P(X,Y)$. According to Hastie et al.[HTF09], the learning problem can thus be described in its basic form for a set of candidate functions

(2.24) $F := \{f | f : image(X) \rightarrow image(Y)\}$

and a loss function L [HTF09] as a minimization problem of the expected empirical risk:

(2.25) $argmin_{f \in F} E(L(f(x), Y))$

In this context, the loss function L measures the deviation of a candidate function f from the correct response in each case. A machine learning algorithm systematically searches the space F for a function f that minimizes equation 2.25.

2.2.2 Neural Networks

We assume that the reader is familiar with feedforward neural networks. However, to ensure a clear mathematical formulation of our approach, we formally introduce them again in the following.

Following Hastie et al.[HTF09], a feed-forward neural network N computes a function $N : \mathbb{R}^m \rightarrow \mathbb{R}^n$ by chaining functions (layers) $l_i : \mathbb{R}^{m_i} \rightarrow \mathbb{R}^{n_i}$, such that:

(2.26) $N = l_h \circ ... \circ l_1$

A layer i correponds to an affine map $A_i x + b_i$, where $A \in \mathbb{R}^{n_i \times m_i}$, $b_i \in \mathbb{R}^{n_i}$ chained with a non-linear activation function $\sigma_i : \mathbb{R}^{n_i} \rightarrow \mathbb{R}^{n_i}$:

(2.27) $l_i(x) = \sigma_i(A_i x + b_i)$

2.2.3 Algorithm Learning

So far, it remains unclear how the problem of learning an algorithm from data can be integrated into the empirical risk minimization framework and thus into machine learning methods. To address this, we build on our previous definition of the learning problem [KGA24].

Let $\Omega := V \times M$, and let $\mathcal{P}_\Omega$ be a probability measure on Ω turning (Ω, P_Ω) into discrete probability space. Consider random variables

(2.28) $X : \Omega \rightarrow \Omega, \quad (v, m) \mapsto (v, m)$

(2.29) $Y : \Omega \rightarrow A_G, \quad (v, m) \mapsto \gamma_G(v, m)$

Algorithm 1

Input:
$(v, m) \in V \times M$ - a configuration.
$G := (V, E)$ - a state graph.
ϕ - a function $\phi : V \to M_f$

$f \leftarrow \phi(v)$
$t \leftarrow None$
if $suc_G(v) \leq 1$ **then**
  $t \leftarrow suc$
else
  $t \leftarrow classify(v, m)$
return $(t, f) \in A_G$

With the hypothesis space $F := \{f \mid f : V \times M \to A_G\}$ and the zero-one loss function L[HTF09], learning an algorithm corresponds to solving the following optimization problem:

(2.30) $argmin_{f \in F} \mathbb{E}_{\mathbb{P}_{X,Y}} [L(f(X), Y)]$

For this work, the decomposition of f into a dependent part, which depends on the marginal distribution $P(Y)$, and an independent part is of great importance. Consider an actuator $a_{f_t, f_m} \in A_G$, which is component-wise defined by $f_t \in T_G$ and $f_m \in M_f$.

Due to the definition of the computation graph, $f_m \in M_f$ depends only on the state set V, while for f_t with a given state $Y = (v, m)$, two cases must be distinguished:

1. $suc(v) \leq 1 \implies f_t = suc$
2. $suc(v) > 1$. Let $v_1, ..., v_n$ be the successors of v. In this case, f_t depends on the predicates $\gamma(v, v_1), ..., \gamma(v, v_n)$, which means it depends on $m \in M$ and, consequently, on the input distribution.

In summary, with knowledge of the function ϕ, an appropriate classifier can be constructed, as outlined in Algorithm 1.

2.3 Distribution Shifts

Empirical Risk Minimization (ERM) is based on the assumption that the distribution of the training data matches that of the test data[HTF09] . However, in real-world applications of our framework[KGA24], this assumption is often not met, leading to degenerating test performance in terms of accuracy[ZLQ+23]. It is precisely this discrepancy that motivates our research question. In this section, we therefore present a taxonomy of qualitative differences between training and test distributions that is commonly used in the literature. Foundational for this analysis is a factorization of a joint distribution:

(2.31) $P(X, Y) = P(X)P(Y|X) = P(Y)P(X|Y)$

Based on this, we can distinguish covariate, label and concept shifts that can be defined as followingly outlined.

Definition 2.3.1
[LWS18] Covariate Shift. A distribution $P_2(X,Y)$ arises from a covariate shift $P_1(X,Y)$, if the following condition holds:

(2.32) $P_2(X,Y) = P_2(X)P_1(Y|X)$

Definition 2.3.2
[LWS18] Label Shift. A distribution $P_2(X,Y)$ arises from a label shift $P_1(X,Y)$, if the following condition holds:

(2.33) $P_2(X,Y) = P_2(Y)P_1(X|Y)$

Definition 2.3.3
[Huy22] Concept Shift. A distribution $P_2(X,Y)$ arises from a concept shift $P_1(X,Y)$, if the following condition holds:

(2.34) $P_2(X,Y) = P_1(X)P_2(Y|X)$

With domain generalization as discussed in Section 2.4, there is a well-established framework to address such shifts. However, each type of shift requires a different approach, which leads to the question, whether these shifts can happen simultaneously. As it turns out, concept shifts can be excluded using theoretical arguments. We therefore limit, in the following, the discussion to covariate and label shifts. To our knowledge, this result is not commonly stated in this form in the literature, which motivates the following formal proof.

Although discrete probability distributions dominate in practical applications, we use the integral notation in the following, as it allows for a more general, measure-theoretic perspective.

Theorem 2.7
Let $P_1(X,Y) \neq P_2(X,Y)$ be distinct distributions, where

(2.35) $P_1(X,Y) = P_1(Y|X)P_1(X) = P_1(X|Y)P_1(Y)$

(2.36) $P_2(X,Y) = P_2(Y|X)P_2(X) = P_2(X|Y)P_2(Y)$

and

(2.37) $P_1(Y|X) = P_2(Y|X)$

(2.38) $P_1(X|Y) = P_2(X|Y)$

Than it holds:

(2.39) $P_2(X)P_2(Y|X) = P_2(X)P_1(Y|X) \neq P_2(Y)P_1(X|Y) = P_2(Y)P_2(X|Y)$

Proof. $P_1(X,Y) \neq P_2(X,Y)$

$\iff P_1(Y|X)P_1(X) = P_1(X|Y)P_1(Y) \neq P_2(Y|X)P_2(X) = P_2(X|Y)P_2(Y)$

$\implies P_1(X) \neq P_2(X) \land P_1(Y) \neq P_2(Y)$

By assumption it holds that $P_1(Y|X) = P_2(Y|X)$

$\iff \frac{P_1(Y)P_1(X|Y)}{P_1(X)} = \frac{P_2(Y)P_2(X|Y)}{P_2(X)}$

$\iff \int_{-\infty}^{\infty} \frac{P_1(Y)}{P_1(X)}\, dy = \int_{-\infty}^{\infty} \frac{P_2(Y)}{P_2(X)}\, dy$

$\iff \frac{1}{P_1(X)} = \frac{1}{P_2(X)}$

$\implies P_1(X) = P_2(X)$, which contradicts $P_1(X) \neq P_2(X)$

Corollary 2.8
Covariate and label shifts never occur simultaneously.

Proof. Let $P_1(X,Y)$ and be $P_2(X,Y)$ be distributions such that

1. $P_1(X,Y) \neq P_2(X,Y)$
2. $P_2(X,Y) = P_2(X)P_1(Y|X) = P_2(Y)P_1(X|Y)$.
   (simultaneous label and covariate shift))

Under these assumptions theorem 2.7 states $P_2(X)P_1(Y|X) \neq P_2(Y)P_1(X|Y)$, which contradicts the second property. Consequently, no such distribution P_2 exists.

In Chapter 4, we explore the implications of shifting input distributions on the relationship between training and test data. In this context, corollary 2.8 is pivotal, as it allows us to narrow the discussion specifically to covariate shifts.

Several approaches have been developed in the past to address the challenges associated with non-identically and independently distributed training data (NIC) as well as shifting training and test distributions. In the following, we provide an overview of key research areas and methods in this domain.

Transfer Learning: Machine learning techniques in this category focus on adapting models trained on a particular task to perform effectively on new or related tasks [YZDP20]. The connection to the broader problem of domain shifts is evident when we frame a shifted domain as either a new or related task.

Meta Learning: Thrun et al.[TP98] define meta-learning as the process by which a learning algorithm learns how to learn. A prominent representative of this approach is MAML

(Model-Agnostic Meta-Learning)[FAL17], which aims to integrate multiple tasks into the training of differentiable models in order to enable rapid adaptation to new tasks with only a few training steps. Based on this paradigm, Li et al.[LYSH17b] propose MLDG as a domain generalization method explicitly addressing distribution shifts.

Few-Shot- and Zero-Shot-Learning: Few-shot learning refers to the ability of a model to quickly adapt to new tasks using only a small number of labeled examples, often by leveraging prior knowledge from related tasks[WYKN20]. Zero-shot learning, in contrast, enables models to handle tasks involving classes or concepts not encountered during training, typically through auxiliary information such as semantic embeddings[WYKN20]. Both paradigms are closely linked to distributional shifts at test time, as they require generalizing beyond the training data to altered or entirely novel tasks.

Representation Learning: Representation learning aims to improve a model's generalizability to different (including unseen) tasks by extracting generic features that are both task-relevant and invariant[NS22]. Domain-invariant embeddings (DIBE) introduced in this work can be regarded as a specific instance of this approach.

Multitask-Learning: Multi-Task Learning (MTL) trains models on multiple related tasks simultaneously, enabling shared representations that improve generalization across tasks[ZY22]. In the context of domain generalization (see Chapter 2.4), MTL has been used to exploit parameter sharing as a form of regularization, enhancing robustness to distribution shifts[ZLQ+23].

Domain Adaptation: This field focuses on adjusting a model trained on a source domain to perform well on a target domain with a shifted distribution[WLL+23a]. Unlike domain generalization, which aims to create models that generalize across multiple domains without explicit target data, domain adaptation directly leverages data from the target domain during training to bridge the distribution gap[WLL+23a].

2.4 Domain Generalization

With domain generalization [BLS11], we introduce an additional paradigm for addressing distributional shifts. In contrast to domain adaptation, few-shot and zero-shot learning, as well as transfer learning, this class of methods enables the out-of-the-box use of a trained model without requiring labeled data from the target distribution [ZLQ+23]. The previously discussed techniques, meta-learning, representation learning, and multitask learning have already been successfully used in the past to implement the domain generalization (DG) paradigm[LYSH17b][MBS13][CDB+19][ZLQ+23]. Since we generally aim to avoid relying on labeled data from the target environment, since this would limit our framework's capability to generalize to entirely new domains, the DG framework is particularly well suited to our requirements, which is why we base our further investigations on it. In this section, we therefore provide a detailed examination of domain generalization, including its key components and properties.

2.4.1 Concepts and Definitions

The concept of domain generalization was introduced in 2011 by Blanchard et al. [BLS11] within the context of a formal framework. In the following, we present the central concepts and definitions based on their original work.

Blanchard et al. [BLS11] assume a shared feature space X and label space Y, and define a *domain* as a joint probability distribution $P(X, Y)$ over these spaces.

Definition 2.4.1 (after Blanchard et al. [BLS11])
Let X be the feature space and Y the label space. A domain is defined as a joint probability distribution $P_{X,Y}(X, Y)$.

Based on this, the authors formulate the following problem: Given training data of the form

$$\mathcal{D} = \{\mathcal{D}_k = \{(x^{(k)}, y^{(k)})\}\}_{k=1}^{K}, \tag{2.40}$$

where each $\mathcal{D}_k$ contains i.i.d. sampled data points from domain P_k, and it holds that

$$i \neq j \implies P_i \neq P_j, \tag{2.41}$$

the goal is to find a model $f : \text{img}(X) \rightarrow \text{img}(Y)$ that minimizes the expected error

$$\mathbb{E}_{P_{\text{test}}}[l(f(X), Y)] \tag{2.42}$$

for distributions P_{test} with $\forall i \in \{1, ..., K\} : P_{\text{test}} \neq P_i$. Zhou et al.[ZLQ+23] as well as Wang et al.[WLL+23b] further categorize domain generalization (DG) into single-source (when $K = 1$) and multi-source (when $K > 1$) settings, depending on the number of available training domains.

2.4.2 Meta-Learning for Domain Generalization (MLDG)

We now turn to algorithmic approaches that aim to address this problem formulation. In Chapter 5, we highlight two methods as particularly promising for addressing covariate shifts within the scope of this work: *Learning to Generalize: Meta-Learning for Domain Generalization* (MLDG) by Li et al.[LYSH17a] and *Gradient Matching for Domain Generalization* (Fish) by Shi et al.[SST+21].

Given their central importance to this thesis, we present both approaches in detail in the following sections. We begin with MLDG, a method proposed by Li et al. that adapts the *Model-Agnostic Meta-Learning* (MAML)[FAL17] framework for the purpose of domain generalization. This adaptation is designed for models that are differentiable and trainable via gradient descent [HTF09], resulting in the MLDG algorithm. In the following, we present MLDG as proposed by Li. et al. [LYSH17a].

In MLDG, we consider real-valued predictors $X \subset \mathbb{R}^m$, responses $Y \subset \mathbb{R}^n$, and model parameters $\Theta \in \mathbb{R}^l$. The algorithm operates over a hypothesis set F consisting of real-valued functions $f : \mathbb{R}^m \times \mathbb{R}^l \rightarrow \mathbb{R}^n$, parameterized by Θ.

MLDG assumes a multi-source training dataset D, as defined in Equation 7.8, consisting of $K > 1$ source domains. The training procedure is iterative, with each iteration (epoch) divided into a meta-train phase, a meta-test phase, and an update phase.

Meta-Train Phase: To simulate distribution shift during training, the dataset D is partitioned into a meta-train set D_{train} and a meta-test set D_{test}, such that $D = D_{\text{train}} \cup D_{\text{test}}$. Given a loss function $l : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, the meta-train loss is computed as:

$$F(\Theta, D_{\text{train}}) = \frac{1}{|D_{\text{train}}|} \sum_{D_{\text{dom}} \in D_{\text{train}}} \frac{1}{|D_{\text{dom}}|} \sum_{(x,y) \in D_{\text{dom}}} l(f(x, \Theta), y) \tag{2.43}$$

A temporary set of adapted parameters is then obtained via a gradient descent step with learning rate α:

$$\Theta' := \Theta - \alpha \nabla F(\Theta, D_{\text{train}}) \tag{2.44}$$

Meta-Test Phase: The updated parameters Θ' are evaluated on the meta-test data to compute a meta-test loss:

$$G(\Theta', D_{\text{test}}) = \frac{1}{|D_{\text{test}}|} \sum_{D_{\text{dom}} \in D_{\text{test}}} \frac{1}{|D_{\text{dom}}|} \sum_{(x,y) \in D_{\text{dom}}} l(f(x, \Theta'), y) \tag{2.45}$$

Update Phase: Finally, the original model parameters Θ are updated using a weighted combination of the meta-train and meta-test losses, controlled by a hyperparameter γ, and an outer learning rate β:

$$\Theta \leftarrow \Theta - \beta \nabla \left[F(\Theta, D_{\text{train}}) + \gamma G(\Theta', D_{\text{test}}) \right] \tag{2.46}$$

This training strategy encourages the model to learn parameters that not only adapt effectively to the training domains but also generalize well to unseen domains. By simulating domain shift through episodic training and structured partitioning of domains, MLDG provides a principled approach to improving robustness under covariate shift.

2.4.3 Gradient Matching for Domain Generalization (FISH)

Similar to MLDG, FISH is a multi-source domain generalization approach. Consequently, we assume a training data set as defined in equation 7.8, where $K > 1$. The following discussion reflects the results of Shi et al.[SST+21].

For a given domain $D_i \in D$, the expected loss is defined as

$$G_i := \mathbb{E}_{(x,y) \sim \mathcal{D}_i} [\ell(f(x, \Theta), y)]. \tag{2.47}$$

The goal of the FISH approach is to identify an update direction in parameter space along which the gradients $G_1, \ldots, G_K$ from the various training domains are well aligned, thereby avoiding conflicting updates. This ensures that a single optimization step not only reduces the overall loss,

but also yields a consistent improvement across all domains — that is, an improvement pointing in similar directions in parameter space. Mathematically, this corresponds to maximizing the pairwise inner products of the domain-specific gradients:

$$\sum_{\substack{i,j \in \{1,\dots,K\} \\ i \neq j}} G_i \cdot G_j. \tag{2.48}$$

Both objectives — minimizing the overall loss and maximizing gradient alignment — can be combined into the following optimization problem:

$$F(\Theta, \mathcal{D}) = \sum_{\mathcal{D}_i \in \mathcal{D}} \sum_{(x,y) \in \mathcal{D}_i} \ell(f(x, \Theta), y) - \gamma \frac{2}{|D|(|D|-1)} \sum_{\substack{i,j \in \{1,\dots,|D|\} \\ i \neq j}} G_i \cdot G_j, \tag{2.49}$$

where $\gamma > 0$ is a hyperparameter that balances the two objectives. The optimization of F is performed analogously to MLDG using gradient descent[HTF09]:

$$\Theta := \Theta - \alpha \nabla F(\Theta, \mathcal{D}). \tag{2.50}$$

It is important to note that this step involves second-order differentiation, as the gradients G_i themselves depend on Θ. To avoid the resulting computational overhead, the authors propose an approximation algorithm (see Algorithm 2) that relies exclusively on first-order gradients.

Algorithm 2 Fish

for iterations = $1, 2, \cdots$ **do**
  $\tilde{\theta} \leftarrow \theta$
  for $D_i \in$ permute($\{D_1, D_2, \cdots, D_S\}$) **do**
    Sample batch $d_i \sim D_i$
    $\tilde{g}_i = \mathbb{E}_{d_i}\left[\frac{\partial \ell((x,y);\tilde{\theta})}{\partial \tilde{\theta}}\right]$ // Grad wrt $\tilde{\theta}$
    Update $\tilde{\theta} \leftarrow \tilde{\theta} - \alpha \tilde{g}_i$
  Update $\theta \leftarrow \theta + \beta(\tilde{\theta} - \theta)$

2.5 Comparison Sorts

In previous research, we assessed our framework [KGA24] through the example of classical comparison sorts [CLRS09], which establish the correct ordering of integers by performing pairwise comparisons using a comparison relation such as $\leq$[CLRS09]. To ensure comparability, we ground our analysis in this work on the same class of algorithms.

2.5.1 Sorting Algebra

For our experiments, we partially leverage sorting algebra[KGA24], a homogeneous algebra particularly designed for sorting. Figure 2.1 provides a detailed overview of its components and functions.

	LoC	CC	DoT
Cluster 1			
Comparison Counting	88	4	8
Straight Insertion Sort	82	4	9
Straight Selection Sort	79	4	8
Shell Sort	133	5	11
Bubble Sort	63	4	7
Cluster 2			
Quick Sort	184	5	19
Binary Insertion Sort	199	6	19
Cluster 3			
Tree Selection	392	8	25
Tree Selection By Iverson	338	8	21
Top-Down-Merge-Sort	391	12	25
Bottom-Up-Merge-Sort	309	11	23
Cluster 4			
Heap Sort	328	15	21
Natural-Two-Way-Merge-Sort	371	17	22

Table 2.1: A taxonomy of sorting algorithms established with respect to algorithm-learning relevant metrics such as lines of code (LoC), cyclomatic complexity (CC), and the dimensionality of training data (DoT) [KGA24].

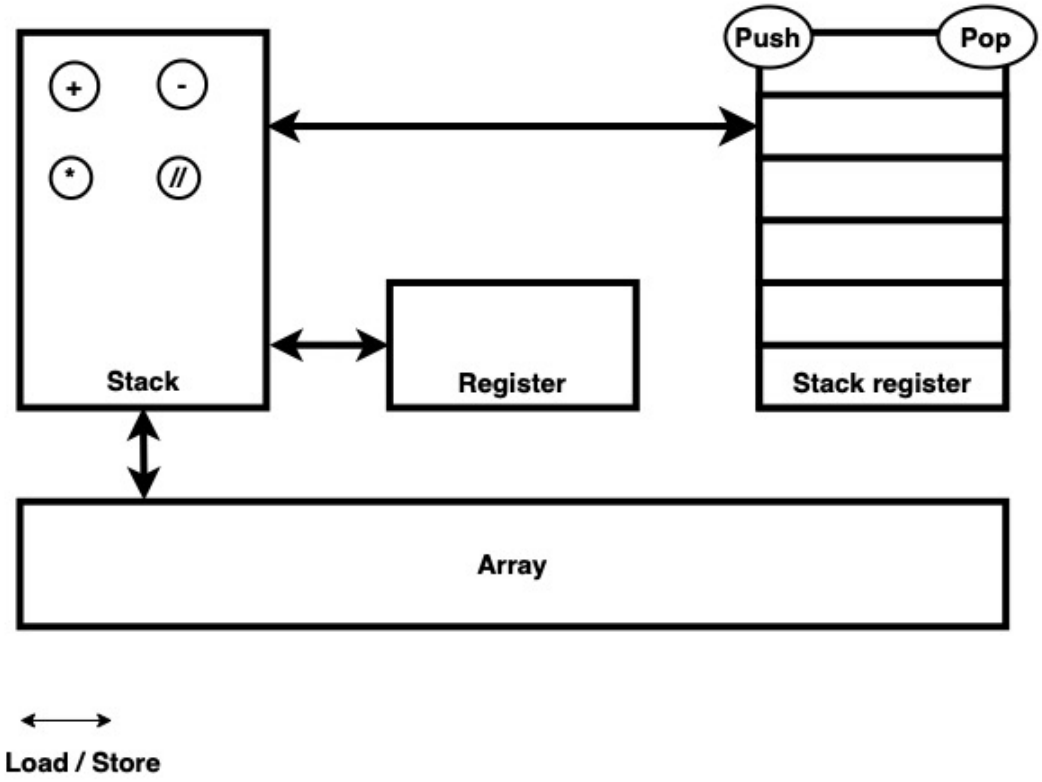

Figure 2.1: Illustration[KGA24] of the sorting algebra architecture, comprising a stack, a fixed-size linear register, a stack of registers, and a variable-length array. Bidirectional arrows represent data flow between these components. The instruction set M_f includes operations for pushing constants ($push\_constant_i$) and metadata ($push\_array\_length$) onto the stack. Arithmetic operations $\circ \in \{+, -, *, //\}$ are performed on the top elements of the stack, while $load$ and $store$ manage data transfer between components.

Taxonomy of Comparison Sorts

The selection of algorithms in previous work[KGA24] was based on a taxonomy of comparison sorting methods, which was determined by their cyclomatic complexity, the dimensionality of the training data, and the number of lines of code. The experiments revealed significant differences in the achieved accuracy between the classes. To account for this in the experimental design of this study, we also consider the taxonomy presented in Table 6.4.

2.6 Summary

In this chapter, we established the theoretical and methodological foundations for the algorithm learning framework presented in this thesis. Central to this foundation is the model of computation based on computation graphs over homogeneous algebras, which enables a formal, algebraic

representation of algorithmic processes. Within this framework, configurations, properties, and predicates form the basis for defining state transitions, while computation graphs structure the execution of functions through directed, predicate-guided traversal.

To enable learning from such structured data, we introduced the concepts of transition algebras and actuators, which allow computations to be reinterpreted as compositions of algebraic functions. This transformation is crucial, as it bridges the gap between algorithmic reasoning and statistical learning, framing the problem as one of supervised classification under the empirical risk minimization (ERM) paradigm.

Recognizing the limitations of ERM under non-stationary data conditions, we discussed the role of distribution shifts — with a focus on covariate and label shifts — and formally demonstrated that these two types of shift cannot occur simultaneously. This insight narrows the scope of our investigation and motivates the adoption of domain generalization techniques.

Finally, we introduced two algorithmic strategies, Meta-Learning for Domain Generalization (MLDG) and Gradient Matching (FISH), that we identify as particularly suited for mitigating the effects of covariate shifts in a later chapter. These methods provide the groundwork for the empirical approaches developed in subsequent chapters.

The chapter concludes by anchoring these concepts in a concrete application domain: comparison sorting algorithms, which serve as a consistent testbed for evaluating the framework throughout this thesis.

3 Related Work

This section situates the present work within the broader landscape of existing literature. Our focus is on the central theme of this study: learning algorithms from memory traces, and we highlight key contributions and developments in this area.

In the context of software systems, Chikofsky et al.[CCM92] define reverse engineering as the process of inferring a more abstract representation of a software system from a more concrete one. However, this approach typically assumes the availability of a complete specification of the system's dynamic behavior, with the reverse engineering process effectively serving as a transformation between different levels of abstraction[KGA24]. This perspective aligns, to some extent, with the view taken in program synthesis research, where the goal is to derive a program from a high-level specification—possibly expressed in natural language [GPS17]. However, a crucial requirement in such approaches is the existence of an explicit specification. In contrast to this, our work adopts a fundamentally different perspective. Rather than assuming access to a complete specification of an algorithm's or system's dynamic behavior, we instead start from memory traces that reflect this behavior empirically.

Such considerations can be traced back several decades. In their 1976 paper Reconstruction of Algorithms from Memory Snapshots of their Execution, Petry et al. [PB76] proposed a method for reconstructing control flow graphs from memory traces. While their work shares similar objectives with ours, there are several important differences, which we elaborate below:

- Structure of snapshots: Petry et al. assume that the structure of each memory snapshot is transparent, meaning that variable names and identities are known.
- Instructions: Their approach assumes access only to the memory contents after each instruction is executed, while the instructions themselves remain unknown.
- Branching: The reconstruction process relies on a temporally ordered sequence of snapshots - i.e., memory traces. A critical aspect of these traces is that they contain information about branching behavior in the underlying algorithm, represented via explicit test nodes.

By contrast, the goal of our work is to develop a method that, in the future, does not require any prior knowledge about the internal structure of a computational state $(v, m) \in V \times M$, nor any meta-information about branching in the training data. Instead, the method should rely solely on training data that reflects state transitions triggered by observable actuator actions.

The idea of combining machine learning with the learning of algorithms is by no means new. A corresponding line of research can be traced back to 1995: Then, Siegelmann et al. [SS95] demonstrated that recurrent neural networks (RNNs) are, in principle, Turing-complete — meaning they can represent any computable function. Building on this concept, Graves et al. introduced the Neural Turing Machine (NTM) in 2014 — a differentiable, memory-augmented model designed to

learn algorithmic behavior from input-output examples. The key difference between the NTM and our approach lies in the objective: rather than merely learning an arbitrary procedure that maps given input-output pairs, our method seeks to recover the underlying process itself—based on the memory traces. In this sense, one could speak of a structure-preserving characteristic.

Finally, there is a range of work that combines machine learning with classical algorithms. For example, Kristo et al. propose a sorting algorithm that accelerates the sorting process using a learned cumulative distribution function (CDF) [KVK21]. While such approaches focus on enhancing specific algorithmic components, our work aims to reconstruct the entire algorithmic process from empirical memory traces, without relying on symbolic specifications or structural assumptions.

4 Sampling Models and Distribution Shifts

As discussed earlier, the task of learning an algorithm from execution data involves a joint distribution of predictors $X \subset V \times M$ and actuators $Y \subset T_G \times M_f$. In this chapter, we introduce two potential stochastic processes that induce such a joint distribution and analyze their implications with respect to changes in the input distribution.

4.1 Distribution of Inputs

As it turns out, the joint distribution of predictors and actuators is ultimately determined by the distribution of inputs. We ground the further discussion on a general computation graph $G = (V, E, M, \phi, \gamma, s, t)$ featuring a countable infinite set of configurations $m \in M$.

Since limiting the discussion to countably infinite instances of M does not limit the computational capabilities[KGA24], we can assume the existence of a distribution of inputs $P_M : \mathcal{P}(M) \to [0, 1]$ turning (M, P_M) into discrete probability space.

4.2 Distribution of Computations

Fundamental to both sampling models is a probability distribution over the set of all computations. To this end, we conceptualize a computation as a multi-stage random experiment, where an initial state $(s, m) \in V\times$ is selected according to the input distribution P_M . We formalize this idea in the following as the probability space (Ω, P_Ω).

4.2.1 Probability Space of Computations

Based on a general computation graph G, we begin our discussion with three fundamental definitions finally leading to the of all computations Ω.

Definition 4.2.1
Let G be a computation graph. The set of initial states is defined as follows:

$$S := \{(s, m) | m \in M\} \tag{4.1}$$

Definition 4.2.2
Let G be a computation graph. The set of terminal states is defined as follows:

$$T := \{(t, m) | m \in M\} \tag{4.2}$$

Definition 4.2.3
Let G be a computation graph. The set of computations is defined as follows:

$$\Omega := \bigcup_{n \in N} S \times (V \times M)^n \times T \tag{4.3}$$

Given these preliminary definitions, we are ready to define a probability measure $P_\Omega : \Omega \to [0, 1]$ turning Ω into a discrete probability space. For this, we assume the inputs to be distributed according to some distribution P_M.

Definition 4.2.4
Let G be some computation graph, S the corresponding set of initial and T of terminal states. The probability $P_\Omega(\omega)$ of some

$$\omega := (s_1, m_1, s_2, m_2, ..., s_n, m_n) \in \Omega \tag{4.4}$$

is defined as:

$$\mathcal{P}_\Omega(\omega) := P_M(m_1) \prod_{i=1}^{n-1} p(s_{i+1}, m_{i+1} | s_i, m_i), \tag{4.5}$$

where

$$p(s_{i+1}, m_{i+1} | s_i, m_i) := \begin{cases} 1 & \text{if } s_i, m_i \vdash_G s_{i+1}, m_{i+1}, \\ 0 & \text{else.} \end{cases} \tag{4.6}$$

Consequently, $(\Omega, \mathcal{P}_\Omega)$ defines a discrete probability space. However, this space alone is insufficient for training data generation, as each sample corresponds to an entire computation. In contrast, a predictor X operates on individual computation states, taking values in $V \times M$. To address this mismatch, we outline two potential processes for generating training data in the following.

4.2.2 Sequential Model

We take a first step toward such a model by introducing a sequential paradigm. While it offers an initial approximation of the sampling process, it is constrained by the assumptions of conventional machine learning methods, which typically require i.i.d. data. These limitations motivate the use of the Monte Carlo model proposed in the subsequent section.

Definition 4.2.5
Let $(\Omega, \mathcal{P}_\Omega)$ be the probability space of computation. The random variable $X_i : \Omega \to V \times M$ is defined as follows:

$$X_i(\omega) = X_i(s, m_0, s_1, m_1, ...) := (s_i, m_i) \tag{4.7}$$

Definition 4.2.6
Let $(\Omega, \mathcal{P}_\Omega)$ be the probability space of computation. The random variable $Y_i : \Omega \to V \times M$ is defined as follows:

$$Y_i(\omega) := \gamma_G(X_i(\omega)) \tag{4.8}$$

To generate a training dataset D, the following steps are performed repeatedly: (1) A computation $\omega \in \Omega$ is sampled from the distribution P_Ω. (2) By applying the random variables, a sequence of training data points is generated and added to the training dataset: $(X_0(\omega), Y_0(\omega)), (X_1(\omega), Y_1(\omega)), ..., (X_n(\omega), Y_n(\omega))$

The tuple-based notation reflects that this procedure preserves the temporal order of data generation. Consequently, the training data points generated in this way in D are not independent and identically distributed. For instance, in the computation graph, a specific state s_i might always follow a given state s_{i+1} , thereby violating a fundamental assumption of empirical risk minimization [HTF09].

4.2.3 Monte-Carlo Model

On the other hand, according to Definition 2.1.7, it is ensured that the predictor $X_i(\omega)$ contains all the information necessary to determine an actuator $Y_i(\omega)$. Consequently, a machine learning model should not take into account correlations between predictors X_i and X_j where $i < j$. In the light of this, we propose an alternative sampling model that addresses this issue.

Let the number of states (s_i, m_i) of a computation $\omega \in \Omega$ be given by $|\omega|$. To generate a training dataset D, the following steps are performed repeatedly: (1) We sample a computation $\omega \in \Omega$ from the distribution P_Ω. (2) The data point $(X_i(\omega), Y_i(\omega))$is added to the training dataset D with the probability $\frac{1}{|\omega|}$

In conclusion, we can say that the data point $(X_i(\omega), Y_i(\omega))_\omega$ is added to the training dataset D with the probability $P_\Omega(\omega)\frac{1}{|\omega|}$. In the following, we are going to adopt the Monte-Carlo-Model.

Implementation: The Python implementation of Monte Carlo sampling is available at https://github.com/pkunz96/Master-Thesis-Induction-Abstraction.

4.3 Theoretical Analysis of Distribution Shifts

These theoretical preliminaries enable a precise analysis of the expected distributional shifts in the context of algorithm learning. In the following section, we focus on covariate shifts, while the following sections are dedicated to concept and label shifts.

Covariate Shifts

We assume M to be a finite countable set of input configurations, such that $(P_M, \mathcal{M})$ forms a discrete probability space of inputs.

Thus, based on the previously in 4.2.3 introduced Monte Carlo model, we can analyze the effects of a changing input distribution. For this, we determine the probability $\mathcal{P}(X = (s, m))$ with respect to $\mathcal{P}_M$.

$$\mathcal{P}(X = (s, m)) \tag{4.9}$$

$$= \sum\nolimits_{\{\omega \in \Omega \mid \exists n \in N : X_n(\omega) = (s,m)\}} \frac{1}{|\omega|} \mathcal{P}_\Omega(\omega) \tag{4.10}$$

$$= \sum\nolimits_{\{\omega \in \Omega \mid \exists n \in N : X_n(\omega) = (s,m)\}} \frac{1}{|\omega|} \mathcal{P}_M(m) \prod_{i=1}^{n} p(s_i, m_i | s_{i-1}, m_{i-1}) \tag{4.11}$$

$$= \sum\nolimits_{\{\omega \in \Omega \mid \exists n \in N : X_n(\omega) = (s,m)\}} \frac{\mathcal{P}_M(m)}{|\omega|} \tag{4.12}$$

This chain of equations illustrates the direct relationship between the input distribution P_M and the distribution of the predictors X. However, it remains to be investigated whether these equations themselves constitute a sufficient criterion for the presence of covariate shifts. In this work this question is addressed empirically — for example, in the subsequent Chapter 5.

Label Shifts

The experimental evidence of covariate shifts presented in Chapter 5 in the context of algorithm learning, together with Theorem 2.7 and Corollary 2.8, allows us to conclude that label shifts can be ruled out.

Concept Shifts

By definition it holds that $Y = \gamma_G$, where γ_G is the computation graphs actuator-state mapping. In other words, the conditional probability $P(Y|X)$ is independent of the distribution of inputs P_M. Consequently, concept shifts can not occur.

4.4 Summary

In this chapter, we introduced Monte Carlo sampling as a proper formal framework for generating training data under controlled input distributions. Additionally, we conducted a theoretical analysis to identify the types of domain shifts that must be addressed during training. Due to the structure of the equation $Y = \gamma_G$, where γ_G defines the actuator-state mapping of the computation graph, the conditional distribution $P(Y|X)$ remains invariant under changes in the input distribution P_M. This implies that concept shifts are, by definition, not possible. However, variations in P_M can induce covariate shifts, which are therefore theoretically expected. For the problem of learning comparison sorts, these covariate shifts are empirically confirmed in Chapter 5. In combination with Theorem 2.7 and Corollary 2.8, this evidence allows us to rule out the presence of label shifts.

5 Experiment I: Quantifying Distribution Shifts

In previous work [KGA24], we observed that changes in input distributions significantly degrade the classification accuracy of models predicting actuators. This empirical finding aligns closely with theoretical predictions presented in the preceding chapter, which anticipated performance impacts due to covariate shifts arising from altered input distributions. Moreover, these results are consistent with the broader literature, which generally highlights reduced model performance under distributional shifts in training data [Mur23; WLL+23b; ZLQ+23].

Although performance degradation under distributional shifts is well documented, in the context of learning sorting algorithms, it remains unclear precisely what types and magnitudes of shifts cause these observed effects. To address this gap, we propose a systematic methodology to quantify distributional shifts explicitly using density-based distance metrics. This approach will provide a concrete measure for assessing domain generalization challenges, offering valuable insights into the selection and evaluation of effective domain generalization strategies, which we subsequently evaluate experimentally.

5.1 Distance Measures

To quantify distributional shifts, the choice of an appropriate metric is crucial. In the light of this, we dedicate this section to a review of common metrics for measuring distances between probability distributions over a σ-algebra[Hen24] in the machine learning literature.

5.1.1 Metric Spaces of Probability Distributions

In order to meaningfully measure distances, it is advised to interpret the set of all probability measures over a σ-algebra as a metric space.

Definition 5.1.1
[For17] A metric d on a set D is a mapping $d : D \times D \to \mathbb{R}_{\geq 0}$ with the following properties:

1. $d(x, y) = 0 \iff x = y$ *(Non-negativity)*
2. $d(x, y) = d(y, x)$ *(Symmetry)*
3. $d(x, z) \leq d(x, y) + d(y, z)$ *(Triangle inequality)*

We denote the tuple (D, d) a metric space.

Following Definition 5.1.1, we introduce this idea formally. For this, let Ω be a non-empty sample space and Σ a σ-algebra over Ω. The set of all probability measures over Σ is given by:

$$C := \{f \mid f : \Sigma \to [0,1],\ f \text{ is a probability measure}\} \tag{5.1}$$

Thus, a proper metric $d : C \times C \to \mathbb{R}_{\geq 0}$ turns (C, d) into a metric space in which distances between distributions can be meaningfully determined. In the interest of relative comparability, we require a fourth property (boundedness):

$$\exists\, c \in \mathbb{R}_{>0} \text{ such that } \forall x, y \in C :\ d(x,y) \in [0,c] \tag{5.2}$$

Consequently, the distance between distributions is upper-bounded by a positive constant c.

5.1.2 Review of Distance Measures

We continue with an evaluation of the distance measures commonly found in the literature with respect to criteria (1–4), in order to ultimately select a suitable measure for our experiments.

Kullback-Leibler-Divergence

$$D_{KL}(P\|Q) = \int_{\mathbb{R}} p(x) \log \frac{p(x)}{q(x)}\, dx \tag{5.3}$$

Kullback-Leibler divergence measures the information loss incurred when distribution P is approximated by distribution Q[Mur22]. However, $D_{\mathrm{KL}}(P\|Q)$ does not satisfy properties (2) and (4)[Bon20].

Jensen-Shannon-Divergence

$$D_{JS}(P,Q) = \frac{1}{2}\Big(\int_{\mathbb{R}} p(x) \log \frac{p(x)}{m(x)}\, dx + \int_{\mathbb{R}} q(x) \log \frac{q(x)}{m(x)}\, dx\Big), \tag{5.4}$$

where $m(x)$ is defined as $m(x) := \frac{1}{2}\,(p(x) + q(x))$.

The Jensen-Shannon Divergence is a symmetrized form of the Kullback-Leibler Divergence [Mur22]. Like the Kullback-Leibler Divergence, the Jensen-Shannon Divergence is non-negative and symmetric [Mur22]. It is also known to be bounded when the logarithm is taken to base 2 [Bon20].

However, while the Jensen-Shannon Divergence itself does not generally satisfy the triangle inequality, its square root does [ES03]. Since the remaining properties translate as well, $\sqrt{D_{\mathrm{JS}}(P,Q)}$ constitutes a true metric, fulfilling properties (1 - 4).

Wasserstein-Distance

$$W_p(P,Q) = \left(\inf_{\gamma \in \Gamma(P,Q)} \int_{X \times X} |x-y|^p\, d\gamma(x,y)\right)^{1/p} \tag{5.5}$$

According to Villani[Vil09], Wasserstein-Distance measures the cost of transforming a distributions P into a distribution Q by relocating probability mass. It satisfies properties (1-3), but is not bounded in general [Vil09].

Total Variation Distance

$$d_{TV}(P,Q) = \frac{1}{2}\int_{\mathbb{R}} |p(x) - q(x)|\, dx \tag{5.6}$$

Total Variation Distance quantifies the maximum difference between the probabilities that the two distributions assign to the same events. According to Feng[FLL]. et al. and Nielsen et al.[NS18] it is a metric satisfying the properties (1-3). Furthermore, it is bounded, with values ranging between 0 and 1 [NS18].

Hellinger Distance,

$$H(P,Q) = \frac{1}{\sqrt{2}}\sqrt{\int_{\mathbb{R}} \left(\sqrt{p(x)} - \sqrt{q(x)}\right)^2 dx} \tag{5.7}$$

From its definition, we can derive that Hellinger Distance quantifies the point-wise discrepancy between the distributions. It satisfys all properties (1-4)[KT17; Rüs14].

5.1.3 Selection

Among the distance measures that satisfy properties (1) to (4), we select Jensen-Shannon Divergence due to its intuitive interpretation as a measure of information loss during distributional approximation. Unlike the Wasserstein Distance, Jensen-Shannon Divergence is bounded in general. While Total Variation and Hellinger distances emphasize pointwise differences in probability mass, the Jensen–Shannon divergence reflects a more globally integrated perspective.

5.2 Experimental Setup

Before presenting the results of our empirical evaluation, we first describe the experimental setup, including the hardware and software used. Particular emphasis is placed on ensuring continuity with our previous work [KGA24], which is especially reflected in the choice of the input distribution model.

5.2.1 Distribution of Inputs

In Chapter 4 we elaborated on the relationship between the distribution of inputs and the thereby through Monte-Carlo inferred joint distribution of predictors X and responses Y. In the light of the input distribution's significance, we ground the remainder of this work on the following well-established[KGA24] distribution of integer inputs.

Definition 5.2.1
[KGA24] Let $X_0 \sim N(\mu_0, \sigma_0)$ and $X_1, X_2, ... \sim N(\mu_1, \sigma_1)$ be a sequence of independent and equally distributed random variables.

We define $P_I : \mathcal{P}(I) \to [0, 1]$ as follows:

$$P_I(A) := \sum_{(a_1,...a_k)\in A} P(\lceil |X_0| \rceil = k, ..., \lceil X_k \rceil = a_k) \tag{5.8}$$

Here, I denotes the set of all integer tuples of finite length, and $A \subseteq I$ is an arbitrary subset. The distribution P_I assigns probability mass to such tuples based on the sampled value of X_0, which determines the tuple length, and the subsequent values $X_1, \ldots, X_k$, which determine the tuple entries. To ensure consistency with the broader learning setup, we adopt a Monte Carlo sampling strategy for generating data instances, following the approach described in Section 4.2.3. In line with prior work, we perform manual feature extraction to isolate the task-specific relevant components from the Sorting Algebra, following the approach outlined in Section 8.2.3.

5.2.2 Distribution of Configurations

As outlined above, each input sequence $i \in I$ deterministically induces a model configuration $m \in M$ through a injective mapping $f : I \to M$, which captures the transformation from raw input data to a model-relevant representation. This mapping allows us to define a probability measure P_M over M by transferring probability mass from the input distribution P_I:

Definition 5.2.2
Let $P_I : \mathcal{P}(I) \to [0, 1]$ be the probability measure over input sequences, and let $f : I \to M$ be a measurable and injective mapping from input sequences to model configurations. The induced distribution $P_M : \mathcal{P}(M) \to [0, 1]$ is defined by:

$$P_M(B) := P_I(f^{-1}(B)) \tag{5.9}$$

for all measurable subsets $B \subseteq M$.

This construction guarantees that for any configuration $m \in M$, the induced probability $P_M(\{m\})$ reflects the probability mass assigned to its preimage:

$$P_M(\{m\}) = P_I(f^{-1}(\{m\})) \tag{5.10}$$

For configurations $m \in M$ without a corresponding preimage under f, the induced probability is zero, consistent with Kolmogorov's axioms[Hen24]:

$$f^{-1}(\{m\}) = \emptyset \implies P_M(\{m\}) = P_I(\emptyset) = 0 \tag{5.11}$$

We choose f as the mapping from an abstract integer sequence to the corresponding python list to be sorted. For the following argumentation, we formally introduce the distribution of the initial model representations as a function parameterized by the parameters of the input distribution.

Definition 5.2.3

$$P^{M}_{\mu^l,\sigma^l,\mu^m,\sigma^m}(B) := P_I(f^{-1}(B)), \tag{5.12}$$

where the parameters of the distribution P_I satisfy:

- $\mu_0 = \mu^l, \sigma_0 = \sigma^l$ *(lengths of tuples)*
- $\mu_1 = \mu^m, \sigma_1 = \sigma^m$ *(values of tuples)*

5.2.3 Density Estimation

In the following, we refer to the distributions $P^{M}_{\mu_0^l,\sigma_0^l,\mu_0^m,\sigma_0^m}$ and $P^{M}_{\mu_1^l,\sigma_1^l,\mu_1^m,\sigma_1^m}$ as:

$$P_0 := P^{M}_{\mu_0^l,\sigma_0^l,\mu_0^m,\sigma_0^m}, \tag{5.13}$$

$$P_1 := P^{M}_{\mu_1^l,\sigma_1^l,\mu_1^m,\sigma_1^m}. \tag{5.14}$$

If a closed-form solution for P_0 and P_1 were available, we could directly analyze the effect of varying the input distributions by adjusting the parameters

$$\mu_i^l, \sigma_i^l, \mu_i^m, \sigma_i^m, \quad i \in \{0, 1\}, \tag{5.15}$$

while computing the distance metric:

$$d(P_0, P_1) := \sqrt{D_{\mathrm{JS}}(P_0, P_1)}. \tag{5.16}$$

However, since a closed-form expression for the distributions is unavailable and their structure remains unknown, we must estimate P_0 and P_1 empirically from data. Consequently, the resulting estimates $\hat{P}_0$ and $\hat{P}_1$ are treated as random variables. This viewpoint enables us to approximate the divergence as follows:

$$\hat{d}(P_0, P_1) := d(\hat{P}_0, \hat{P}_1). \tag{5.17}$$

Naturally, the accuracy of the estimated divergence $\hat{d}(P_0, P_1)$ critically depends on the accuracy of the distributional estimates $\hat{P}_0$ and $\hat{P}_1$. A straightforward approach for obtaining these estimates is to use relative empirical frequencies. From a theoretical perspective, we can assume stochastic convergence of our frequentist estimates[Hen24] and therefore derive:

$$\lim_{n\to\infty} \mathbb{E}(d(P_0, \hat{P}_0)) = 0. \tag{5.18}$$

Consequently, the discrepancy between two independently and sequentially drawn estimates, $\hat{P}_0^t$ and $\hat{P}_0^{t+1}$, is also expected to vanish asymptotically:

$$\lim_{n\to\infty} \mathbb{E}(d(\hat{P}_0^t, \hat{P}_0^{t+1})) \tag{5.19}$$

$$\leq \lim_{n\to\infty} \mathbb{E}(d(\hat{P}_0^t, P_0) + d(P_0, \hat{P}_0^{t+1})) \tag{5.20}$$

$$= \lim_{n\to\infty} \left[\mathbb{E}(d(\hat{P}_0^t, P_0)) + \mathbb{E}(d(P_0, \hat{P}_0^{t+1}))\right] \tag{5.21}$$

$$= \lim_{n\to\infty} [0 + 0] = 0. \tag{5.22}$$

In general, density estimation in high-dimensional spaces is notoriously difficult due to the curse of dimensionality [HTF09]. Indeed, our measurements for Insertion Sort, as presented in Figure 5.1, reveal substantial yet expected discrepancies between independent frequentist density estimates. Given that the amount of training data necessary for reliable density estimation scales exponentially with dimensionality[HTF09], we constrain the remainder of our analysis to lower-dimensional settings—specifically, to InsertionSort with a 9-dimensional dataset as outlined in Table 2.1.

Due to the limited effectiveness of frequentist estimates and the unknown structure of the distribution we continue with a non-parametric method for approximation using kernel density estimation[Sil86]. The basis for the estimation is a dataset

$$D := \{(x_{1,1}, \ldots, x_{1,n}|y_1), \ldots (x_{m,1}, \ldots, x_{m,n}|y_m)\}, \tag{5.23}$$

where a tuple $(x_{i,1}, \ldots, x_{i,n} \mid y_i)$ combines a realization of the random variables X and Y. For further analysis, we transform D into a predictor matrix P defined as follows:

$$P = \begin{bmatrix} x_{1,1} & \cdots & x_{1,m} \\ \vdots & \vdots & \vdots \\ x_{n,1} & \cdots & x_{,m} \end{bmatrix} \tag{5.24}$$

We proceed under the assumption that $\det(P) \neq 0$ such that P^{-1} exists[DFO20]. The bandwidth matrix H is determined using Silverman's rule of thumb[Sil86]:

$$H := \left(\frac{4}{d+2}\right)^{\frac{2}{d+4}} n^{-\frac{2}{d+4}} \Sigma \tag{5.25}$$

Here, Σ corresponds to the covariance matrix of the predictors in P, which is defined as follows[Sil86]:

$$\Sigma := \frac{1}{n-1}(P - \hat{P})^\top (P - \hat{P}), \tag{5.26}$$

where

$$\hat{P} := [\mu_{\mathbf{1}}, ..., \mu_n]^T, \mu_{\mathbf{i}} = AP, \tag{5.27}$$

and

$$A := \begin{bmatrix} \frac{1}{n} & \cdots & \frac{1}{n} \\ \vdots & \vdots & \vdots \\ \frac{1}{n} & \cdots & \frac{1}{n} \end{bmatrix}, A \in \mathbb{R}^{m\times n}. \tag{5.28}$$

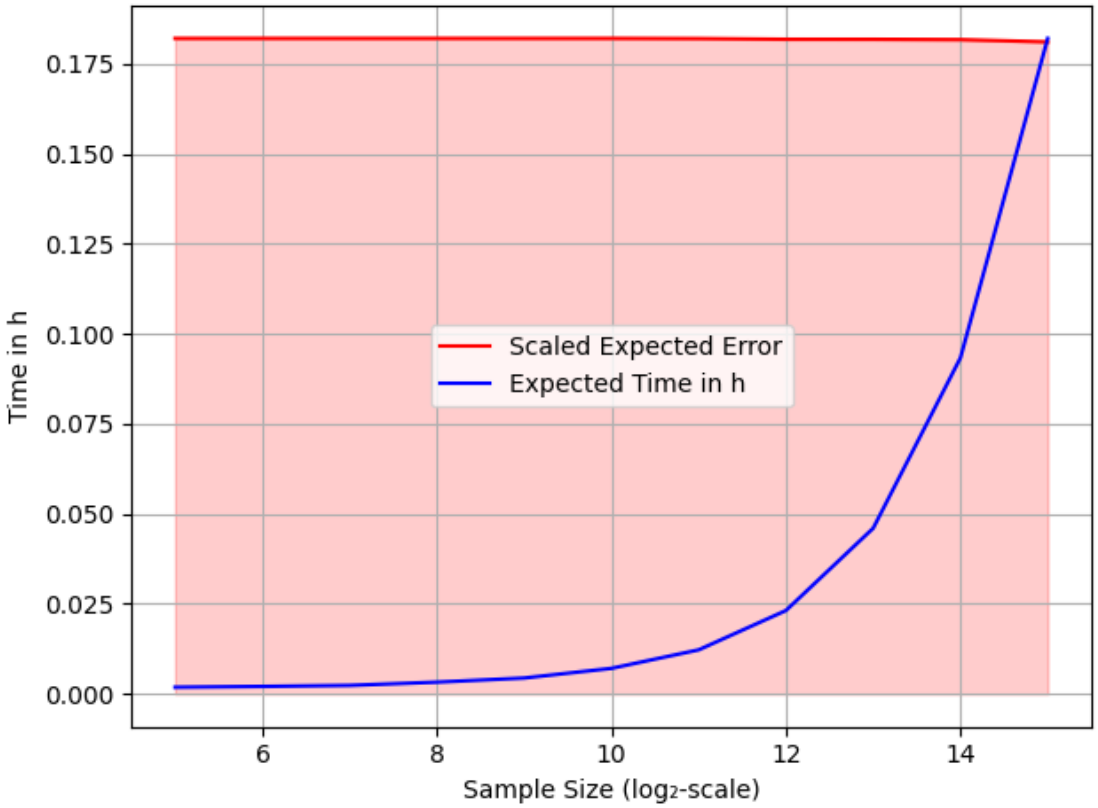

Figure 5.1: We measured the deviation between successive frequentist density estimates in terms of the square root of Jensen-Shannon divergence. To facilitate interpretation, the expected error was rescaled proportionally to the maximum observed expected sampling time. For instance, with a maximum expected sampling duration of 2 hours, an original error value of 0.75 corresponds to a scaled value of 1.5. The underlying input distributions were defined with the parameters: $\mu^l = 5$, $\sigma^l = 0$, $\mu^m = 10$, $\sigma^m = 10$. For each sample size, the sampling process was repeated five times, and both the deviations and durations were averaged across these runs. A key observation is that the expected sampling time increases exponentially with sample size, while the expected error decreases only marginally, especially at larger sample sizes. These findings suggest that frequentist density estimation may be inefficient, especially for large datasets. The slow rate of error reduction relative to computational cost emphasizes the need for more effective alternatives - such as kernel density estimation (KDE) [Sil86] - which offer more practical convergence characteristics.

As shown in appendix A, H is invertible if P is invertible. If P is not invertible, we apply Singular Value Decomposition SVD[DFO20] as a preprocessing step and infer H using the projected data. The kernel density estimate at point $x \in \mathbb{R}^d$ using a multivariate Gaussian kernel is then defined as[Sil86]:

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{(2\pi)^{m/2} det(H)^{1/2}} \exp\left(-\frac{1}{2}(x - x_i)^\top H^{-1}(x - x_i)\right)$$

μ^m	σ^m	μ^l	σ^l
0	1	5	0
0	2	5	0
2	1	5	0
2	2	5	0
4	1	5	0
4	2	5	0

Table 5.1: The table summarizes the parameter configurations that define the input distributions. Here, μ^m denotes the expected value of a tuple entry, and σ^m its standard deviation. Similarly, μ^l also represents the expected length of a tuple, with σ^l indicating the standard deviation of the length.

where:

- n is the number of data points
- m is the number of dimensions
- H is the bandwidth matrix
- $det(H)$ is the determinant of H
- H^{-1} is the inverse of the bandwidth matrix

The sample size n is critical for the quality of the estimation. Starting from $n = 8$, we systematically double it and measure the expected deviation ϵ_i across all configurations $c_1, ..., c_k$ as outlined in Table 5.1 - five samples of size n are drawn per configuration - between two successive estimates, or formally:

$$\epsilon_i := \mathbb{E}(JS_{DS}(\hat{P}_i^t - \hat{P}_i^{t+1})) \tag{5.29}$$

Based on the previously developed theoretical framework, we analyze how variations in input distributions affect the resulting distributions of the predictors. To this end, we estimate the distributions P_i for Insertion Sort. We then compute the pairwise Jensen–Shannon divergence between these distributions and determine the maximum error

$$\epsilon := max\{\epsilon_1, ..., \epsilon_k\}. \tag{5.30}$$

5.2.4 Hardware and Software

This experiment was conducted on a PC equipped with an AMD Ryzen 9 5950X CPU, 64GB of RAM, and an NVIDIA GeForce RTX 3090 GPU. A summary of the software environment is provided in Table 8.2. The source code is available at https://github.com/pkunz96/Master-Thesis-Induction-Abstraction.

Software	Version
Python	3.12
SciPy	1.15.2
Numpy	2.1.3
Pandas	2.2.3
Matplot	3.10.1
Seaborn	0.13.2
CuPy	13.4.1
cuML	24.04.00

Table 5.2: Summary of the Installed Software Versions on Ubuntu 24.04.2 LTS for Experiment I

5.3 Results

In the previous section, the experimental setup and the applied methodology were explained in detail. This section now presents the analysis of the results obtained from them.

Figure 5.2 shows the pairwise Jensen-Shannon divergences for *Insertion Sort* based on the input distribution parameters presented in Table 5.1. The values along the main diagonal approach a distance of 0, which is not only expected due to identical parameters but also required by property x inherent to the JS distance. In contrast, distribution pairs with differing parameters consistently yield a maximum JS distance of 1. Consequently, we conclude that even minor changes in the input distribution can substantially alter the distribution of predictors, thereby posing a significant challenge for domain generalization.

5.4 Summary

This experiment investigates how shifts in input distributions affect the distribution of predictors in the context of algorithm learning using the examples of sorting algorithms.

A review of common distance measures leads to the selection of the square root of Jensen-Shannon Divergence as the most suitable metric.

The experimental setup uses the Monte Carlo model as discussed in Section 4.2.3 for data generation, while varying the distribution of integer sequences to be sorted. Since the underlying predictor distributions are unknown, kernel density estimation (KDE) is employed to approximate them.

Empirical analysis using Insertion Sort as an example show that: (1) Even minor shifts in input distribution significantly alter the predictor distributions (covariate shift). (2) For identical input parameters, the distance approaches zero, as expected. (3) For differing parameters, it reaches a value close to its maximum of 1, signaling high sensitivity to input changes.

These findings underscore the inherent challenges of domain generalization in learned algorithmic behavior and emphasize the necessity of explicitly quantifying distributional differences

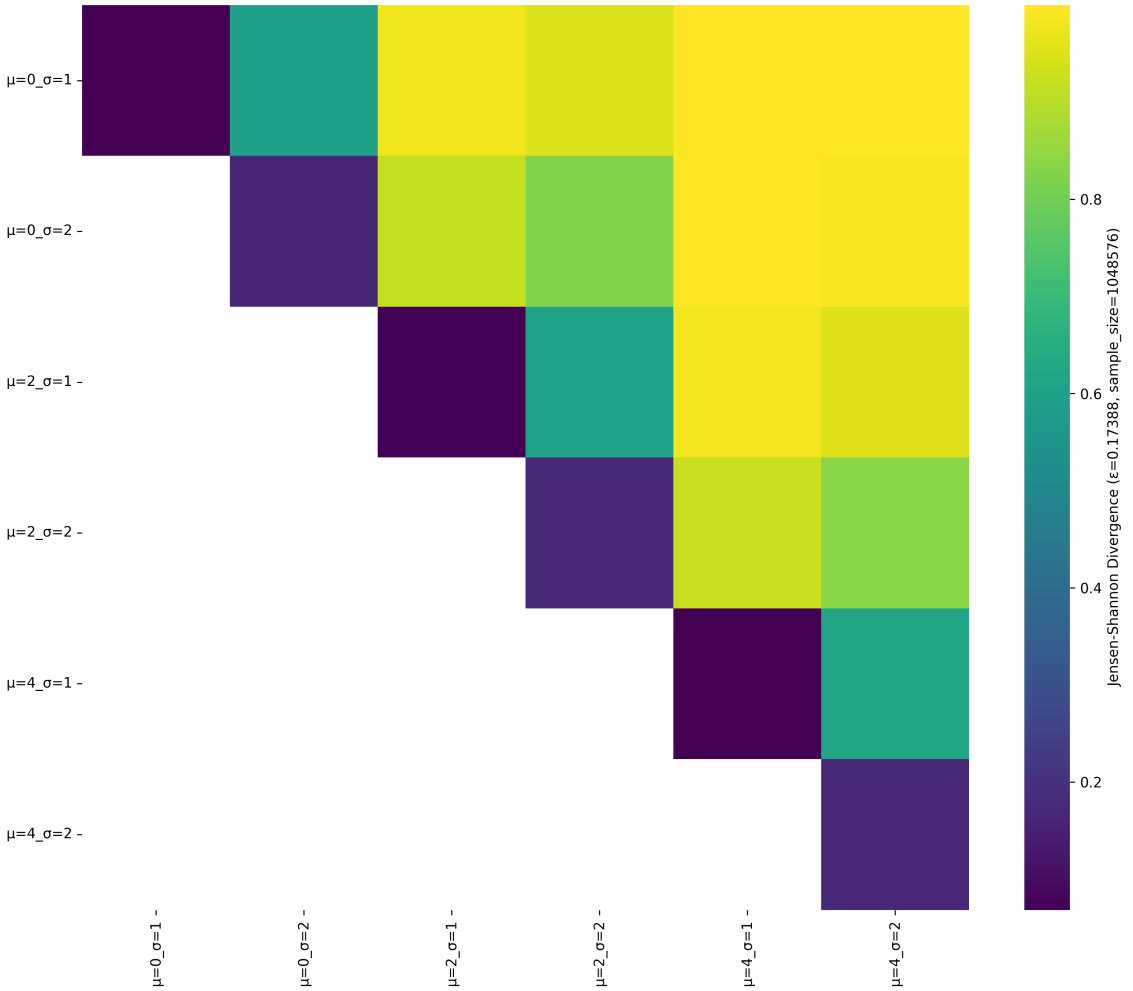

Figure 5.2: Illustration of the pairwise square root of Jensen-Shannon divergences for the Insertion Sort algorithm, computed using the input distribution parameters outlined in Table 5.1. The matrix's diagonal entries approach zero, as expected when comparing identical distributions—consistent with the identity and symmetry properties of the chosen metric. In contrast, off-diagonal entries, corresponding to distribution pairs with differing parameters, consistently reach the maximum divergence value of 1. This pattern reveals a high sensitivity of predictor distributions to even minor input variations, highlighting the significant challenge domain generalization poses in this setting.

when evaluating such systems. Furthermore, the presence of significant covariate shifts — together with Theorem 2.7 and Corollary 2.8 — allows us to rule out the occurrence of label shift in our setting. This insight informs the methodology of the subsequent chapter, where we draw on these results to guide the selection of appropriate domain generalization techniques to mitigate the observed performance degradation.

6 Review of Domain Generalization Methods

In the previous chapter, we quantified the distribution shifts that occur in the context of learning sorting algorithms and identified them as covariate shifts based on their characteristics. Building on this analysis and additional requirements, this chapter focuses on selecting an appropriate framework for addressing covariate shifts, as well as suitable methods within that framework. We begin by assessing the suitability of various approaches and identify Domain Generalization as the most appropriate. To inform the selection of specific domain generalization methods, we justify the use of the taxonomy proposed by Wang et al. as a solid foundation, enabling a systematic and well-reasoned selection of methods for the empirical evaluation in the next chapter.

6.1 Prerequisites

Before we turn to selecting an appropriate framework, this section summarizes the context, requirements, and constraints, as well as the key factors and requirements relevant to this work.

6.1.1 Context

As a consequence of our preliminary analysis, we focus on methods for handling covariate shifts. Specifically, we assume a multi-source domain generalization setting, where multiple training datasets $D_1, \ldots, D_k$ are available, each drawn from a distinct distribution $D_i \sim P_i(X, Y)$, with $P_i \neq P_j$ for all $i \neq j$. The test distribution, in turn, is assumed to differ from all training distributions or formally: $\forall i \in \{1, ..., k\} : P_i \neq P_{test}$

6.1.2 Requirements

We aim to develop a classification model that generalizes robustly to covariate shifts in the test distribution P_{test} and is deployable out-of-the-box without additional adjustments. To ensure scalability and consistency, we prioritize an end-to-end training approach that minimizes manual preprocessing, feature engineering, intermediate steps, and domain-specific adaptations. This enables efficient application to new input distributions without requiring further fine-tuning.

6.1.3 Constraints

Building on previous work [KGA24], we initially limit the number of training data points per domain— rather than across the entire dataset — as follows:

(6.1) $2^e, e \in \{13, 14\}$

6.2 Framework Selection

In the preceding chapters, we have extensively presented methods for handling shifting distributions. Building on this, the previous sections once again reflect on the requirements and constraints of the problem domain – the learning of algorithms. Based on this requirement profile, we now make a well-founded selection of a framework, whose effectiveness we will experimentally validate in the following.

Property	**Condition**
Number of Domains K	$K > 1$
$P_{training}$ vs. P_{test}	$\neq$
$Y_{training} = Y_{test}$	Yes
Covariate Shifts	Yes
Concept Shifts	No
Label Shifts	No
Access to P_{test}	No

Table 6.1: A summary of the key properties of the generalization problem as discussed by this work, where K equals the number of domains included in the training data set and $Y_{training} = Y_{test}$ indicating that the labels does not change between training and test time. We additionally assume that test distribution P_{test} is unavailable at training time.

Framework	**Requirement**	**Condition**
Supervised Learning (ERM)	$K > 1$	$K = 1$
Multi-Task Learning	$P_{training} \neq P_{test}$	$P_{training} = P_{test}$
Transfer Learning	$Y_{training} = Y_{test}$	$Y_{training} \neq Y_{test}$
Zero-Shot Learning	P_{test} not available	P_{test} required
Domain Adaption	P_{test} not available	P_{test} required

Table 6.2: A tabular comparison[ZLQ+23] of the previously introduced methods for handling changing distributions with the requirements outlined in table 6.1 in the context of the problem statement.

Domain generalization as a framework meets all the requirements outlined in Table 6.1, whereas Table 6.2 provides an overview of the properties that other frameworks lack. Therefore, we focus on this framework for the remainder of this work.

6.3 Literature Review

It remains to be determined which domain generalization methods are best suited for addressing the expected domain shifts in this work. To identify such methods, we conduct a literature review. As a first step, we introduce a structured taxonomy to navigate the domain generalization landscape.

6.3.1 Methodology

The articles *"Generalizing to Unseen Domains: A Survey on Domain Generalization"(2023)* by Wang et al.[WLL+23b] and *"Domain Generalization: A Survey"(2023)* by Zhou et al.[ZLQ+23] are two widely adopted surveys published in highly regarded venues, namely TKDE[IEE25c] and TPAMI[IEE25d]. Both structure existing work on Domain generalization hierarchically. Given their credibility and comprehensive coverage, we adopt these works as a baseline for our analysis.

Sources and Search Strategy

To ensure the inclusion of all relevant survey publications on domain generalization, we conduct a systematic search across the following academic databases:

- arXiv[Uni25],
- IEEE Xplore[IEE25e],
- ACM Digital Library[IEE25a],
- Springer Link[Lim25]

If a source supports filtering survey papers, we query it with the keyword 'domain generalization' and use that filter. Otherwise, we query it with the following keyword combinations:

- 'domain generalization, survey
- 'domain generalization, review
- 'domain generalization, meta-analysis

Among the filtered results, we preselect those that contain the aforementioned keywords in their title for further analysis. In Table 6.3, we list the identified publications along with meta-information such as the publication year, the h-index of the first author, and citation count, which serves as a basis for the subsequent selection steps.

The previously outlined criteria serve as the basis for selecting a suitable survey. Finally, we examine the most recently accepted papers at the following major AI conferences to identify gaps in the form of later published high-impact publications on the topic:

- NIPS24[Neu24] and NIPS23[Neu23]
- ICLR24[Int24a] and ICLR23[Int23a]
- ICML24[Int24b] and ICML23[Int23b]

Title	Year	Publisher	Citations	h-Index
Domain generalization through meta-learning: A survey[KYF24]	2024	-	10[Alp25e]	3[Alp25e]
Domain Generalization in Computational Pathology: Survey and Guidelines[JRX+23]	2023	-	14[Alp25o]	20[Alp25o]
Domain Generalization for Medical Image Analysis: A Review[YOS+24]	2024	IEEE	15[Alp25c]	12[Alp25c]
Federated Domain Generalization: A Survey[LWZ+24]	2024	-	50[LWZ+24]	6[LWZ+24]
A Comprehensive Review of Trends, Applications and Challenges In Out-of-Distribution Detection[GF22]	2022	-	50[LWZ+24]	6[LWZ+24]
Generalizing to Unseen Domains: A Survey on Domain Generalization[WLL+23b]	2023	TKDE	323[Alp25j]	45[Alp25j]
Domain Generalization: A Survey[ZLQ+23]	2023	TPAMI	323[Alp25h]	28[Alp25h]

Table 6.3: Overview of articles on domain generalization identified using the methodology outlined previously. The 'h-Index' column represents the overall h-index of the first author. Citation count and i-index values were primarily extracted from Google Scholar on January 27, 2025. For the works of Wang et al. and Zhou et al., citation data from IEEE Xplore were used due to discrepancies in Google Scholar, which referenced earlier versions of these publications. The publisher is listed for publications that have undergone a peer-review process.

Selection Criteria

Among the preselected publications, we include only peer-reviewed works, as indicated by the publisher's name in Table 6.3, for further consideration. We then prioritize these publications based on the number of citations as the primary criterion, followed by the publication date. In cases of ambiguity, we resolve ties by considering the h-index of the first author. As for papers published in the aforementioned venues, we restrict our selection to those with at least 50 citations, using this threshold as an indicator of rapid adoption and high impact.

Results

Surveys: Applying the criteria outlined above we identify the taxonomy of Wang et al. as the most suitable choice. Wang et al. [WLL+23b] distinguish between three fundamental types of DG methods: data manipulation, representation learning, and approaches addressing the learning procedure. We introduce these in the following.

Conference Papers: At ICLR23, Bai et al.[BLZ23] introduced a domain generalization approach centered on temporal adaptation. Their method addresses scenarios where models must adapt to evolving distributions over time, exemplified by a credit risk prediction model that adjusts to shifting economic conditions. While this work is valuable in settings where domain shifts occur due to temporal factors, our research focuses on generalization to unseen domains at any point in time, rather than adapting models dynamically over time. Consequently, we consider this publication less relevant to our research objectives. Furthermore, the remaining conference papers did not meet our established citation threshold.

6.4 Taxonomy of Domain Generalization Methods

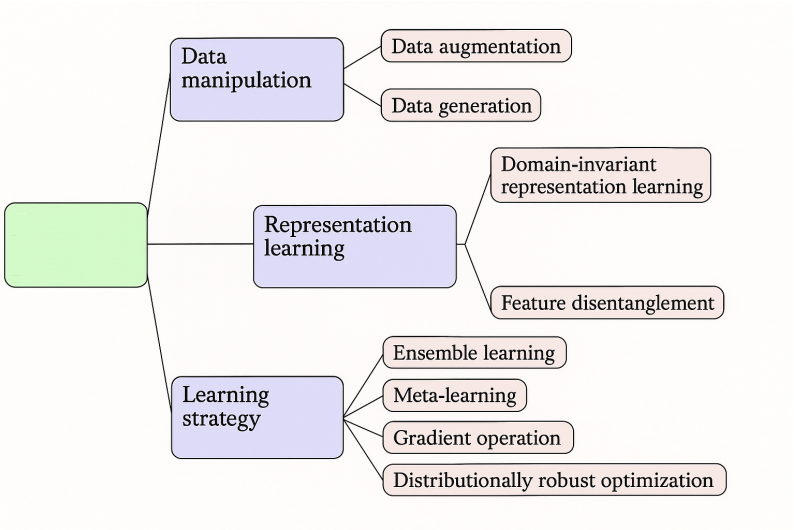

Figure 6.1: Taxonomy of Wang et al.[WLL+23b] showcasing the main categories: data manipulation, representation learning, and learning strategy.

In this section, we provide a detailed overview of the taxonomy proposed by Wang et al. [WLL+23b] on domain generalization, as illustrated in Figure 6.1. This taxonomy categorizes domain generalization methods into three fundamental groups: data manipulation, representation learning, and learning strategy. In the following subsections, we explore each category. Unless stated otherwise, this section follows the framework proposed by Wang et al. [WLL+23b].

6.4.1 Data Manipulation

This class of algorithms modifies the training data set to increase the diversity and quantity of training data and thereby improve generalization performance. Formally, this can be expressed in terms of some training data point x, a classifier h to be learned, and some transformation M to infer synthetic training data:

(6.2) $x' = M(x)$

The training objective is then to minimize the joint loss for both actual and generated training data, or formally:

(6.3) $min_h[\mathbb{E}_{x,y}(l(h(x), y) + \mathbb{E}_{x',y}(l(h(x'), y)]$

Depending on the implementation of M, we can further distinguish between two subcategories: in case of data augmentation, M corresponds to static, manually applied transformations such as cropping or rotation, whereas in the case of data generation, stochastic models like variational autoencoders (VAEs)[KW22] or generative adversarial networks (GANs)[GPM+14] are employed to implement T for data synthesis.

6.4.2 Representation Learning

In [BBCP06] Ben-David et. al. prove domain-independent feature representations to be transferable across domains. Their work serves as theoretical justification for the representation learning framework. In this context, an algorithms aims at learning a classification function

(6.4) $h(x) = f \circ g(x),$

where g represents a domain-invariant feature extraction function. A machine learning algorithm of this kind is designed to minimize the following expression:

(6.5) $min_h[\mathbb{E}_{x,y}(l(h(x), y) + \lambda l_{reg}] = min_{f,g}[\mathbb{E}_{x,y}(l(f(g(x)), y) + \lambda l_{reg}],$

where λl_{reg} denotes regularization term. We can further distinguish between domain-invariant learning and feature-disentanglement as detailed in the following.

Domain-Invariant Learning

The goal is to improve a model's performance on unseen domains by learning domain-invariant feature spaces. Subcategories according to Wang et al.[WLL+23b] are:

- kernel methods,
- explicit feature methods,
- domain adversarial learning,
- invariant risk minimization.

Feature Disentanglement

Here, we aim at improving a model's performance on unseen domains by splitting features into domain-specific and domain-invariant components (disentanglement). Wang et al.[WLL+23b] further classify these methods into:

- multi-component analysis,
- generative modeling,
- causality-inspired method.

6.4.3 Learning Strategy

Learning strategy-based approaches to domain generalization adjust the underlying learning procedure to improve the generalization performance of the resulting models. We discuss the respective subcategories following Wang et al.[WLL+23b] in the following.

Ensemble Learning

In case of ensemble learning for domain generalization, a training data point is considered as a superposition among the domains represented in the training data. An ensemble of machine learning models is set up in a way that ensures that at test time the most suitable model within the ensemble is in charge of making the prediction. Hastie et al.[HTF09] define ensemble learning as the process of combining simpler base models into stronger learner.

Meta-Learning

Thrun et.al[TP98] define meta-learning as the process by which a learning algorithm learns how to learn. More specifically, Finn et al.[FAL17] transfer this idea to gradient-descent-based optimization in the form of 'Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks'. Li et al.[LYSH17b] finally developed this approach further leading to the MLDG as introduced in Section 2.4.2.

Gradient-Operation

This family of methods leverages gradient information to encourage the model to learn generalized representations. For example, gradient matching approaches such as FISH [SST+21], described in Section 2.4.3, represent this class. FISH employs a customized optimization objective that maximizes the scalar product of loss gradients across domains, thereby promoting inter-domain alignment.

Distributionally Robust Optimization

Introduced by Rahimian et al.[RM22], distributionally robust optimization (DRO) is a method designed to optimize a model's performance under the worst-case distribution within a predefined set of plausible probability distributions, also referred to as the uncertainty set.

Self-Supervised Learning.

Self-supervised learning-based leverages auxiliary tasks, such as constructing representations from unlabeled data, to facilitate the learning robust, domain-invariant features. A representative of this class is, for instance, Carlucci et al.'s work 'Domain generalization by solving jigsaw puzzle'.

6.5 Method Selection

Building on the insights from Wang et al.'s taxonomy[WLL+23b], we now focus on identifying works in domain generalization that are most relevant to this context. We begin by specifying the criteria that candidate approaches must satisfy.

6.5.1 Criteria

Foundational work must meet the following criteria to be considered as suitable in the context of this work, namely: (1) it must be able to handle covariate shifts, (2) it should not make any assumption on the distribution of features in advance (3) it must support end-to-end training including the extraction of meaningful features, (4) the work should have undergone a rigorous peer-review, (5) empirical results reflect the performance across different types of data sets, and finally (6) reproducibility must be ensured through provided code.

6.5.2 Discussion

To further streamline the selection process, we first discuss the suitability of the overarching categories[WLL+23b]: Data Manipulation, Representation Learning, and Learning Strategy.

Data Manipulation

We begin our evaluation with 'data augmentation'. For this, we assume a perfect implementation of $classify$ in the form of a function $\delta : V \times M \rightarrow T_G$, which assigns the correct function of the transition algebra to a computation state $(v, m), v \neq t$.

Central to the application of data augmentation is the choice of an appropriate label-preserving transformation. Specifically, the transformation $T : V \times M \rightarrow V \times M$ must satisfy the following property:

$$\delta(v_1, m_1) = T_i \implies \delta(T(v_1, m_1)) = T_i \tag{6.6}$$

To verify that T satisfies this requirement, it must be checked that the values assigned by the relevant predicates remain unchanged. But these are unknown unknown in advance, highlighting the impracticality of this approach.

As for data generation, where T is becomes of stochastic nature, consider the following predicate:

$$q_1(m) := q_2(m) \wedge q_3(m) \tag{6.7}$$

In this case, q_2 and q_3 equally influence the value of q_1. Now, let us assume that q_2 compares two values $\pi_1(m)$ and $\pi_2(m)$ which are encoded in some $m \in M$ with respect to the relation $<$:

$$q_2(x) = \begin{cases} 1, & \text{if } \pi_1(x) < \pi_2(x) \\ 0, & \text{else} \end{cases} \tag{6.8}$$

A deviation $\epsilon > \pi_2(x) - \pi_1(x)$, as expectable due to $T's$ stochasticity, would change the value of the predicate q_1 and thereby fail to preserve the semantics of the training data point.

In summary, unlike in other application domains such as computer vision [VM19], the semantics do not change gradually but rather in a discrete manner. This shows that data manipulation techniques cannot be easily applied to this problem, as they require strict control over the predicate values.

Representation Learning

Consider a computation graph $G = (V, E, M, \phi, \gamma, s, t)$, where $(v_1, m_1) \in V \times M, v \neq t$ is a computation state. As previously discussed in Section 2.1, then it holds:

$$\exists! v_2 \in V, m_2 \in M : \gamma(v_1, v_2)(\Pi_{(\pi_1,\ldots,\pi_n)}(m_1)) = 1 \iff (v_1, m_1) \vdash_G (v_2, m_2) \tag{6.9}$$

Based on this observation, we define the function $g : V \times M \to V \times \{0, 1\}^m$ as follows:

$$g(v, m) = (v, \gamma(e_1)(\Pi_{(\pi_1,\ldots,\pi_n)}(m_1)), \ldots, \gamma(e_m)(\Pi_{(\pi_1,\ldots,\pi_n)}(m_1)))^T \tag{6.10}$$

This function generates for each edge e in G a binary representation, if it is passable given some $m \in M$. Consequently, for a state (v, m), the progression of the computation is thus fully explained by a predicate over the latent feature space $g(v, m)$.

This perspective enables the decomposition of $classify : V \times M \to T_G$ in the following manner:

$$classify = f \circ g \tag{6.11}$$

This decomposition corresponds to the previously introduced approach of representation learning. Therefore, it seems only natural to pursue representation learning as a means of addressing the problem of changing test distributions.

Learning Strategy

This class encompasses a wide range of methods. For example, [**<empty citation>**] et al. use adversarial training to learn a domain-invariant latent feature space, while [**<empty citation>**] relies on ensemble methods combined with SVMs. In this context, we do not rule out any of the works a priori.

Conclusions

We exclude data manipulation techniques from our further considerations and focus on the categories representation learning and learning strategy.

6.5.3 Selection

After these preliminary steps, we are now ready to select suitable domain generalization methods for the experimental evaluation. To this end, we apply the previously defined selection criteria to the two most cited publications within each category of Wang et al.'s taxonomy. Table 6.4 presents a detailed overview of the evaluation results. From the remaining non-excluded candidates, we select the most cited method in each applicable category. Specifically, we choose *Learning to Generalize: Meta-Learning for Domain Generalization* by Li et al. [LYSH17b] and *Gradient Matching for Domain Generalization* by Shi et al. [Alp25k], which introduces the FISH algorithm, as discussed in Section 2.4.3. These methods satisfy all outlined requirements and will form the foundation of our subsequent empirical evaluation.

6.6 Summary

This chapter presents a structured review of domain generalization methods to address covariate shifts in the context of algorithm learning. After evaluating several frameworks, domain generalization is selected as the most suitable due to its alignment with the problem's constraints and goals. A literature review leads to the adoption of Wang et al.'s taxonomy, which categorizes domain generalization methods into data manipulation, representation learning, and learning strategy. Data manipulation is ruled out due to incompatibility with the task's discrete semantics, while representation learning and learning strategy are deemed appropriate. Ultimately, two peer-reviewed and widely cited methods — Meta-Learning (MLDG) and Gradient Matching (FISH) — are finally selected for empirical evaluation in Chapter 8.

Table 6.4: Lists for each category of Wang et al.'s taxonomy the two most cited papers and their disqualifier, if existent. The disqualifier codes are as follows: OT (Off-Topic), ETE (End-To-End Training), FE (Feature Extraction), PR (Peer-Review), DA (Distributional Assumptions), EVDS (Experimental Validation and Data Set), PD (Family of Probability Distributions).

Method	Disqualifier	Citations
Kernel Methods		
DICA[MBS13]	FE, ETE	1347[Alp25f]
SCA[GBKZ16]	FE, ETE	520[Alp25q]
Explicit Feature Alignment		
Unified Deep Supervised Domain Adaptation and Generalization[MPAD17]	EVDS	1021[Alp25t]
Domain Generalization for Object Recognition with Multi-task Autoencoders[GKZB15]	EVDS	804[Alp25d]
Domain Adversarial Learning		
Domain-Adversarial-Training of Neural Networks[GUA+16]	OT	10685[Alp25i]
Domain Generalization with Adversarial Feature Learning[LPWK18]	EVDS	627[IEE25b]
Invariant Risk Minimization		
IRM[ABGL20]	PR	2406[Alp25l]
Ensemble Learning		
Domain Adaptive Ensemble Learning[ZYQX20]	EVDS	314[Alp25b]
Domain Generalization with Domain-Specific Aggregation Modules[DC18]	EVDS	169[Alp25g]
Meta-Learning		
Learning to Generalize: Meta-Learning for Domain Generalization[LYSH17b]	None	1587[Alp25m]
MetaReg: Towards Domain Generalization using Meta-Regularization[BSC18]	None	810[Alp25n]
Gradient-Operation		
Self-Challenging Improves Cross-Domain Generalization[HWXH20]	EVDS	729[Alp25r]
Gradient Matching for Domain Generalization[SST+21]	None	318[Alp25k]
Distributionally Robust Optimization		
Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization[SKHL20]	PD	1887[Alp25a]

Method	Disqualifier	Citations
Out-of-Distribution Generalization via Risk Extrapolation (REx)[KCJ+21]	PD	991[Alp25p]
Self-Supervised Learning		
Domain Generalization by Solving Jigsaw Puzzles[CDB+19]	ETE	1018[CDB+19]
SelfReg: Self-supervised Contrastive Regularization for Domain Generalization[KPKL21]	ETE	335[Alp25s]

7 Domain-Invariant Binary Embeddings (DIBE)

We aim at leveraging the structure of computation graphs to enforce domain-invariant representations via a binary bottleneck, aiming for improved generalization across domains compared to the previously identified baselines. The idea of constraining representational capacity is well-established in deep learning, notably in the context of autoencoders [GBC16]. In Section 6, we demonstrated that the classifier $classify$, which predicts the correct function from the transition algebra based on a computation state (v, m), can be decomposed as follows:

(7.1) $classify = f \circ g,$

where $g : V \times M \to V \times \{0,1\}^m$ for some $m \in \mathbb{N}$, and $f : V \times \{0,1\}^m \to T_G$. In the following, we refer to g as encoder and to f as decoder.

Building on this decomposition, in this chapter, we propose a domain generalization method tailored to computation graphs and the structure of $classify$, specifically designed to satisfy the requirements set out in Section 6.1.2.

7.1 Theoretical Analysis

In this section, we analyze the extent to which this decomposition is suitable for constructing a classifier that can generalize across domains and thereby consistently achieve high accuracy. We begin our discussion with a brief recap of the relevant properties of computation graphs.

7.1.1 Computation Graphs and Algorithm Learning Revisited

The control flow within a computation graph is governed by predicates over property spaces. For a computation based on properties $\pi_1, \ldots, \pi_n$, the corresponding property space is defined as follows:

(7.2) $\Pi^M_{(\pi_1,\ldots,\pi_n)} := \{(\pi_1(m), \ldots, \pi_n(m)) \mid m \in M\}$

If the computation is currently in a state $v \in (V \setminus \{s\})$ with possible successor states $v_1, \ldots, v_l$, that is, formally

(7.3) $(v, v_1), \ldots, (v, v_l) \in E,$

then the transition to one of the possible successor states is governed by:

(7.4) $\forall m_i \in M \exists m_{i+1} \in M : (v, m_i) \vdash_G (v_i, m_{i+1}) \iff \gamma(v, v_i)(\pi_1(m), ..., \pi_n(m)) = 1$

Here, the function γ assigns to each outgoing edge (v, v_i) a predicate:

$$p : \Pi^{M}_{(\pi_1,\ldots,\pi_n)} \to \{0,1\} \tag{7.5}$$

Statement (7.4) thus establishes that a transition from state v to state v_i is possible if and only if the corresponding predicate evaluates to 1 when applied to the property values of m_i.

Suppose there are exactly k states v that have multiple successor states in the sense of Equation (7.3), namely $v_1, \ldots, v_l$ with $l > 1$. Let $l_{\max}$ denote the maximum number of successors l among those k states. Then there are at most

$$h := k \cdot l_{\max} \tag{7.6}$$

predicates $p_1, \ldots, p_h$ that can be evaluated simultaneously. For a given state v, the successor state v' is fully determined by the vector:

$$(p_1(\Pi_{(\pi_1,\ldots,\pi_n)}(m)), \ldots, p_h(\Pi_{(\pi_1,\ldots,\pi_n)}(m))) \in \{0,1\}^h \tag{7.7}$$

In other words, this vector contains all the information required to implement the function $classify : V \times M \to T_G$, and thus to predict the correct operation of the transition algebra in Algorithm 1.

If the topology of the computation graph G is known — as assumed in previous work[KGA24] — then h can be concretely bounded from above. Therefore, we will treat h as a fixed constant in the following.

7.1.2 Domain Generalization through a Binary Bottleneck

In the following, we analyze to what extent a latent feature space of the form $\{0,1\}^h$ can enable the classifier $f \circ g$ to generalize across domains. To support this analysis, we adopt a machine learning perspective — specifically, the domain generalization (DG) framework[BLS11] — and assume access to a training dataset of the form:

$$\mathcal{D} = \left\{\mathcal{D}_k = \left\{(x^{(k)}, y^{(k)})\right\}\right\}_{k=1}^{K}, \quad x^{(k)} \in \Pi_{(\pi_1,\ldots,\pi_n)}(M), \quad y^{(k)} \in T_G \tag{7.8}$$

A key insight underlying our approach is that the decoder f loses the ability to distinguish between domains as the number of domains increases, due to the syntactic structure of inputs $x \in \{0,1\}^h$. Assume that $\mathcal{D}$ contains $K > 2^h$ domains. For each domain $\mathcal{D}_i$ and each input $x \in \mathcal{D}_i$, we compute the output $f(x)$. Since each domain is finite, this process terminates for all i. By the pigeonhole principle, after at most $2^h + 1$ domains, there must exist at least two domains whose samples are mapped to the same output under f. This reasoning can be extended recursively: as K increases, f is forced to map inputs from different domains to overlapping outputs, and thus becomes unable to distinguish between most domains. Consequently, the latent representation produced by g must focus on features that are invariant across domains, which encourages domain generalization.

A second line of motivation arises from the notion of semantic consistency: Since the decoder f has no access to information about the origin domain, each binary code $z \in \{0,1\}^h$ must

be interpreted independently of the domain. As the number of domains increases, the probability grows that semantically identical functions from different contexts must be mapped to the same code. To ensure consistent predictions in such cases, the system is forced to learn encodings that are robust across domains. This structural requirement for consistency acts as an implicit mechanism that promotes domain invariance.

These observations motivate an empirical investigation into whether binary embeddings constructed in this way indeed support generalization across domains. Since the decoder f operates without domain-specific context and must interpret latent codes uniformly, the system is implicitly encouraged to produce representations that capture features invariant to domain shifts. We therefore hypothesize that enforcing a binary bottleneck of size h can promote domain robustness — a claim we validate experimentally in the following sections.

7.2 Feasibility Analysis

In this section, we analyze the technical feasibility of implementing DIBE in the context of our running example — comparison-based sorting algorithms. We begin by noting that the control flow of the representatives from the four classes in our taxonomy — Insertion Sort, Quick Sort, Heap Sort, and Top-Down Merge Sort — is governed exclusively by the relation $\leq$ and its logical combinations using $\wedge$ (and) and $\vee$ (or)[MP69]. It is well established that neural networks are capable of computing logical operations such as $\wedge$ and $\vee$. As we will show in the following, the relation $\leq$ can likewise be approximated by feedforward neural networks in a domain-invariant manner for practical purposes.

To this end, we consider a vector $x \in \mathbb{R}^n$, and for its entries x_i and x_j with $i < j$, we aim to compute the relation $\leq$ in a pairwise manner. In the following, let π be an injective mapping of the form:

$$\pi : \{(i,j) \in \{1,...,n\} \times \{1,...,n\} | i < j\} \times \{1,...,n\} \rightarrow \binom{n}{2} \tag{7.9}$$

This mapping assigns a linear ordering to the tuples (i, j). Based on this, we define a weight matrix $W^\alpha \in \mathbb{R}^{\binom{n}{2} \times n}$ with $\alpha \in \mathbb{R}_{>0}$ as follows:

$$\forall (i,j) \in \{(i,j) \in \{1,...,n\} \times \{1,...,n\} | i < j\} : w_{\pi((i,j)),k} = \begin{cases} -\alpha & \text{if } k = i \\ +\alpha & \text{if } k = j \\ 0 & \text{else} \end{cases} \tag{7.10}$$

We consider a vector $\mathbf{x} \in \mathbb{R}^n$ and aim to compute the relation $\leq$ pairwise for its entries x_i and x_j with $i < j$. Let π be an injective mapping that assigns a linear ordering to the tuples (i, j). Based on this, we define a weight matrix $W^\alpha \in \mathbb{R}^{\binom{n}{2} \times n}$ with $\alpha \in \mathbb{R}_{>0}$ as follows.

In combination with the sigmoid function $\sigma : \mathbb{R}^{\binom{n}{2}} \rightarrow \mathbb{R}^{\binom{n}{2}}$,

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{1}^\top \mathbf{x}}}, \tag{7.11}$$

we define the mapping $L_i : \mathbb{R}^n \rightarrow \mathbb{R}^{\binom{n}{2}}$ as

$$L_i(\mathbf{x}) := \sigma(W^\alpha \mathbf{x}) \tag{7.12}$$

which satisfies the following properties for all $(i, j) \in \{(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\} \mid i < j\}$ and for arbitrary sequences $\mathbf{x}^{(l)} \in \mathbb{R}^n$:

- $\lim_{l\to\infty}(x_i^{(l)} - x_j^{(l)}) = -\infty \Rightarrow \lim_{l\to\infty} L_i(\mathbf{x}^{(l)})_{\pi(i,j)} = 1$
- $\lim_{l\to\infty}(x_i^{(l)} - x_j^{(l)}) = \infty \Rightarrow \lim_{l\to\infty} L_i(\mathbf{x}^{(l)})_{\pi(i,j)} = 0$
- $\lim_{l\to\infty}(x_i^{(l)} - x_j^{(l)}) = 0 \Rightarrow \lim_{l\to\infty} L_i(\mathbf{x}^{(l)})_{\pi(i,j)} = 0.5$

The parameter α controls the convergence behavior. To obtain a representation that is as close to binary as possible, we extend L_i by incorporating additional layers L_{i+1} and L_{i+2} as follows:

$$L_{i+1}(\mathbf{x}) := identity(\mathbf{x} + \mathbf{b}), \mathbf{b} \in \{-0.5\}^{\binom{n}{2}} \tag{7.13}$$

$$L_{i+2}(\mathbf{x}) := \sigma(\Lambda x), \Lambda_{ij} := \begin{cases} \lambda \in \mathbb{R}_{>>1} & \text{if } i = j \\ 0 & \text{else} \end{cases} \tag{7.14}$$

The composition $L_{i+2} \circ L_{i+1} \circ L_i(\mathbf{x})$ thus achieves the desired approximation behavior. In fact, the mappings L_i, L_{i+1}, and L_{i+2} can be interpreted as layers of a feedforward neural network. Since this is a composition of differentiable functions, the resulting function is itself differentiable[For17] and therefore, from a theoretical standpoint, amenable to training via gradient-based optimization methods such as gradient descent[HTF09].

7.3 Architecture and Training Procedure

The objective of the remainder of this chapter is to systematically derive an architecture and training procedure for the encoder $g : V \times M \to \{0, 1\}^m$ and decoder $f : \{0, 1\}^m \to T_G$. As an architectural basis, we employ feedforward neural networks because (1) they enable end-to-end training-based feature extraction[HTF09], (2) they can be trained efficiently via gradient descent within our end-to-end framework[HTF09], and (3) the Universal Approximation Theorem [HSW89] guarantees they have sufficient capacity to approximate the target functions. Given the continuous nature of neural networks, the primary engineering challenge is enforcing the discrete representation while preserving properties like differentiability.

7.3.1 Overview

Before delving into the technical details, we first provide an overview of the concept. We choose a neural feedforward network as our model, which is trained to approximate the mapping $classify$. The training process consists of three phases: pretraining, clustering, and fine-tuning leveraging binary regularization, emebdding clustering, and contrastive learning as underlying machine-learning techniques.

During the **pretraining** phase, the network learns to predict the correct functions of the transition algebra using labeled training data. Two loss functions are employed: the first penalizes deviations

from the correct prediction, while the second regulates the output values of the k-th layer towards the binary values 0 and 1. In the subsequent **clustering** phase, we treat the first k layers as encoder g. We apply g to all training data points and perform a clustering step to group similar embeddings. This step is grounded on the assumption that semantically similar data points are grouped within the latent feature space, a well-studied effect in literature[RPY+25]. Finally, contrastive learning[LHS20] is applied for **fine-tuning**. A contrastive loss function penalizes deviations in the representation of data points with identical clustering labels in the latent feature space. This ensures, under the assumption that semantically equivalent training data points were grouped in the latent space, that data points with the same response, even if originating from different domains, are represented identically and discretely in the latent space.

7.3.2 Architecture

Following our earlier discussion on DIBE and inspired by the autoencoder architecture[GBC16], we decompose a feed-forward neural network $N : \mathbb{R}^m \to \mathbb{R}^n$ into an encoder $g : \mathbb{R}^m \to \mathbb{R}^l$ and a decoder $f : \mathbb{R}^l \to \mathbb{R}^n$, such that:

$$N = f \circ g \tag{7.15}$$

To enforce the binary bottleneck, we choose the latent dimension l such that $l \ll m$ and $l \ll n$. Given the classification nature of our problem, the network is trained to approximate the conditional probability distribution of an operation $t \in T_G$ given an input x, i.e., $N(x) \sim P(t \mid x)$. In this context, the $softmax$ function is a natural choice for the output layer activation:

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{7.16}$$

At this point, we deliberately refrain from imposing further constraints on the remaining activation functions, as we consider these to be hyperparameters. In the subsequent section, we explain how the output of f is constrained via a loss function to enforce a binary latent space.

7.3.3 Pretraining

At the end of the pretraining phase, semantically equivalent training data points, that is, $(x_i, y_i), (x_j, y_j) \in D_j$ with $y_i = y_j$, should be spatially grouped by the encoder and projected into a latent feature space (embedding space) in a binary and compact form. The training process is guided by two loss functions. We expect such a spatial group as a consequence of the need for semantic consistency, as outlined earlier. The so-called representation loss L_{rep} penalizes values in the latent feature space that deviate from 0 and 1. In contrast, L_{class} penalizes misclassifications.

$$L_{rep}(\mathbf{x}) = \sum_{i=1}^{n} (x_i - 1)^2 x_i^2 \tag{7.17}$$

As classification loss L_{class} we choose categorical cross entropy[GBC16].

$$L_{class}(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}) \tag{7.18}$$

By summing L_{rep} and L_{class}, we obtain the overall pretraining loss function:

$$L_{pre}(\Theta, D_i) = \sum_{(x,y) \in D_i} \lambda_1 L_{rep}(x) + \lambda_2 L_{class}(y, N_\Theta(x)) \tag{7.19}$$

Note that both the representation loss function L_{rep} and the classification loss function L_{class} are differentiable, and so is their sum[For17]. A critical factor in our approach is the effective integration of training data from multiple domains $D_1, \ldots, D_j \in \mathcal{D}$.

Similar to gradient matching methods such as FISH, the MLDG algorithm enables gradient descent updates of Θ that aim to reduce the loss across all domains $D_1, \ldots, D_j$ simultaneously. We aim to leverage this effect to improve generalization, and therefore structure the pretraining phase accordingly. However, an ablation study to quantify and assess the relative impact of MLDG and DIBE on generalization performance is essential.

7.3.4 Clustering

We assume that, during pretraining, semantically equivalent data points—that is, those corresponding to identical states and predicate evaluations—are mapped to nearby regions of the latent space, even across different domains. While DIBE is theoretically grounded in the concept of a binary bottleneck, the latent representations produced after pretraining remain continuous and potentially unbounded. Despite the regularizing effect of the representation loss L_{rep}, the latent space retains the expressiveness of $\mathbb{R}$, as any interval in $[\alpha, \beta], \alpha < \beta$ can be bijectively mapped to $\mathbb{R}$[For17]. To impose a true bottleneck and promote compact, discrete - ideally binary - representations of semantically equivalent inputs, we introduce a contrastive learning phase. As a prerequisite, we perform a clustering step to identify groups of latent representations that likely correspond to the same semantics, thus enabling contrastive learning to enforce consistent encoding across domains.

Furthermore, we treat the choice of clustering algorithm as a hyperparameter. Possible options include Lloyd's algorithm[Llo82], DBSCAN[EKSX96], or other unsupervised clustering methods, depending on the structure of the latent space and the nature of the training data.

Based on thereby identified clusters $C_1, ..., C_n$ we define a labeling function as a basis for the contrastive loss in the final training phase. For this, we refer to the pretrained extractor as g_{pre}.

$$\gamma(x_1, x_2) \begin{cases} 0, & \text{if } \exists j \in \mathbb{N} : g_{pre}(x_1), g_{pre}(x_2) \in C_j \\ 1, & \text{else} \end{cases} \tag{7.20}$$

Fine-Tuning

During fine-tuning, domain-specific differences between training data points with identical labels should be eliminated. To achieve this, the training of the neural network N is continued with a modified objective function L_{fine}.

$$L_{fine}(\Theta) = \sum_{i=1}^{K} \sum_{(x,y)\in D_i} \lambda_1 L_{rep}(x) + \lambda_2 L_{class}(y, N_\Theta(x)) + \lambda_3 L_{con}(\Theta, x) \tag{7.21}$$

Note that L_{fine} still contains L_{rep} to stabilize the binary representation. In the course of evaluation, this might be dropped, as the actual values used for encoding are of minor importance. The core novelty of this phase is the contrastive loss L_{con} ideally penalizing the divergence of semantically equivalent data points in the latent feature space.

$$\begin{aligned} L_{con}(\Theta, x_1) = \sum_{i=1}^{K} \sum_{(x_2,y_2)\in D_i} & [(1-\gamma(x_1,x_2))\frac{1}{2}\|g_\Theta(x_1) - g_\Theta(x_2)\|^2 \\ & + \gamma(x_1,x_2)\frac{1}{2}\max(0, m - \|g_\Theta(x_1) - g_\Theta(x_2)\|)^2] \end{aligned} \tag{7.22}$$

Assume that the training data points x_1 and x_2 belong to the same cluster from the second phase. In this case, their Euclidean distance is incorporated into the loss. Otherwise, the loss increases according to the extent to which the minimum distance m is undershot.

7.4 Hyperparamters

Besides the clustering algorithm, there are further hyperparameters not optimized by the training algorithm and must therefore be set before the training begins. The described architecture includes the following hyperparameters:

1. $\lambda_1, \lambda_2, \lambda_3$ as weights within the loss term.
2. $m \in \mathbb{R}$ - the min distance between embeddings.

7.5 Summary

This chapter introduces Domain-Invariant Binary Embeddings (DIBE), a method designed to improve domain generalization in the context of learning predicates over computation graphs by enforcing a binary bottleneck. Motivated by the need for domain-invariant representations, DIBE leverages the structure of computation graphs and theoretical insights from binary bottlenecks and semantic consistency. The approach consists of three phases: pretraining to align semantically equivalent inputs, clustering to group latent embeddings, and fine-tuning with contrastive learning to enforce discrete, domain-agnostic representations. The architecture uses feedforward neural networks trained with customized loss functions to produce compact binary embeddings. The effectiveness of DIBE is evaluated in the next chapter, where it is compared against domain generalization baselines, including MLDG and Gradient Matching with FISH.

8 Experiment II: Learning Algorithms under Distribution Shifts

In Chapter 6, we identified two potential approaches for addressing distribution shifts in the context of learning algorithms—particularly comparison-based sorting—through gradient matching, namely FISH and MLDG. Additionally, in Chapter 7, we proposed a further domain-specific domain generalization method, DIBE. In this chapter, we empirically test the hypothesis that MLDG, FISH, and DIBE outperform vanilla neural networks trained using the Empirical Risk Minimization (ERM) principle in terms of domain generalization performance.

Section 8.1 begins by analyzing the implementation requirements and demonstrates how these are addressed using our TensorFlow-based approach. Section 8.2 outlines the experimental setup, including distributional parameters, training data sets, optimization techniques as well as used the hardware and software configuration. Finally, in Section 8.3, we evaluate the experimental results with respect to the stated hypothesis. The chapter concludes with a summary of key findings in Section 8.5.

8.1 Traning Pipeline

This section details the practical implementation of our experimental framework. Since previous implementations in earlier work[KGA24] were already built on TensorFlow[Ten25], we continue to use this framework to ensure better comparability. Followingly, we describe the specific requirements posed by MLDG, FISH, and DIBE and how these are addressed to ensure meaningful and interpretable results.

8.1.1 Requirements

Neither TensorFlow[Ten25] nor Keras[Cho+15] natively support the MLDG or FISH algorithms, requiring custom implementation. While FISH is realized using first-order gradient approximations, MLDG demands explicit computation of second-order gradients, which we implement using nested gradient tapes as provided by TensorFlow[Ten25].

As an original contribution of this work, DIBE is, by its nature, not natively supported by either TensorFlow or Keras. Furthermore, DIBE combines gradient-based training with clustering mechanisms, which are also not standard components of TensorFlow, necessitating the integration of additional external libraries. As outlined in the previous section, the encoder projects binary predicate assignments into a latent feature space, where a state-independent design is both feasible and desirable—in particular, to improve regularization and to avoid spurious correlations. In

contrast, the decoder, which is responsible for predicting the correct operation of the transition algebra—and thus the subsequent state—relies critically on explicit state information. An additional layer of technical complexity arises from the custom loss functions L_{rep} and, in particular, L_{con}. While L_{rep} operates solely within the latent feature space and can be integrated into Keras as a standard custom loss function, L_{con}, used during fine-tuning, relies on identifying the cluster associated with each training data point x based on its latent representation $g(x)$. This involves two steps: first, determining the original input $x = g^{-1} \circ g(x)$ from its latent image, and second, retrieving the corresponding preassigned cluster. Although Keras supports the implementation of custom loss functions, the framework does not, to our best knowledge, provide a mechanism for recovering the original input x from a given latent representation $g(x)$.

Another challenge arises from the need for a tight and coordinated integration of the pretraining, clustering, and fine-tuning steps, along with the requirement for repeated execution to estimate domain-specific performance metrics, such as layer-wise loss or accuracy. With respect to the training data, a multi-source domain generalization (DG) setup—as considered in this work—poses an additional challenge. Training and test datasets must be generated, managed, conditionally merged, and appropriately evaluated based on various distributional parameters. TensorFlow and Keras do not natively support this aspect either.

Taken together, these requirements—combined with the limited native support provided by TensorFlow and Keras—justify the design and implementation of a dedicated framework, which can serve both current and future experimental studies involving DIBE.

8.1.2 Design

The design shown in Figure 8.1 addresses the previously outlined requirements. The entry point is an implementation of *AbstractSearch*, which uses the *build_search_space* method to enumerate the search space as a set of *Configuration-Hyperparameter* pairs.

A concrete implementation selects tuples from this space and invokes a callback that (1) generates training, validation, and test datasets, and (2) constructs the model architecture as a sequence of layers based on the given `Configuration`. The data sampling logic corresponds to the one introduced in Chapter 5.

Each training phase—such as pretraining and fine-tuning in DIBE—is represented by a *Procedure* object. These are executed sequentially by a *Pipeline*, which also estimates metrics like accuracy by averaging results across multiple iterations.

AbstractSearch builds the pipeline by invoking two callbacks: one that returns an *InitialProcedureBuilder*, and one that provides a list of *SubsequentProcedureBuilder* instances. These, along with the constructed architecture, are passed to the *Pipeline*, which instantiates and executes each *Procedure* in order. The callbacks are overridden by the specific implementation of *AbstractSearch* to define the training sequence.

To support modified training procedures such as MLDG or FISH, a specialized subclass of *Procedure* can be implemented in which the *execute_training_step* method is overridden. This method performs a single training step and uses a callback to pass the computed *accuracy*, total loss, and layer-wise loss. The overarching training loop in *Procedure* calls this method once per epoch.

During training, the loss function receives four inputs: the output tensor produced by the layer, the complete tensor of all input samples (predictors), the indices of the samples included in the current mini-batch, and the corresponding true response values (targets). This setup allows the loss function to compute sample-wise losses accurately while also enabling more advanced use cases, such as incorporating global information, index-dependent weighting, or regularization techniques that depend on the entire input space.

8.1.3 Implementation

We continue with the description of our implementation, focusing on those aspects that had the greatest impact on the results obtained. The complete source code is linked in Section 8.2.6.

Training-Steps

We begin with the implementations of the training strategies ERM, MLDG, Gradient Matching with FISH, and DIBE. Each training strategy corresponds to a subclass of *Procedure*, which overrides the *execute_training_step* method.

```
    def _execute_training_step(self):
        loss = None
        x_data, y_data = self.merge_training_data_dict(self.training_data_dict)
        x_sample, y_sample, x_index = Procedure.create_batch(x_data, y_data, self.batch_size)
        with tf.GradientTape(persistent=False) as tape:
            layer_params = []
            current_layer = self.layer
            while current_layer is not None:
                tape.watch(current_layer.w1)
                tape.watch(current_layer.b1)
                layer_params.append((current_layer.w1, current_layer.b1))
                current_layer = current_layer.successor
            loss, prediction = self.layer.run(x_sample, x_sample, x_index, y_sample)
        dw_list = [param[0] for param in layer_params]
        db_list = [param[1] for param in layer_params]
        gradients = tape.gradient(loss, dw_list + db_list)
        self.optimizer.apply_gradients(zip(gradients, dw_list + db_list))
        layer_loss_dict = Procedure.extract_loss_per_layer(self.layer)
        accuracy = Procedure.calc_accuracy(self.layer, self.layer.predict(x_sample), y_sample)
        self._on_training_step_completed(loss, accuracy, layer_loss_dict)
```

Listing 8.1: Implementation of ERM

Listing 8.1 presents the implementation of a training step within the ERM framework. A key characteristic is the cross-domain synthesis of training data, which serves as the basis for sampling a mini-batch. For a given mini-batch, the gradient of the network is first computed using *GradientTape*, after which the weights w_i, b_i of the layers L_i are updated. The Adam optimizer[KB15] is used for this process.

```
    def _execute_training_step(self):
        x_training_data, y_training_data, x_meta_training_data, y_meta_training_date \
            = self._merge_training_data()

        x_training_sample, y_training_sample, x_training_index \
            = Procedure.create_batch(x_training_data, y_training_data, self.batch_size)
        x_meta_training_sample, y_meta_training_sample, x_meta_training_index \
            = Procedure.create_batch(x_meta_training_data, y_meta_training_date, \
            self.batch_size)

        model: Layer = self.layer
        model_cpy: Layer = self.layer_cpy
        self._sync()

        model_params = []
        model_params_cpy = []

        current_layer = model
        while current_layer is not None:
            model_params.append((current_layer.w1, current_layer.b1))
            current_layer = current_layer.successor

        current_layer = model_cpy
        while current_layer is not None:
            model_params_cpy.append((current_layer.w1, current_layer.b1))
            current_layer = current_layer.successor

        with tf.GradientTape(persistent=True) as tape:
            for var in model_params:
                tape.watch(var)
            loss, prediction \
                = model.run(x_training_sample,\
                x_training_sample, x_training_index, y_training_sample)

        dw_list = [param[0] for param in model_params]
        db_list = [param[1] for param in model_params]
        gradients = tape.gradient(loss, dw_list + db_list)

        dw_list_cpy = [param[0] for param in model_params_cpy]
        db_list_cpy = [param[1] for param in model_params_cpy]
        self.optimizer.apply_gradients(zip(gradients, dw_list_cpy + db_list_cpy))




        with tf.GradientTape(persistent=True) as meta_tape:
            for var in model_params_cpy:
                meta_tape.watch(var)
            meta_loss, _ = \
            model_cpy.run(x_meta_training_sample,\x_meta_training_sample, \

            x_meta_training_index, y_meta_training_sample)
```

```
        meta_gradients = meta_tape.gradient(meta_loss, dw_list_cpy + db_list_cpy)

        gradient_sums = []
        for index in range(len(gradients)):
            gradient_sums.append(gradients[index] + self.beta * meta_gradients[index])

        self.meta_optimizer.apply_gradients(zip(gradient_sums, dw_list + db_list))

        layer_loss_dict = Procedure.extract_loss_per_layer(self.layer)
        accuracy = Procedure.calc_accuracy(self.layer, \
            self.layer.predict(x_training_sample), y_training_sample)
        self._on_training_step_completed(loss, accuracy, layer_loss_dict)

        del tape
        del meta_tape
```

Listing 8.2: Implementation of MLDG

Since the computation of second-order gradients incurs considerable computational overhead, we implement the approximate first-order variant[SST+21]. This is reflected in the code by the absence of nested *GradientTape* contexts. Shi et al.[SST+21] report only minor performance losses in their experiments, which justifies this trade-off for an initial evaluation.

First, domain-specific meta-train and meta-test data are sampled, and corresponding mini-batches are created. The copy of the main model, previously instantiated, is synchronized with the original model's weights via a call of *sync*.

During the meta-train step, the original model is used to compute gradients with respect to the task loss. These gradients are applied to the copied model's parameters, resulting in an adapted model. In the meta-test step, a mini-batch from the meta-test data is used to compute the meta-loss on the updated model copy. Gradients of this meta-loss are then computed using *GradientTape*. A linear combination of the meta-traing-gradient and the meta-test gradient weighted by a factor β, is formed and applied to the original model.

```
    def _execute_training_step(self) -> None:
        x_data, y_data = Procedure.merge_training_data_dict(self.training_data_dict)
        x_training_sample, y_training_sample, x_training_index \
            = Procedure.create_batch(x_data, y_data, self.batch_size)
        pre_update_loss, prediction \
            = self.layer.run(x_training_sample, x_training_sample, \
                x_training_index, y_training_sample)
        pre_update_layer_loss_dict = Procedure.extract_loss_per_layer(self.layer)
        self._sync()
        model_params_cpy = []
        current_cpy_layer = self.model_cpy
        while current_cpy_layer is not None:
            model_params_cpy.append((current_cpy_layer.w1, current_cpy_layer.b1))
            current_cpy_layer = current_cpy_layer.successor
        training_data_set_names = list(self.training_data_dict)
```

```
        random.shuffle(training_data_set_names)
        for data_set_name in training_data_set_names:
            x_training_data, y_training_data = self.training_data_dict[data_set_name]
            x_training_sample, y_training_sample, x_training_index = \
                Procedure.create_batch(x_training_data, y_training_data, self.batch_size)
            with tf.GradientTape(persistent=True) as tape:
                for var in model_params_cpy:
                    tape.watch(var[0])
                    tape.watch(var[1])
                loss, prediction = self.model_cpy.run(x_training_sample, x_training_sample,\
                    x_training_index, y_training_sample)
            dw_list_cpy = [param[0] for param in model_params_cpy]
            db_list_cpy = [param[1] for param in model_params_cpy]
            gradients = tape.gradient(loss, dw_list_cpy + db_list_cpy)
            for i in range(len(dw_list_cpy)):
                dw = gradients[i]
                db = gradients[len(dw_list_cpy) + i]
                if dw is not None:
                    dw_list_cpy[i].assign_sub(self.alpha * dw)
                if db is not None:
                    db_list_cpy[i].assign_sub(self.alpha * db)
            del tape
        current_layer = self.layer
        current_cpy_layer = self.model_cpy
        while current_layer is not None:
            w1_update = current_layer.w1 + \
                tf.constant(self.beta, dtype=current_layer.w1.dtype) * \
                    (current_cpy_layer.w1 - current_layer.w1)
            b1_update = current_layer.b1 + \
                tf.constant(self.beta, dtype=current_layer.w1.dtype) * \
                    (current_cpy_layer.b1 - current_layer.b1)
            current_layer.w1.assign(w1_update)
            current_layer.b1.assign(b1_update)
            current_layer = current_layer.successor
            current_cpy_layer = current_cpy_layer.successor
        x_data, y_data = Procedure.merge_training_data_dict(self.training_data_dict)
        accuracy = Procedure.calc_accuracy(self.layer, self.layer.predict(x_data), y_data)
        self.\
            _on_training_step_completed(pre_update_loss, accuracy, pre_update_layer_loss_dict)
```

Listing 8.3: Implementation of FISH

We continue our discussion with FISH, as shown in Listing 8.3. Analogous to MLDG, a copy of the underlying model is created during the initialization of the associated *Procedure* instance. At the beginning of each training step, this copy is synchronized with the current model parameters using *sync*. Subsequently, the model copy is updated step by step and sequentially for each domain according to Algorithm 2. For this purpose, a mini-batch is sampled for each domain and a gradient descent step is performed based on it.

While MLDG was designed as a generic method for differentiable machine learning models, FISH was specifically developed and evaluated for use in neural networks. Accordingly, it can be assumed that the effectiveness of FISH is closely linked to the underlying architecture as well as the

optimization method used in the original work. Therefore, we do not use an adaptive optimization method such as Adam[KB15], but instead rely on classic stochastic gradient descent (SGD)[GBC16].

To update the actual model parameters and in line with Algorithm 2, we finally compute a weighted moving average between the previous parameters and the corresponding values of the temporarily updated model from the previous step.

Finally, we turn our attention to DIBE. The three-phase training described in Section 7.3.1 corresponds to a two-stage pipeline consisting of:

1. **Pretraining:** MLDGProcedure
2. **Fine-Tuning:** Procedure

The *Procedure* class implements ERM as shown in Listing 8.1, while *MLDGProcedure* realizes the MLDG method according to Listing 8.2. The main specific implementation effort related to DIBE lies in the loss functions and clustering. We will discuss these aspects in more detail in the following section.

Training-Loop

The previously introduced training steps are embedded into a unified training loop within the *train* method (see Listing 8.4) of the *Procedure* class. This loop iteratively executes the individual training steps and is also responsible for the evaluation and live visualization of training metrics. In the following, we focus in particular on the implementation of early stopping[GBC16], which is used as a regularization technique to prevent overfitting and to improve the model's generalization ability.

```
    def train(self) -> Layer:
        self.clear()
        best_validation_loss = None
        best_param_list = None
        no_improvement_counter = 0
        epoch_counter = 0
        for epoch in range(self.epochs):
            epoch_counter += 1
            self._execute_training_step()
            self._validate_model(epoch)
            self.epoch_arr.append(epoch)
            self._update_visualization()
            if self.early_stopping is not None:
                cur_loss = self.validation_loss_list[epoch]
                if best_validation_loss is None or cur_loss < best_validation_loss:
                    best_validation_loss = cur_loss
                    no_improvement_counter = 0
                    best_param_list = []
                    cur_layer = self.layer
                    while cur_layer is not None:
                        best_param_list.append((tf.Variable(cur_layer.w1),\
```

```
                            tf.Variable(cur_layer.b1)))
                    cur_layer = cur_layer.successor
            else:
                no_improvement_counter += 1
            if no_improvement_counter > self.early_stopping:
                cur_layer = self.layer
                for w, b in best_param_list:
                    cur_layer.w1 = w
                    cur_layer.b1 = b
                    cur_layer = cur_layer.successor
                break
    self.epochs = epoch_counter
    self._test_model()
    plt.ioff()
    return self.layer
```

Listing 8.4: Training Loop

The early stopping[GBC16] implementation is based on the observed loss. To this end, the implementation keeps track of the lowest loss measured during training. If the loss does not decrease after a training step compared to the current best value, a counter is incremented. Once this counter exceeds a predefined threshold, the training is terminated, and the last model configuration is returned.

Loss-Functions and Clustering

We now turn to the implementation of the custom loss functions required for DIBE, namely the binary representation loss L_{rep} and the clustering-based contrastive loss L_{con}. Since L_{con} depends critically on the clustering of training samples in the latent space, we also detail the specific clustering procedure employed in this context.

```
def binary_representation_loss(
        out_val: tf.Tensor,
        x: tf.Tensor,
        x_index: tf.Tensor,
        y: tf.Tensor
    ) -> tf.Tensor:
        binary_loss = tf.math.log((tf.reduce_sum(tf.square(tf.subtract(out_val, 1)) \
            * tf.square(out_val))) + tf.constant(1.0))
    return binary_loss
```

Listing 8.5: DIBE's Binary Representation Loss

Listing 8.5 shows the implementation of L_{rep} using TensorFlow. To mitigate the risk of exploding gradients, a logarithmic term was introduced. This modification necessitates the addition of a constant 1 to ensure numerical stability in cases where the inner loss approaches zero. However, due to the strict monotonicity of the logarithm, this transformation does not alter the location of the

overall minimum.

```
def create_contrastive_loss(cluster_labels: List[int], margin: float = 1.0) \\
-> Callable[[tf.Tensor, tf.Tensor, tf.Tensor], tf.Tensor]:

    cluster_labels_tf = tf.constant(cluster_labels, dtype=tf.int32)

    def contrastive_loss(
            out_val: tf.Tensor,
            x: tf.Tensor,
            x_index: tf.Tensor,
            y: tf.Tensor
        ) -> tf.Tensor:

        batch_size = tf.shape(out_val)[0]
        indices = tf.where(tf.range(batch_size)[:, None] < tf.range(batch_size))

        out_val_1 = tf.gather(out_val, indices[:, 0])  # First element of each pair
        out_val_2 = tf.gather(out_val, indices[:, 1])

        distances = tf.sqrt(tf.reduce_sum((out_val_1 - out_val_2) ** 2, axis=1)\
            + tf.constant(10**-2))

        cluster_1 = tf.gather(cluster_labels_tf, tf.gather(x_index[:, 0], indices[:, 0]))
        cluster_2 = tf.gather(cluster_labels_tf, tf.gather(x_index[:, 0], indices[:, 1]))
        labels = tf.cast(tf.equal(cluster_1, cluster_2), tf.float32)

        c_margin = tf.constant(margin, dtype=tf.float32)

        loss_similar = tf.cast(labels, tf.float32) * (distances ** 2)
        loss_dissimilar = (1 - tf.cast(labels, tf.float32)) \
            * tf.square(tf.maximum(0.0, c_margin - distances))

        loss = tf.reduce_mean(loss_similar + loss_dissimilar)
        return loss + tf.math.log((tf.reduce_sum(tf.square(tf.subtract(out_val, 1)) \
            * tf.square(out_val))) + tf.constant(1.0))

    return contrastive_loss
```

Listing 8.6: DIBE's Contrastive Loss

In Listing 8.6, the inner function *contrastive_loss* implements the contrastive loss in a fully vectorized form, enabling TensorFlow to offload computation to the GPU [Ten25]. This vectorization is crucial for scalable and efficient training. A central component in computing this loss is the assignment of cluster labels to individual data points. To facilitate this, the contrastive loss is defined as a closure that captures the cluster_labels argument, which is obtained during the preceding clustering phase of DIBE. The loss computation follows the formulation presented in Equation 7.22, which pulls together the latent representations of training data points sharing the same label, while pushing

apart representations of differently labeled points—provided their pairwise distance falls below a specified margin.

```
    def _create_subsequent_procedure_builders(
        self,
        hyperparameters: Hyperparameters,
        training_data_dict: Dict[str, Tuple[tf.Tensor, tf.Tensor]],
        validation_data_dict: Dict[str, Tuple[tf.Tensor, tf.Tensor]],
        test_data_dict: Dict[str, Tuple[tf.Tensor, tf.Tensor]]
    ) -> List[Callable[[Procedure], Procedure]]:

        def fine_tuning(procedure: Procedure):
            first_layer: Layer = procedure.layer
            last_extractor_layer: Optional[Layer] = None
            cur_layer: Layer = first_layer
            while cur_layer is not None:
                if cur_layer.loss_function is None:
                    cur_layer = cur_layer.successor
                else:
                    last_extractor_layer = cur_layer
                    break
            if last_extractor_layer is not None:
                predictor_first_layer: Layer = last_extractor_layer.successor
                last_extractor_layer.successor = None
                last_extractor_layer.loss_weight = tf.constant(1.0)
                x_data, y_data = Procedure.merge_training_data_dict(training_data_dict)
                predictions = first_layer.predict(x_data)
                inertia = []
                labels = []
                models = []
                k_range = list(range(2, 10))
                optimal_k = None
                for k in k_range:
                    kmeans = KMeans(n_clusters=k, random_state=42)
                    kmeans.fit(predictions)
                    inertia.append(kmeans.inertia_)
                    labels.append(kmeans.labels_)
                    models.append(kmeans)
                kneedle = KneeLocator(k_range, inertia, curve="convex", \
                    direction="decreasing")
                optimal_k = kneedle.elbow
                if optimal_k is None:
                    silhouette_scores = []
                    for model in models:
                        score = -1.0
                        if len(np.unique(labels)) > 1:
                            score = silhouette_score(predictions, model.labels_)
                        silhouette_scores.append(score)
                    optimal_k = k_range[silhouette_scores.index(max(silhouette_scores))]
                contrastive_loss_function: Callable[[tf.Tensor, tf.Tensor, tf.Tensor], tf.
Tensor] = create_contrastive_loss(labels[optimal_k - 1], 1.5)
```

```
                last_extractor_layer.loss_function = contrastive_loss_function
                last_extractor_layer.successor = predictor_first_layer
                return Procedure(
                    first_layer,
                    training_data_dict,
                    validation_data_dict,
                    test_data_dict,
                    epochs=hyperparameters.epochs, learning_rate=hyperparameters.alpha,
                    batch_size=hyperparameters.batch_size,
                    optimizer=hyperparameters.optimizer,
                    cache_validation_data=False
                )
        return [fine_tuning]
```

Listing 8.7: DIBE's Contrastive Loss

We now turn to the clustering phase, which is implemented using an instance of `ProcedureBuilder`. This maps the pretraining *Procedure* onto the subsequent fine-tuning *Procedure*, while enriching the pretraining results through embedding clustering, as outlined in Listing 8.7.

First, the final extractor layer is identified, and all subsequent layers are temporarily removed. The training data is then projected into the latent space and clustered using Lloyd's algorithm [Llo82]. The number of clusters is determined using the kneedle algorithm[SAIR11], implemented via the kneed library[Arv20]. If *kneed* fails to determine an appropriate number of clusters, a fallback mechanism based on the silhouette score[Rou87] is employed. The contrastive loss function is then instantiated using the identified cluster labels, with an inter-embedding margin set to 1.5. Finally, the complete model is restored, the contrastive loss function is attached, and the corresponding fine-tuning *Procedure* (implementing ERM) is instantiated.

8.2 Experimental Setup

This section details the experimental setup used to evaluate the performance of various neural architectures and domain generalization methods in the context of learning comparison sorts. We outline the procedure followed during training, including the model architectures, hyperparameter search spaces, and evaluation metrics. Additionally, we describe the generation of training data under domain shift conditions, the domain definitions based on input distributions, and the hardware and software environment in which the experiments were conducted.

8.2.1 Procedure

Analogous to our previous work [KGA24], we implement neural models to learn $classify$ for the comparison-based sorting algorithms Insertion Sort, Quick Sort, Heap Sort, and Top-Down Merge Sort — each serving as a representative of a distinct category in the previously defined classify taxonomy in Table 2.1. These models are trained using vanilla neural networks under Empirical Risk Minimization (ERM) as a baseline, as well as with the domain generalization methods MLDG, gradient matching with FISH, and the encoder-decoder-based DIBE architecture.

These approaches are evaluated in a multi-source domain generalization setting, using the training datasets described in Section 8.2.3. Regarding regularization[HTF09], training is terminated after 50 epochs without improvement in loss (early stopping [GBC16]). For hyperparameter optimization, we employ Bayesian optimization — specifically, the Tree of Parzen Estimator (TPE)[BBBK11]. The search space is explored up to 5% for DIBE and 20% for the remaining methods. This discrepancy arises from computational constraints and the large search space associated with DIBE, which comprises up to 9,216 possible candidates (versus 672 for FISH, for instance).

8.2.2 Search Space

We proceed with the specification of the search spaces explored, which are tailored according to the training method employed. To begin, we outline the general characteristics that are consistent across all methods.

General Configuration

In line with previous work[KGA24], we exclusively train feedforward neural networks. For the input layer and the remaining intermediate layers, we consider sigmoid[GBC16] and ReLU[HSM+00] as activation functions. As the task involves multi-class classification, we consistently use the softmax-function[GBC16] in the output layer to produce a probability distribution over the operations of the underlying transition algebra. Except for DIBE, the number of neurons per hidden layer is uniformly selected from $2^e, 7 \leq e \leq 10$. Categorical cross-entropy is used as a loss function that penalizes the divergence between the learned conditional probability distribution and the empirical conditional probability distribution represented by the training data [GBC16].

ERM, MLDG and FISH

Based on preliminary experiments, we observed indications of overshooting [GBC16]. Therefore, we choose for the corresponding stochastic gradient descent updates a relatively small learning rate $\alpha \in \{0.01, 0.001, 0.0001\}$. For MLDG, we select $\alpha \in \{0.01, 0.001, 0.0001\}$ and $\gamma, \beta \in 0.001, 0.0001$, with parameter names consistent with the descriptions in Section 2.4.2. For FISH, we choose $\alpha, \epsilon \in \{0.1, 0.01, 0.001, 0.0001\}$, following the parameter naming in Algorithm 2.

DIBE

For DIBE, a distinction is made between the encoder and decoder networks during initialization. The first hidden layer is allowed to have a size of 2^e, where $e \in \{7, 8, 9, 10\}$. The dimensionality of the latent feature space was selected with reference to the ideas presented in Section 7.1.1, though these were not followed strictly.

For all algorithms under consideration, the latent feature space is substantially smaller than the first hidden layer, which naturally raises the question of how to determine the size of the

intermediate layer. The i-th intermediate layer's number of neurons l is determined by the following formula:

$$l = N_0 - i \cdot \left\lceil \frac{N_0 - D}{L} \right\rceil, \tag{8.1}$$

where N_0 denotes the size of the first hidden layer, D is the latent dimensionality and L is the number of intermediate layers.

For the extractor, one or two hidden layers are permitted. In contrast, the extractor network consistently involves 32 or 64 neurons per hidden layer.

As previously discussed, the representation of predicate values in the latent feature space should be independent of the state of the underlying sorting algebra. Accordingly, this value is passed directly passed to the decoder network.

8.2.3 Training Data

Analogous to Experiment 1, we assume a distribution model for the integer sequences to be sorted. This model induces a distribution over the initial configurations within the Sorting Algebra, and thereby — in accordance with the Monte Carlo sampling described in Chapter 4.2.3 — defines a joint distribution over $(v, m) \in X \subset V \times M$ and the corresponding responses $Y \subset T_G$.

Following previous work, we leverage knowledge of the inner structure of the Sorting Algebra for manual feature engineering. From an instance (v, m), we extract the state v and, via a mapping f, derive the following components of the Sorting Algebra: the element at the top of the stack, the register file, the register file at the top of the stack register, and the elements in the array referenced by indices.

As a result, for a given input distribution, there exists a joint distribution over the predictors $f(X)$ and responses $Y \subset T_G$, from which we can sample to generate training data. To avoid a CPU bottleneck during training, we precomputed, for each distribution of inputs, the corresponding sets of predictor and response samples to bootstrap[HTF09] from (10^6 samples per domain to bootstrap from). As empirically shown in Chapter 5, changes in the parameters of the input distribution lead to a covariate shift in the predictors $f(X)$. Accordingly, each specific parameter set of the input distribution defines a domain D_i.

Table 8.1 provides an overview of the input distributions considered and the corresponding derived domains D_i. Each training, validation, or test dataset $D = D_1 \cup \ldots \cup D_n$ is composed of multiple domains D_i, each of equal size $|D_i| = 2^k$, with $k \in 13, 14$. Table 8.1 also summarizes the composition of the training, validation, and test datasets in terms of the parameters of the input distributions.

In general, the handling of domains within a training dataset is determined by the chosen optimization method. In the case of ERM minimization via gradient descent, we merge the individual domains into a unified dataset.

8.2.4 Metrics

To evaluate model performance, we compute the overall accuracy A on the complete test dataset $D = D_1 \cup \ldots \cup D_n$, where $D_1, ..., D_n$ denote the individual domains. In addition, we calculate domain-specific accuracies A_i for each domain D_i. As the results are inherently stochastic — due to both the sampling process involved in training data generation and the stochastic nature of the optimization — a single measurement is not sufficiently informative. Therefore, for each point in the search space, training is repeated five times. The expected values $\mathbb{E}(A)$ and $\mathbb{E}(A_i)$ are then estimated via averaging.

8.2.5 Evaluation Procedure

Our evaluation considers each combination of a sorting algorithm a and a domain-specific sample size $n = |D_i| \in \{8096, 16384\}$. For each such combination, we identify the model hyperparameter configuration that maximizes the expected accuracy $\mathbb{E}(A)$, and we refer to this value as A_{opt} or 'maximum cross-domain accuracy' representing the accuracy that can ideally be achieved without prior knowledge of the domain affiliation of a test data point.

The ability of a model to generalize across multiple domains is commonly measured in the literature by the minimum domain - specific accuracy. In the context of this work, this ability — and thus this metric—is of central importance. Therefore, for domains $D_1, \ldots, D_n$ and each model hyperparameter configuration, we compute the minimum $\min\{\mathbb{E}(A_1), \ldots, \mathbb{E}(A_n)\}$. We refer to the corresponding maximum of these minima, the worst-case accuracy of best generalizing model, as A_{min} or 'minimum domain-specific accuracy'.

Even if a model hyperparameter configuration achieves a high minimum domain-specific accuracy, it may still perform equally poorly across all domains — that is, operate at an accuracy level of A_{min}. To quantify the maximum deviation from the cross-domain optimum A_{opt}, we additionally compute the difference $A_{\text{opt}} - A_{\text{min}}$.

To assess the relative generalization performance of the underlying approaches - ERM, MLDG, FISH, and DIBE - for some sorting algorithm and a domain specific sample size n, we sort the trained models based on A_{min} in descending order and determine for each approach its rank as the position of its best-performing model within this ordered list.

8.2.6 Hardware and Software

This experiment was conducted on a PC equipped with an AMD Ryzen 9 5950X CPU, 64GB of RAM, and an NVIDIA GeForce RTX 3090 GPU. A summary of the software environment is provided in Table 8.2. The source code used in our experiments is publicly available at https://github.com/pkunz96/Master-Thesis-Induction-Abstraction.

Software	Version
Python	3.12
scikit-learn	1.6.1
kneedle	0.8.5
hyperopt	0.2.7
matplotlib	3.10.1
Tensorflow	2.19.0

Table 8.2: Installed Software Versions on Ubuntu 24.04.2 LTS

8.3 Results

We now turn to the evaluation of the results for the algorithms Insertion Sort, Quick Sort, Heap Sort, and Top-down Merge Sort. An analysis for Insertion Sort including tabular data is presented in this section, whereas the tabular results for the remaining algorithms are documented in Appendix B.

8.3.1 Insertion Sort

In case of Insertion Sort, with a training dataset size of 8096 samples per domain, the achieved maximum cross-domain accuracy A_{opt} is 95.57%. For a domain-specific dataset size of 16384 samples, this value drops slightly to 94.04%. We outline the respective architectures and configurations in Table 8.3.

Method	Neuron Count	Activation Functions	α	β	γ	A_{opt}
MLDG (8096)	10, 512, 512, 2	r, r, s, σ	10^{-2}	1	10^{-1}	$\approx 95.57\%$
DIBE (16384)	10, 256, 129, 3, 32, 32, 2	$r, s, s\ r, r, r, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 94.03\%$

Table 8.3: A layer-wise overview of the architecture-hyperparameter combinations that achieved the highest cross-domain accuracy A_{opt} for Insertion Sort across varying domain-specific sample sizes, $n \in \{8096, 16384\}$. The activation functions are encoded as follows: r stands for $relu$, s stands for $sigmoid$ and the softmax-function is represented by σ.

Table 8.4 provides an overview of the five best-performing architectures and hyperparameter configurations, evaluated based on the minimum domain-specific accuracy A_{min}. Assuming a domain-specific sample size of 8096, MLDG proves particularly effective. In contrast, with a sample size of 16384 samples per domain, DIBE demonstrates superior performance.

Depending on the sample size, the performance loss $A_{opt} - A_{min}$ varies. In case of 8096 samples per domain, the loss in accuracy is $\approx 3,33\%$. As the sample size grows to 16834 samples per domain, this value rises to $\approx 4,42\%$.

Table 8.5 presents the relative performance of the examined training approaches in terms of their rank. In case of a sample size of $n = 16384$, the gap between DIBE and the remaining approaches becomes particularly pronounced. On the other hand, the relatively strong performance of ERM in the case of a smaller sample size ($n = 8096$) is particularly striking.

Method	**Neuron Count**	**Activation Function**	α	β	γ	A_{min}
MLDG (8096)	10, 512, 512, 2	r, r, s, σ	10^{-2}	1	10^{-1}	$\approx 92.24\%$
MLDG (8096)	10, 512, 512, 2	r, r, s, σ	1	1	10^{-2}	$\approx 91.83\%$
ERM (8096)	10, 512, 512, 512, 2	$s, r, s\ s, \sigma$	1	-	-	$\approx 91.39\%$
MLDG (8096)	10, 1024, 1024, 1024, 2	$r, s, s\ s, \sigma$	10^{-2}	1	10^{-1}	$\approx 91.27\%$
MLDG (8096)	10, 128, 128, 128, 2	$r, r, r\ s, \sigma$	10^{-2}	1	10^{-1}	$\approx 89.64\%$
DIBE (16384)	10, 256, 129, 3, 32, 32, 2	$r, s, s\ r, r, r, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 89.61\%$
DIBE (16384)	10, 512, 257, 3, 32, 32, 2	$r, s, s\ r, r, s, \sigma$	1	1	1	$\approx 86.87\%$
DIBE (16384)	10, 256, 129, 3, 64, 64, 2	$r, r, s\ s, s, r, \sigma$	1	10^{-4}	10^{-4}	$\approx 84.42\%$
DIBE (16384)	10, 256, 129, 3, 32, 32, 2	$r, r, s\ r, r, s, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 84.17\%$
DIBE (16384)	10, 256, 129, 3, 64, 64, 2	$r, r, s\ s, s, r, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 81.84\%$

Table 8.4: A layer-wise analysis of the top five architectures and hyperparameter configurations that achieved the highest minimum domain-specific accuracy A_{min} for Insertion Sort, evaluated across two sample sizes per domain, $n \in \{8096, 16384\}$. MLDG shows superior performance at the smaller sample size, while DIBE proves more effective when the sample size is larger. The parameter names for MLDG and DIBE are consistent with Section 2.4.2, whereas the parameter names for FISH are aligned with Algorithm 2.

Method	Rank
n = 8096	
ERM	3
MLDG	1
FISH	33
DIBE	13
n = 16384	
ERM	290
MLDG	298
FISH	311
DIBE	1

Table 8.5: The table presents the rankings of domain generalization approaches - MLDG, FISH, and DIBE - along with ERM as a baseline, evaluated using Insertion Sort. A higher rank corresponds to a lower ability to generalize across varying domains, as measured by smaller values of A_{min}.

8.3.2 Quick Sort

As shown in the taxonomy in Table 2.1, the use of Quick Sort leads to a significant increase in both the dimensionality of the training data and the number of classes to distinguish. At the same time, a drop in maximum cross-domain accuracy A_{opt} to approximately $\approx 52.47\%$ for 8096 samples per domain and $\approx 52.54\%$ for 16384 samples per domain can be observed.

Notably, in the case of the larger training datasets, the ERM-based approach achieves the highest cross-domain accuracy, and the same trained model reaches the third-highest value for A_{min} (see Tables B.4 and B.5).

For a domain-specific sample size of 8096, MLDG produces the most convincing results with respect to both A_{opt} and A_{min} (see Tables B.4 and B.5). However, it should be noted that these results are based on different architectures.

Compared to earlier work [KGA24], the cross-domain accuracy achieved has dropped significantly. In contrast, no substantial decrease in domain-specific accuracy due to changes in input distributions is observed. Specifically, A_{min} remains at approximately $\approx 47.83\%$ for a domain-specific sample size of 8096, and $\approx 47.31\%$ for 16384 samples. Accordingly, the gap $A_{\text{opt}} - A_{\text{min}}$ is approximately $\approx 4.64\%$ and $\approx 5.23\%$, respectively.

8.3.3 Heap Sort

With Heap Sort, and the resulting further increase in the dimensionality of the predictors and the number of classes to distinguish, both A_{opt} and A_{min} continue to decline - regardless of the size of the underlying domain-specific dataset. For instance, a model trained using MLDG achieves a

maximum cross-domain accuracy A_{opt} of approximately $\approx 38.13\%$ for a sample size of 8096, while this value drops to approximately $\approx 30.08\%$ for a domain-specific sample size of 16384 (see Table B.7). This is particularly notable, as larger training datasets are typically associated with better generalization performance in the literature.

The tendency toward lower sensitivity to distribution shifts persists, albeit at a lower performance level. In the case of a domain-specific sample size of 8096, MLDG achieves a domain-specific accuracy A_{min} of approximately $\approx 34.44\%$, and - following the previously described trend - a correspondingly lower value of approximately $\approx 29.85\%$ for the larger sample size (see Table B.8).

In terms of the ranking of the methods, it can be observed that MLDG consistently performs best, regardless of domain-specific sample size, while DIBE exhibits the lowest performance with respect to A_{min} in both cases. FISH also outperforms the third-ranked method, ERM, in both scenarios. These findings are summarized in Table B.9.

8.3.4 Top-Down Merge Sort

Top-Down Merge Sort continues the trend toward decreasing expected model performance with increasing dimensionality of the training data. Specifically, the maximum cross-domain accuracy, depending on the domain-specific dataset size, is approximately 25.13% (8096) and 22.18% (16384), respectively (see Table B.10).

Accordingly, A_{min} also decreases to approximately 21.2% and 20.19%, respectively, such that we can report $A_{opt} - A_{min}$ as approximately 3.39% and 4.81% (see Table B.11). These values are consistently achieved by MLDG.

Analogous to Insertion Sort, the performance of DIBE, measured in terms of A_{min}, improves with the larger domain-specific sample size of 16384, such that the method achieves the second-best result in this setting. The details can be found in Table B.12.

8.3.5 Meta-Analysis

Based on our previous discussion, certain patterns have emerged that we would like to summarize here: With regard to A_{min}, MLDG consistently proves to be the most effective choice—except in the case of Insertion Sort, when using a domain-specific sample size of 16,384. In contrast, DIBE only performs well in isolated cases when trained on sufficiently large datasets. Additionally, both within individual domains and across domains, there is a noticeable decline in accuracy as the number of training samples or the number of classes to be distinguished increases.

8.3.6 Data

A comprehensive CSV file containing all architecture- and hyperparameter-specific metrics from this experiment is available at https://github.com/pkunz96/Master-Thesis-Induction-Abstraction/blob/main/Results/Experiment%20II/results_experiment_II.csv.

8.4 Ablation Study

To evaluate the contribution of individual components within the DIBE architecture, we conduct an ablation study using the Insertion Sort domain as a case study. Specifically, we assess performance resulting from the removal or modification of key elements in the architecture. The following configurations are tested:

1. No Binary Representation Loss: The binary representation loss L_{rep} is removed from the objective function.
2. No Contrastive Loss: The contrastive clustering-based loss L_{con} is disabled during fine-tuning, effectively removing the clustering mechanism from the training pipeline.
3. ERM-based Pretraining: Instead of using MLDG for the initial pretraining phase, we employ a standard ERM-based minimization strategy. The subsequent clustering and fine-tuning steps remain unchanged.

8.4.1 Results

The results of the ablation study are part of CSV file containing all architecture- and hyperparameter-specific metrics at https://github.com/pkunz96/Master-Thesis-Induction-Abstraction/blob/main/Results/Experiment%20II/results_experiment_II.csv.

8.5 Summary

The second experiment tests the hypothesis that domain generalization methods (MLDG, FISH, and DIBE) outperform standard Empirical Risk Minimization (ERM) in handling distribution shifts during sorting algorithm learning. A custom TensorFlow-based framework was implemented to accommodate each method's specific needs, including gradient approximations, clustering procedures, and custom loss functions. The models were evaluated on various sorting tasks under multi-source domain generalization, using both standard and out-of-distribution domains. Results show that MLDG consistently achieves the best worst-case domain-specific accuracy across most sorting tasks, while DIBE excels only with larger datasets. Overall, domain generalization methods—especially MLDG — validate the hypothesis by offering improved generalization under distribution shifts compared to ERM.

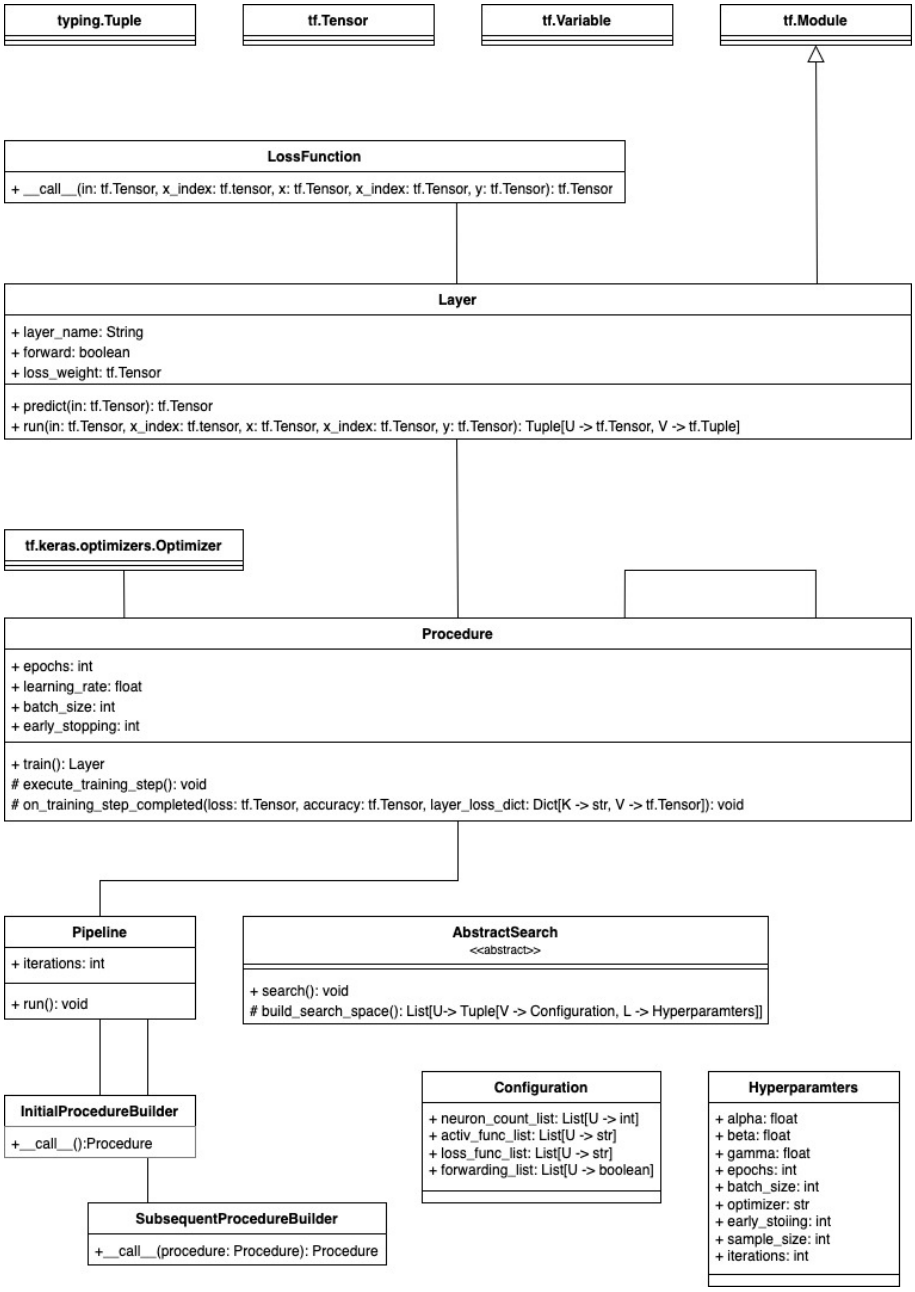

Figure 8.1: UML diagram of the custom learning framework: The structure shows the relationships between key components such as Layer, LossFunction, Procedure, Pipeline, and Configuration. It also illustrates the integration of TensorFlow classes like tf.Tensor, tf.Variable, and tf.Module.

μ_1	σ_1
Training Data	
0	100
50	100
100	100
150	100
0	50
0	150
Validation Data	
0	100
50	100
100	100
150	100
0	50
0	150
Test Data	
0	100
50	100
100	100
150	100
0	50
0	150
50	50
50	150
100	50
100	150
150	50
150	150
200	200

Table 8.1: The table shows the distribution of the parameters μ_m (mean) and σ_m (standard deviation) of the underlying distribution model following Definition 5.2.1, which, according to the considerations in Chapter 5, determines the distribution of predictors and responses. The input lengths of the integer sequences to be sorted follow a fixed distribution with an expected value of $\mu_0 = 20$ and a standard deviation of $\sigma_0 = 5$, which is not shown in the table. The training and validation domains share identical distributions, whereas the test data includes additional domains (including extreme cases and outliers).

9 Foundations of Computational Abstraction

Abstraction plays a fundamental role in managing complexity, both in everyday reasoning and in scientific analysis. One notable example is Sorting Algebra[KGA24], which facilitates sorting at a high level of abstraction—meaning with simplified and reduced terminology. Given the objective of learning algorithms from memory traces, a natural question arises: to what extent can algorithmic behavior itself be abstracted algorithmically to streamline the derivation of corresponding representations? A central aim of computation graphs over homogeneous algebras is to support the analysis of algorithms across different abstraction levels[KGA24]. Yet, the formal tools needed to precisely define and manipulate abstraction within this framework are still lacking. This section seeks to address that shortcoming by developing the necessary theoretical foundations.

In the upcoming subsection, we will develop the mathematical concepts needed to rigorously capture the abstraction of computation. The subsequent subsection shifts focus to the application of these theoretical frameworks, specifically in the context of learning algorithms, demonstrating how these ideas can be applied to the problem of learning algorithms.

9.1 Morphisms of Quotient Algebras

In academic literature, the idea of data abstraction defined as the separation between a data type's behavioral specification and its implementation[Lis87] is well-established. In the following, we develop a formalism that accounts for data abstraction and, additionally, for what refer to as control-flow abstraction. Here, control-flow abstraction refers to the idea of treating parts of a control-flow[All70] graph as semantic-preserving black-box.

Our starting point is the concept of structural abstraction, which aims to eliminate linear state transitions within control flows. Building on this, we introduce the concept of modular abstraction, which enables the encapsulation of functionally coherent segments of the control flow.

Data abstraction is addressed within this formal framework along two dimensions: (1) through the selection of operations defined by the underlying homogeneous algebra, and (2) through the implementation of abstract data types in the form of modularly encapsulated logic that combines algebraic operations to define new meta-operations. For this, in Section 9.1.1, we formally introduce structural abstraction, and based on that, we formally define modular abstraction in Section 9.1.2.

9.1.1 Structural Abstraction

Crucial for the further considerations is the transition from the previously introduced computation graphs to a more abstract perspective. For this, we will interpret a computation graph $G = (V, E, M, \phi, \gamma, s, t)$ as a function ϕ_G on a carrier set $V \times M$, so that a computation can be viewed as a sequence of function applications within the algebraic structure $(V \times M, \phi_G)$. This approach results in the following two definitions.

Definition 9.1.1
Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph. The function ϕ_G is defined as follows:

$$\phi_G : V \times M \to V \times M \tag{9.1}$$

$$\phi_G(v_1, m_1) = \begin{cases} (v_2, m_2) & \exists v_2, m_2 \in V \times M : (v_1, m_1) \vdash_G (v_2, m_2) \\ (v_1, m_1) & else \end{cases} \tag{9.2}$$

Remark: Since ϕ_G is a function $V \times M \to V \times M$ its combination with its domain or image image yields a homogeneous algebra again, as summarized in the following definition.

Definition 9.1.2
Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph. We refer to the tuple

$$(V \times M, \phi_G) \tag{9.3}$$

as the homogeneous computation algebra.

For the subsequent discussion, we aim to provide the most compact representation possible. Consider a computation graph G, where the states $v_1, \ldots, v_n$ are connected by edges (v_i, v_{i+1}), and all states, except for v_n, have at most one outgoing edge. In this scenario, the state (v_n, m_n), starting from v_1 with a given m_1, is already determined by $\vdash_G$. The possible intermediate states (v_i, m_i), however, are irrelevant to the computation. This observation motivates the followingly introduced equivalence relation $\sim_G$, which ultimately leads to the concept of a computation graph's quotient algebra.

Definition 9.1.3
Let $G = (V, E, M, \phi, \gamma, s, t)$ be a computation graph. Without loss of generality we assume:

$$\forall v \in V : deg_{in}(v) > 1 \vee deg_{out}(v) > 1 \implies \phi(v) = identity \tag{9.4}$$

The relation

$$\sim_G \subset (V \times M) \times (V \times M) \tag{9.5}$$

is defined as follows:

$(v_1, m_1) \sim_G (v_2, m_2) :\iff (v_1, m_1) = (v_2, m_2)$ *or*

1. $(v_1, m_1) \vdash_G^+ (v', m') \vdash_G^+ (v_2, m_2)$ *and* $deg_{in}(v') = deg_{out}(v') = 1$

2. *or* $(v_2, m_2) \vdash_G^+ (v', m') \vdash_G^+ (v_1, m_1)$ *and* $deg_{in}(v') = deg_{out}(v') = 1$

Theorem 9.1
$\sim_G \subset (V \times M) \times (V \times M)$ *is a equivalence relation.*

Proof. The properties of reflexivity and symmetry follow directly from the definition, while transitivity can be verified straightforwardly.

An equivalence class with respect to $\sim_G$ groups together those states $z \in V \times M$ that deterministically transition into one another during a computation and are thus equivalent in this sense. To achieve a possibly compact representation of a computation, we adapt ϕ_G to act on the quotient set:

(9.6) $V \times M \backslash \sim_G$

As outlined in the following steps, this representation eliminates specific intermediate states, thereby reducing the complexity of the computation while preserving its essential properties.

Definition 9.1.4
Let $G = (V, E, M, \phi, \gamma, s, t)$ *be a computation graph and* $\sim_G$ *its transition relation. The function*

(9.7) $\Phi_G : V \times M \backslash \sim_G \rightarrow V \times M \backslash \sim_G$

defined as follows:

$$\Phi_G(x) = \begin{cases} [y]_{\sim_G} & \text{if } [x]_{\sim_G} \neq [y]_{\sim_G} \exists t_1 \in [x]_{\sim_G} \exists t_2 \in [y]_{\sim_G} : \phi_G(t_1) = t_2 \\ [x]_{\sim_G} & \text{else} \end{cases} \tag{9.8}$$

With the following definition, we combine $(V \times M \backslash \sim_G)$ and Φ_G into a homogeneous algebra, using the set of equivalence classes with respect to $\sim_G$ as the carrier set.

Definition 9.1.5
Let $G = (V, E, M, \phi, \gamma, s, t)$ *be a computation graph. We refer to*

(9.9) $(V \times M \backslash \sim_G, \Phi_G)$

as the homogeneous quotient algebra Q of the computation graph G.

However, due to the definition of Φ_G being based on representatives $y \in [x]_{\sim_G}$, it is essential to prove that this definition is independent of the choice of specific representatives. Formally speaking, we have to show that Φ_G is well-defined. This property is verfied in corollary 9.2.

Corollary 9.2

(9.10) $y \in [x]_{\sim_G} \implies \Phi_G([y]_{\sim_G}) = \Phi_G([x]_{\sim_G})$

Proof. $x \sim_G y \iff [x]_{\sim_G} = [y]_{\sim_G} \implies \Phi_G([y]_{\sim_G}) = \Phi_G([x]_{\sim_G})$

The application of the function Φ_G corresponds to a single computation step. However, for further considerations, it is technically beneficial to be able to refer to the result after n steps. To this end, we introduce the function Φ_G^n, which assigns an element of the quotient algebra the result of the $n-th$ computation step.

Definition 9.1.6

$$\Phi_G^n(x) = \begin{cases} \Phi_G \circ \Phi_G^{n-1}(x) \text{ if } n \geq 1 \\ x \text{ else} \end{cases} \tag{9.11}$$

Previously, we have stated that a quotient algebra Q preserves the essential properties of the computation. Formally, this reflects particularly as follows:

1. $\forall m_1, m_2 : m_1 \neq m_2 \implies$

$$[(s, m_1)]_{\sim_G} \neq [(s, m_2)]_{\sim_G} \wedge [(t, m_1)]_{\sim_G} \neq [(t, m_2)]_{\sim_G}$$

In this context, the fact that distinct initial and terminal states correspond to different equivalence classes under $\sim_G$ is a crucial foundation for the following theorem. This assertion can be readily verified by applying the definition of $\sim_G$. However, the theorem goes further by demonstrating that Φ_G^n not only respects this distinction but also translates the computational semantics of $\vdash_G^+$ for both the initial and terminal states. This ensures that the semantic integrity of the computation is preserved throughout the mapping.

Theorem 9.3
$(s, m_1) \vdash_G (t, m_n) \iff \exists n \in N : \Phi_G^n([(s, m_1)]_{\sim_G}) = [(t, m_n)]_{\sim_G}$

Proof. The theorem follows by induction.

In summary, the quotient algebra can be regarded as a condensed representation of all essential computational steps. Building on this foundation, we introduce the formal framework for rigorously defining abstraction between computation graphs. An abstraction G_2 of a computation graph G_1 should compute the same function and preserve the control flow of G_1. This concept can be formally captured through the isomorphism of their quotient algebras.

Definition 9.1.7
Let G_1 and G_2 be computation graphs. We consider their quotient algebra

$$Q_1 := (V_1 \times M_1, \Phi_{G_1}) \tag{9.12}$$

$$Q_2 := (V_2 \times M_2, \Phi_{G_2}) \tag{9.13}$$

isomorphic if there is bijective mapping

$$f : V_1 \times M_1 \backslash \sim_{G_1} \rightarrow V_2 \times M_2 \backslash \sim_{G_2} \tag{9.14}$$

such that the following holds true:

$$f \circ \Phi_{G_1}(x) = \Phi_{G_2} \circ f(x) \tag{9.15}$$

The isomorphism is denoted in abbreviated form as $f : Q_1 \to Q_2$. When a quotient algebra Q_1 transfers its semantic properties to another quotient algebra Q_2 via an isomorphism f, and the computations in this context are considered equivalent, this equivalence should also manifest formally, including in the reverse direction. In general, the inverse mapping of an isomorphism is also an isomorphism. This property is demonstrated for the quotient algebras under consideration in the following corollary.

Corollary 9.4
$f : Q_1 \to Q_2$ is an isomorphism. $\iff$ *f^{-1} is an isomorphism.*

Proof. f^{-1} is an isomorphism.

$\iff f^{-1} \circ \Phi_2(y) = \Phi_1 \circ f^{-1}(y)$

We define $y := f(x)$, so that the following holds true:

$\iff f^{-1} \circ \Phi_{G_2} \circ f(x) = \Phi_1(x)$

$\iff \Phi_{G_2} \circ f(x) = f \circ \Phi_1(x)$

$\iff f$ is an isomorphism.

Building on this foundation, we can now investigate how an isomorphism translates a computation with respect to a quotient algebra Q_1 into a computation with respect to Q_2.

Assuming semantic equivalence, one would intuitively expect no difference between executing n steps in Q_1 before transitioning to Q_2 and transferring the initial configuration directly to Q_2 before performing the steps. The following corollary formalizes this intuition.

Corollary 9.5
$f : Q_1 \to Q_2$ is an isomorphism. $\implies f \circ \Phi^n_{G_1}(x) = \Phi^n_{G_2} \circ f(x)$

Proof. We assume $f \circ \Phi^n_{G_1}(x) = \Phi^n_{G_2}(x) \circ f$ for a fixed $n \in \mathcal{N}$. Under this assumption follows:

$f \circ \Phi^{n+1}_{G_1}(x)$

$= f \circ \Phi_{G_1} \circ \Phi^n_{G_1}(x)$

$= \Phi_{G_2} \circ f \circ \Phi^n_{G_1}(x)$

$= \Phi_{G_2} \circ \Phi^n_{G_2} \circ f(x) = \Phi^{n+1}_{G_2} \circ f(x)$

We now focus our attention on initial and terminal configurations. To this end, we first introduce the concept of a fixed point $[x]_{\sim_G}$, which is an equivalence class that cannot be left once reached by a computation, or formally:

(9.16) $\forall n \in \mathcal{N} : \Phi_G^n([x]_{\sim_G}) = [x]_{\sim_G}$

In this sense, a fixed point marks the termination of a computation.

Definition 9.1.8
Let $Q := (V \times M \backslash \sim_G, \Phi_G)$ be a quotient algebra. We denote

(9.17) $x \in V \times M \backslash \sim_G$

a fixed-point if and only if the following holds true:

(9.18) $\Phi_G(x) = x$

If an isomorphism is intended to express the equivalence of two computations, it must map fixpoints onto fixpoints, leading to the next corollary.

Theorem 9.6
Let Q_1 and Q_2 be isomorphic quotient algebras, and let

(9.19) $f : Q_1 \to Q_2$

be the corresponding isomorphism. Then the following holds:

(9.20) $\forall x \in V_1 \times M_1 : \Phi_{G_1}(x) = x \iff \Phi_{G_2}(f(x)) = f(x)$

Proof. $x = \Phi_{G_1}(x)$

$\iff f^{-1}(f(x)) = \Phi_{G_1}(f^{-1}(f(x))) = f^{-1}(\Phi_{G_2}(f(x)))$

$\iff \Phi_{G_2}(f(x)) = f(x)$

Analogously, we require $f : Q_1 \to Q_2$ to map initial configurations with respect to Q_1 - configurations that have no predecessor - onto initial configurations with respect Q_2. The following theorem investigates to what extent this is the case.

Theorem 9.7
Let Q_1 and Q_2 be isomorphic quotient algebras, and let

(9.21) $f : Q_1 \to Q_2$

be the corresponding isomorphism. Then the following holds:

$(\forall x, y \in V_1 \times M_1 : [x]_{\sim_{G_1}} \neq [y]_{\sim_{G_1}} \implies \Phi_{G_1}([x]_{\sim_{G_1}}) \neq [y]_{\sim_{G_1}})$

$\implies (\forall x \in V_2 \times M_2 : [x]_{\sim_{G_2}} \neq f([y]_{\sim_{G_1}}) \implies \Phi_{G_2}([x]_{\sim_{G_2}}) \neq f([y]_{\sim_{G_1}}))$

Proof. Suppose $\exists x \in V_2 \times M_2 : [x]_{\sim_{G_2}} \neq f([y]_{\sim_{G_2}}) \wedge \Phi_{G_2}([x]_{\sim_{G_2}}) = f([y]_{\sim_{G_1}}))$.

$\implies f^{-1} \circ \Phi_{G_2}([x]_{\sim_{G_2}}) = \Phi_{G_1} \circ f^{-1}([x]_{\sim_{G_2}}) = f^{-1} \circ f([y]_{\sim_{G_1}}) = [y]_{\sim_{G_1}}$

and $f^{-1}([x]_{\sim_{G_2}}) \neq [y]_{\sim_{G_1}}$

$\implies \Phi_{G_1}(f^{-1}([x]_{\sim_{G_2}})) = [y]_{\sim_{G_1}}$ and $f^{-1}([x]_{\sim_{G_2}}) \neq [y]_{\sim_{G_1}}$

$\implies (\exists x, y \in V_1 \times M_1 : [x]_{\sim_{G_1}} \neq [y]_{\sim_{G_1}} \wedge \Phi_{G_1}([x]_{\sim_{G_1}}) = [y]_{\sim_{G_1}})$

Remark 1: In this theorem, an initial configuration is one without a predecessor. Formally, this intuition is captured by the condition:

(9.22) $(\forall x, y \in V_1 \times M_1 : [x]_{\sim_{G_1}} \neq [y]_{\sim_{G_1}} \implies \Phi_{G_1}([x]_{\sim_{G_1}}) \neq [y]_{\sim_{G_1}})$

Remark 2: Although f maps fixed points to fixed points and initial configurations to initial configurations, it is possible for a fixed point $[x]_{\sim_{G_1}}$ to be mapped to a fixed point .

(9.23) $f([x]_{\sim_{G_1}}) = [v, s]_{\sim_{G_2}}, v \neq t_2,$

meaning that $[v, s]_{\sim_{G_2}}$ is not a terminal state in the sense of the underlying computation graph G_2. Similarly, an element $[(v, m)]_{\sim_{G_1}}$ with $v = s_1$ could be mapped to a non-initial configuration:

(9.24) $f([(v, m)]_{\sim_{G_1}}) = [(v', m)]_{\sim_{G_2}}, v \neq s_2$

To account for this, we adjust the definition of an isomorphism between quotient algebras as follows:

Definition 9.1.9
Let G_1 and G_2 be computation graphs. We consider their quotient algebras

(9.25) $Q_1 := (V_1 \times M_1, \Phi_{G_1})$

and

(9.26) $Q_2 := (V_2 \times M_2, \Phi_{G_2})$

isomorphic, written $Q_1 \cong Q_2$, if there is bijective mapping

(9.27) $f : V_1 \times M_1 \backslash \sim_{G_1} \rightarrow V_2 \times M_2 \backslash \sim_{G_2}$

such that the following holds true:

1. $f \circ \Phi_{G_1}(x) = \Phi_{G_2} \circ f(x)$
2. $f([(s_1, m)]_{\sim_{G_1}}) = [(s_2, m')]_{\sim_{G_2}}$
3. $f([(t_1, m)]_{\sim_{G_1}}) = [(t_2, m')]_{\sim_{G_2}}$

The following theorem consolidates the preceding results by showing the equivalence of computations within their respective computation graphs and the corresponding quotient algebras, Q_1 and Q_2, when the quotient algebras are isomorphic.

Corollary 9.8
Let G_1 and G_2 be computation graphs, and let $f : Q_1 \to Q_2$ be an isomorphism. Then the following holds:

$$(s_1, m_1) \vdash^{+}_{G_1} (t_1, m_2)$$

$$\iff \exists n \in \mathcal{N} : \Phi^n_{G_1}([(s_1, m_1)]_{\sim_{G_1}}) = [(t_1, m_2)]_{\sim_{G_1}}$$

$$\iff \exists n \in \mathcal{N} : f \circ \Phi^n_{G_1}([(s_1, m_1)]_{\sim_{G_1}}) = f([(t_1, m_2)]_{\sim_{G_1}})$$

$$\iff \exists n \in \mathcal{N} : \Phi^n_{G_2} \circ f([(s_1, m_1)]_{\sim_{G_1}}) = f([(t_1, m_2)]_{\sim_{G_1}})$$

$$\iff (s_2, m_3) \vdash_{G_2} (t_3, m_4), \textit{ where}$$

$$(s_2, m_3) \in f([(s_1, m_1)]_{\sim_{G_1}}) \wedge (t_3, m_4) \in f([(t_1, m_2)]_{\sim_{G_1}})$$

After these preparatory steps, we now proceed to the core of this chapter: In the following, we will define mathematically and rigorously what we mean by the abstraction of a computational process.

Definition 9.1.10
Let G_1 and G_2 be computation graphs. We denote G_2 a structural abstraction of G_1, written $G_2 \lhd_s G_1$, if and only if:

1. $Q_1 \cong Q_2$,
2. *and* $|V_1| < |V_2|$

In this context, a computation graph $G_2$is considered an abstraction of another computation graph G_1 if it preserves the control flow of G_1 while representing the computation through a more compact homogeneous algebra, with fewer functions and states. Unlike approaches that define the equivalence of a computation solely based on its inputs and outputs, this approach also retains key characteristics of the process through which these results are derived.

Theorem 9.9
Structural abstraction is not decidable.

Proof. Let G be the set of all computation graphs. Then it holds:

$$A_s := \{(G_1, G_2) \in G^2 | G_2 \lhd_s G_1\} \tag{9.28}$$

is not decidable. We proof the theorem by reducing the halting problem on empty set on $H_0 \leq A_s$.

9.1.2 Modular Abstraction

Structural abstraction does not allow individual sections of the control flow to be treated as a black box. In practical applications, this limitation means that the existing formalism, for example, does not support the systematic analysis of function calls. To overcome this limitation, we introduce the concept of modular abstraction.

The concept of abstraction in the context of algorithms can refer to both control flow and data representation. When analyzing computation graphs and their corresponding quotient algebras, control flow manifests as transitions between elements of the quotient set. However, these elements do not represent control flow in isolation. Instead, they link it to the structures of the underlying homogeneous algebra. To enable a purely algebraic analysis of control flow, we associate each element of the quotient algebra with a representative state from the underlying computation graph. Technically, this is achieved by introducing an equivalence relation on the quotient set, defined as follows:

Definition 9.1.11
Let $Q := (V \times M \setminus \sim_G, \Phi_G)$ *be a quotient algebra. The relation*

(9.29) $\sim_Q \subset (V \times M \setminus \sim_G) \times (V \times M \setminus \sim_G)$

is defined as:

(9.30) $\forall x, y \in (V \times M \setminus \sim_G) : \quad x \sim_Q y \iff \exists (v_1, m_1) \in x, ; (v_2, m_2) \in y : v_1 = v_2$

Proof. The properties of reflexivity, symmetry, and transitivity can be verified straightforwardly.

To motivate the abstraction concept introduced below, we first consider it from the perspective of computation graphs and then derive a corresponding algebraic definition. Let G_1 and G_2 be computation graphs. Intuitively, we can regard G_2 as an abstraction of G_1 if the following conditions hold: (1) For equivalent inputs (up to renaming), both G_1 and G_2 produce equivalent outputs (also up to renaming). (2) The states in G_2 serve as meta-states that represent coherent subcomputations within G_1. Formally, this corresponds to an injective mapping from nodes in G_1 to nodes in G_2. (3) Each direct transition between two nodes in G_2 corresponds to a path in G_1 between the respective mapped nodes. To ensure modular closure, none of the intermediate nodes on this path (except for the start and end nodes) may themselves be mapped to any node in G_2. Followingly, we turn to to the formalization of these considerations using algebraic methods.

Definition 9.1.12
Let G_1 *and* $G_2 = G = (V, E, M, \phi, \gamma, s, t)$ *computation graphs, where*

(9.31) $G_1 = (V_1, E_1, M_1, \phi_1, \gamma_1, s_1, t_1)$,

(9.32) $G_2 = (V_2, E_2, M_2, \phi_2, \gamma_2, s_2, t_2)$

with quotient algebras

(9.33) $Q_1 := (V_1 \times M_1, \Phi_{G_1})$,

(9.34) $Q_2 := (V_2 \times M_2, \Phi_{G_2})$.

We denote G_2 a modular abstraction of G_1, written

(9.35) $G_2 \lhd_s G_1$,

if there is an injective mapping

(9.36) $f : V_2 \times M_2 \backslash \sim_{G_2} \rightarrow V_1 \times M_1 \backslash \sim_{G_1}$

such that: $\forall x, y \in V \times M \exists n \in \mathbb{N} \forall 1 \leq n_0 < n$:

1. $f \circ \Phi_2([x]_{\sim_{G_2}}) = \Phi_1^n \circ f([x]_{\sim_{G_2}})$
2. $[f([y]_{\sim_{G_2}})]_{\sim_{Q_1}} \neq [\Phi_1^{n_0} \circ f([x]_{\sim_{G_2}})]_{\sim_{Q_1}}$
3. $[[x]_{\sim_{G_2}}]_{\sim_{Q_2}} = [[y]_{\sim_{G_2}}]_{\sim_{Q_2}} \implies [f([x]_{\sim_{G_2}})]_{\sim_{Q_1}} = [f([y]_{\sim_{G_2}})]_{\sim_{Q_1}}$

By requiring the mapping f to be injective, we preserve the semantics of the computations directly as shown by the following equation:

(9.37) $f \circ \Phi_2([x]_{\sim_{G_2}}) = \Phi_1^n \circ f([x]_{\sim_{G_2}}) \implies \Phi_2([x]_{\sim_{G_2}}) = f^{-1} \circ \Phi_1^n \circ f([x]_{\sim_{G_2}})$

Condition (1) ensures that every computation in the abstract quotient algebra Q_2 can be simulated by finitely many steps in the concrete quotient algebra Q_1. Condition (2) ensures modular closure, as it prevents any sub-computation by G_1 from reverting to shared meta-states with G_2. Whenever a computation step in the abstract graph G_2 involves distinct states v_1, v_2 that also appear in G_1, their corresponding images under f must preserve this distinction. This is guaranteed by Condition (3). However, it remains an open question whether consistency of state transitions between states in G_2 and their corresponding images in G_1 is ensured solely as a consequence of Conditions (1)–(3). The following theorem answers this question in the affirmative.

Theorem 9.10
Let $G_2 \lhd_m G_1$ be a modular abstraction with corresponding quotient algebras

(9.38) $Q_1 := (V_1 \times M_1, \Phi_1), Q_2 := (V_2 \times M_2, \Phi_2)$.

Suppose:

1. $[[x]_{\sim_{G_2}}]_{\sim_{Q_2}} = [[y]_{\sim_{G_2}}]_{\sim_{Q_2}}$
2. $[\Phi_{G_2}([x]_{\sim_{G_2}})]_{\sim_{Q_2}} = [\Phi_{G_2}([y]_{\sim_{G_2}})]_{\sim_{Q_2}}$
3. $f \circ \Phi_2([x]_{\sim_{G_2}}) = \Phi_1^{n_1} \circ f([x]_{\sim_{G_2}})$
4. $f \circ \Phi_2([y]_{\sim_{G_2}}) = \Phi_1^{n_2} \circ f([y]_{\sim_{G_2}})$

Then it follows that:

$$[\Phi_1^{n_1} \circ f([x]_{\sim_{G_2}})]_{\sim_{Q_1}} = [\Phi_1^{n_2} \circ f([y]_{\sim_{G_2}})]_{\sim_{Q_1}}$$

Proof. $[\Phi_2([x]_{\sim_{G_2}})]_{\sim_{Q_2}} = [\Phi_2([y]_{\sim_{G_2}})]_{\sim_{Q_2}}$

$\implies [f \circ \Phi_2([x]_{\sim_{G_2}})]_{\sim_{Q_1}} = [f \circ \Phi_2([y]_{\sim_{G_2}})]_{\sim_{Q_1}}$

$\iff [\Phi_1^{n_1} \circ f([x]_{\sim_{G_2}})]_{\sim_{Q_1}} = [\Phi_1^{n_2} \circ f([y]_{\sim_{G_2}})]_{\sim_{Q_1}}$

Building upon the previous results, we now investigate whether a complete computation in the abstract structure G_2 can be simulated through corresponding computation steps in the concrete structure G_1. For the correct simulation of complete computations of G_2 by G_1, it is essential that the semantic structure is preserved consistently across all computation steps. In particular, even if multiple computation steps are performed in G_2, the corresponding simulation in G_1 must faithfully preserve the semantics of each individual step in sequence. The following theorem confirms that this property is indeed ensured in the context of modular abstraction.

Theorem 9.11
Let $G_2 \lhd_m G_1$ be a modular abstraction with corresponding quotient algebras

(9.39) $Q_1 := (V_1 \times M_1, \Phi_1), Q_2 := (V_2 \times M_2, \Phi_2),$

and let

(9.40) $f : V_2 \times M_2/{\sim_{G_2}} \rightarrow V_1 \times M_1/{\sim_{G_1}}$

be the associated injective mapping. Then it follows that:

(9.41) $\forall x \in V_2 \times M_2/{\sim_{G_2}},\ \forall m \in \mathbb{N},\ \exists n \in \mathbb{N} : \quad f \circ \Phi_2^m(x) = \Phi_1^n \circ f(x).$

Proof. We proceed by induction on $m \in \mathbb{N}$. By the definition of modular abstraction, there is some $n_1 \in \mathbb{N}$ such that the theorem holds true for $(m = 1)$:

(9.42) $f \circ \Phi_2(x) = \Phi_1^{n_1} \circ f(x)$

In the following, we assume inductively $n_m \in \mathbb{N}$ such that the following holds true for some $m \in \mathbb{N}$:

(9.43) $f \circ \Phi_2^m(x) = \Phi_1^{n_m} \circ f(x)$

We proceed under this assumption and prove the statement to be true for $m + 1$:

$$\begin{aligned} f \circ \Phi_2^{m+1}(x) &= f \circ \Phi_2 \circ \Phi_2^m(x) \\ &\overset{IH}{=} f \circ \Phi_2 \circ f^{-1} \circ \Phi_1^{n_m} \circ f(x) \\ &\overset{BC}{=} \Phi_1^{n_1} \circ f(x) \circ f^{-1} \circ \Phi_1^{n_m} \circ f(x) \\ &= \Phi_1^{n_1} \circ \Phi_1^{n_m} \circ f(x) \\ &= \Phi_1^{n_1+n_m} \circ f(x), \end{aligned}$$

where IH refers to the inductive hypothesis and BC to the base case.

It remains to be shown that modular abstraction also preserves fixed points, or vividly speaking, preserves the termination properties of G_2. The following corollary establishes this property.

Corollary 9.12
Let $G_2 \lhd_m G_1$ be a modular abstraction with quotient algebras

(9.44) $Q_1 := (V_1 \times M_1, \Phi_1), Q_2 := (V_2 \times M_2, \Phi_2)$

Then it follows:

$$\Phi_2(x) = x \iff \forall n \in \mathbb{N} \exists n_0 \geq n : \Phi_1^{n_0} \circ f(x) = f(x)$$

Proof. $\Phi_2(x) = x$.

$$\iff f(x) = f \circ \Phi_2(x) \wedge \exists n \in \mathbb{N} : f \circ \Phi_2(x) = \Phi_1^n \circ f(x)$$

$$\iff \exists n \in \mathbb{N} : f(x) = f \circ \Phi_2(x) = \Phi_1^n \circ f(x)$$

The central result of this chapter is the undecidability of modular abstraction. At its core, modular abstraction addresses the question of which parts of an algorithm are explicitly observable—because they result from the application of functions in a homogeneous algebra—and which parts are implicitly encoded within those functions. As a consequence of the undecidability result presented below, it is algorithmically impossible to determine whether two procedures share the same implicit structure (encoded in functions of the homogeneous algebra), the same explicit structure (represented by their computation graphs), and the same semantics. In general, algorithms are not capable of performing this kind of abstraction.

Theorem 9.13
Let C be the set of all computation graphs. Then it holds:

(9.45) $A_m := \{(G_1, G_2) \in C^2 | G_2 \lhd_m G_1\}$

is not decidable.

Proof. We proof the theorem by showing that $H_0 \leq A_m$ holds. For this, let

(9.46) $G_2 = (V_2, E_2, M_2, \phi_2, \gamma_2, s_2, t_2)$,

be an abstract computation graph computing the identity function $id : \mathbb{N} \to \mathbb{N}$, where

1. $V_2 := \{s_2, s_1, t_1, t_2\}$,
2. $E_2 := \{(s_2, s_1), (s_1, t_1), (t_1, t_2)\}$,
3. $M_2 := \mathbb{N}$,
4. $\phi_2(v) = identity$,
5. $\gamma_2(e)(n) = 1$.

Let w be the encoding of an arbitrary Turing machine M. In our previous work[KGA24], we showed that computation graphs over the algebra $(\mathbb{N}, (f_i)_{i \in I})$ are Turing complete. Therefore, for every Turing machine M, we can assume the existence of a computation graph G_1 with the same state graph that operates over the given algebra:

(9.47) $G_1 = (V_1, E_1, M_1, \phi_1, \gamma_1, s_1, t_1)$, where

1. $V_1 \cap V_2 = \{s_1, t_1\}$
2. $M_2 := \mathbb{N}$

We construct a computation graph $G_3 = (V_3, E_3, M_3, \phi_3, \gamma_3, s_1, t_1)$ that simulates both G_2 and G_1 simultaneously. For $w \in H_0$, the graph computes the identity function, such that G_2 represents an abstraction of G_3 and for $w \notin H_0$, it computes the everywhere undefined function.

To this end, let $c : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be a computable function that is injective in both its first and second argument, with inverses satisfying $c^{-1}(c(x, y)) = x$ and $c^{-2}(c(x, y)) = y$.

The remaining components of G_3 are defined as follows:

1. $V_3 := V_1 \cup V_2$
2. $E_3 := (E_1 \cup E_2) \setminus \{(s_1, t_1)\}$
3. $M_3 := \mathbb{N}$
4. $\gamma_3(v_1, v_2) := \begin{cases} p_1(n) := 1, v_1 = s_2 \\ p_2(n) := 1, v_1 = t_1 \\ p_2(n) := \gamma_2(v_1, v_2)(c^{-2}(n)), else \end{cases}$
5. $\phi_3(v) := \begin{cases} f_1(n) := c(n, 0), v = s_2 \\ f_2(n) := c^{-1}(n), v = t_1 \\ f_2(n) := c(c^{-1}(n), \phi_2(v)(c^{-2}(n))), else \end{cases}$

In a computable manner, we have constructed a computation graph G_3 from G_1 such that the following holds:

$$(9.48)\quad (G_3, G_1) \in A_m \iff G_1 \lhd_m G_3 \iff w \in H_0$$

If A_m were decidable, then H_0 would also be decidable. However, this leads to a contradiction, since H_0 is known to be undecidable[Tur36]. Consequently, A_m, and therefore modular abstraction, is undecidable.

The attentive reader will have noticed that a predicate actually expects an element of the property space, rather than an element of the homogeneous algebra. However, it is clear that the properties of G_1 can be appropriately integrated into G_2, so that this is merely a technical detail, which would have complicated the notation.

9.2 Three-Level-Architecture

In the previous chapter, we discussed the concept of algorithmic abstraction in detail and clarified it through the notions of structural and modular abstraction. It became clear that the algorithmic search for suitable abstractions is limited not only by the vastness of the search space but also by the fundamental undecidability of modular abstraction.

When learning algorithms from training data, we face an additional challenge: the data typically embodies a specific level of abstraction, which may not align well with the requirements of machine learning methods.

Take, for instance, a linear sequence of states $v_1, \ldots, v_n$, each associated with functions drawn from a homogeneous algebra. Rather than applying these functions sequentially, a well-designed ML model should be capable of capturing their composition directly—reflecting the corresponding quotient structure.

A more complex example is Sorting Algebra[KGA24]. On closer inspection, it reveals non-trivial structural properties that encapsulate a considerable amount of complexity, thereby facilitating the learning of sorting procedures—primarily because it operates on a higher level of abstraction. In practical applications, however, the available training data is often situated at a significantly lower abstraction level.

This leads us to the central question: To what extent is it possible—despite the undecidability of modular abstraction—to reconstruct higher-level representations from lower-level training data, making them more suitable for ML-based processing? To explore this further, we introduce the notion of a three-level architecture, which serves as a conceptual framework for analyzing how structural abstractions might be approximated through statistical methods.

9.2.1 Definition

The three-level architecture is designed to formally cover three layers of analysis in a computational process. The lowest level defines the universe of discourse by specifying a basic carrier set M along with fundamental operations on M. The higher levels also operate on this same carrier set, but they rely on function compositions of the elementary algebraic operations, which motivates the following definition of coverage. This enables a more abstract and structured perspective on the computational process at higher levels. To illustrate: At the lowest level, we deal with bitwise manipulations, these correspond to CPU instructions at the intermediate level and finally, at the top level, we reach a high-level perspective, such as that of high-level programming languages.

Definition 9.2.1
Let M_1 and M_2 be homogeneous algebras over a common carrier set M, defined as:

(9.49) $M_1 := (M, (f_i)_{i \in I_1})$

(9.50) $M_2 := (M, (f_i)_{i \in I_2})$

We denote M_1 is a cover of M_2, written

(9.51) $M_1 \models M_2$,

if and only if:

$$\forall i \in I_2 \; \exists n \in \mathbb{N} \; \exists S \subseteq I_1 \; \exists (i_1, \ldots, i_n) \in S^n : \quad f_i = f_{i_1} \circ \cdots \circ f_{i_n}$$

The following corollary ensures the operational cohesion between the layers. It formally captures the translatability of high-level operations into low-level operations within the framework of the following definition.

Corollary 9.14
Let M_1, M_2, M_3 be homogeneous algebras over a common carrier set M. Then it holds:

(9.52) $M_1 \vDash M_2 \wedge M_2 \vDash M_3 \implies M_1 \vDash M_3$

Proof. $M_2 \vDash M_3 \wedge f_{i,i\in I_3}$

$\implies \exists j_1, ..., j_n \in I_2 : f_i = f_{j_n} \circ ... \circ f_{j_1}$

Let $f_{j_k}, j_k \in I_2$ and $M_1 \vDash M_2$.

$\implies \exists l_1, ..., l_m \in I_1 : f_{j_k} = f_{l_m} \circ ... \circ f_{l_1}$

$\implies \exists t_1, ..., t_s \in I_1 : f_i = f_{t_s} \circ ... \circ f_{t_1}$

$\implies M_1 \vDash M_3$

This lays the formal foundation for defining the three-level architecture, which models computations over a fixed carrier set with gradually increasing levels of structural abstraction.

Definition 9.2.2
Let G_1, G_2, G_3 be computation graphs defined over the algebras

(9.53) $M_j := (M, (f_i)_{i\in I_j}), \quad j \in \{1, 2, 3\},$

where all algebras share the same carrier set M, and the following covering relations hold:

$$M_1 \vDash M_2 \quad \textit{and} \quad M_2 \vDash M_3.$$

We say that these computation graphs conform to a three-layer architecture *if and only if the following conditions are satisfied:*

1. $G_3 \lhd_s G_2 \lhd_s G_1$
2. $|I_3| \leq |V_{\text{deg}}^{G_3}|$, *where* $|V_{\text{deg}}^{G_3}|$ *denotes the number of nodes in* G_3 *with a predecessor with multiple successors.*

In particular, Condition (2) ensures that the top level provides a maximally compact representation of the functional processes. At this level, integration with the algorithm learning framework introduced in Section 2.2.3 should take place, as it offers the most concise form of representation.

9.3 Summary

This section develops a rigorous mathematical foundation for computational abstraction using computation graphs over homogeneous algebras. It introduces structural abstraction by defining quotient algebras that eliminate semantically irrelevant intermediate states while preserving computational behavior. Two computation graphs are structurally equivalent if their quotient algebras are isomorphic. To address the limits of structural abstraction, modular abstraction is introduced, allowing parts of computations to be treated as black boxes while preserving control flow semantics—though this too is shown to be undecidable. The section culminates in the concept of a three-level architecture, formalizing how computations can be organized into hierarchical layers of abstraction based on function composition. This framework bridges low-level operations with higher-level algorithmic structure, offering a more tractable setting for learning algorithms.

10 Discussion and Outlook

This thesis addresses the problem of learning algorithms from memory traces, using computation graphs over homogeneous algebras as the underlying computational model. In earlier work, it was shown that, assuming a clear distinction between state and user data, such computation graphs can be reconstructed from memory traces[KGA24]. This reduces the task of algorithm learning to predicting branching behavior.

Formally, this is framed as a classification task: a machine learning model must predict the correct operation of the transition algebra for a given element of the homogeneous algebra — that is, the underlying data structure. This prediction guides traversal through the corresponding computation graph in accordance with the intended control flow.

At the outset of this work, we identified two key challenges: (1) Previous experiments have shown that machine learning models experience significant drops in accuracy when the distribution of the data processed by a learned algorithm changes. (2) Control flow paths that are rarely executed tend to be underrepresented in the training data, leading to poor classification performance on those paths.

To address these limitations, and based on the distinction between induction - drawing conclusions about a population from samples using statistical methods - and abstraction - deriving logic rules - this work investigated the following research questions:

1. Induction: How can the adverse effects of distribution shifts and imbalanced training data on model accuracy be mitigated?
2. Abstraction: How can algorithmic abstraction be conceptually and formally defined in the context of rule-based systems?
3. Abstraction: How can the homogeneous and transition algebras be inferred directly from training data under relaxed assumptions, specifically:
    a) when state and payload data are not distinguishable, and
    b) when only a set of primitives defining the homogeneous algebra's operations is known?

In this context, the shift from induction to abstraction is central: abstraction enables algorithm learning in a rule-based manner that is inherently robust to changes in input data distribution. This chapter concludes the thesis by reflecting on the insights gained in relation to the research questions and by discussing the conceptual framework within which these insights are valid. The remainder of this chapter is organized around the questions posed above.

10.1 Imbalanced Training Data

The challenges arising from imbalanced training data were not directly addressed in this work. However, our findings regarding the fundamental limitations of abstracting algorithmic behavior, i.e. logic rules governing control flow, in Section 9 in a purely algorithmic manner highlight the importance of inductive – and thus statistically grounded – approaches. Consequently, the relevance of this issue has grown. Future research will therefore need to engage more deeply with this topic in the context of algorithm learning within the formal framework of computation graphs over homogeneous algebras.

10.2 Distribution Shifts

In addition to our formal investigation of the concept of abstraction in the context of algorithm learning, our considerations on how to handle distribution shifts form a central pillar of this work. Our considerations on handling distribution shifts are based, on the one hand, on theoretical analysis and argumentation, and on the other hand, on empirical investigations using running examples in the domain of comparison sorts. While the results of the theoretical approach have implications that go beyond individual classes of algorithms, the empirically derived findings remain largely limited to comparison sorts, consequently, we have have to differentiate between these scientific paradigms.

10.2.1 Theoretical Results

Our investigation of distribution shifts is fundamentally based on the sampling models introduced in Section 2.4.1, particularly the Monte Carlo model. These models define stochastic processes that induce a joint distribution over training data points $P(X, Y)$, thereby formally supporting the assumption of independent and identically distributed (i.i.d.) samples — a central premise of the Empirical Risk Minimization (ERM) framework.

On this formal basis, we were able to generally rule out the presence of concept shifts arising from the specific structure of the algorithm learning problem. Furthermore, we theoretically established a relationship between the input distribution of an algorithm and the induced joint distribution $P(X, Y)$. However, we did not provide a formal proof that a change in the input distribution necessarily leads to a change in the joint distribution $P(X, Y)$. A possible direction for future work would be to investigate the conditions under which such a statement can be made. A corresponding formal criterion could enable the selection of appropriate domain generalization methods, even across comparative sorting tasks, without prior empirical validation.

As already discussed in Section 2.4.1, there is a fundamental incompatibility between covariate shifts and label shifts in this context. For computation graphs combined with the Monte Carlo model as a sampling model, it is therefore sufficient to empirically measure covariate shifts in order to rule out not only concept shifts but also potential label shifts.

10.2.2 Conceptual Results

With DIBE, we propose a domain generalization approach that aims to learn a domain-invariant representation of predicates from computation graphs in a latent feature space, using binary regularization, embedding clustering, and contrastive learning. Our theoretical justification of DIBE in Section 7.1 does not refer to a specific class of algorithms, but rather to the general structure of computation graphs over homogeneous algebras and predicates. From a theoretical perspective, the approach is therefore not limited to comparison-based sorting.

According to the taxonomy of Wang et al.[WLL+23a], DIBE can primarily be classified as a form of 'representation learning', or more specifically as a form of 'explicit feature alignment', as DIBE aims at shaping the latent feature space in a binary way. However, due to its staged procedure and the clustering step used to generate pseudolabels in the latent feature space, there are also overlaps with approaches that Wang et al. categorize under 'learning strategy', more specifically 'self-supervised learning'.

10.2.3 Empirical Results

In the course of this work, we empirically investigated the quality and extent of shifts in the joint distribution $P(X, Y)$ caused by changing input distributions, using classical comparison-based sorting algorithms as an example. Furthermore, we analyzed the effectiveness of the domain generalization framework based on selected representative methods. Accordingly, the specific results are limited to these representatives. However, it seems reasonable to assume that the methods—such as the quantification of covariate shifts presented in Chapter 5—can also be transferred to other classes of algorithms. We now proceed with the reflection on Experiments I and II.

Reflections on Experiment I

In Experiment I, we observed significant covariate shifts in the predictor distribution $P(X)$ of Insertion Sort, even for minor changes in the distribution of input data to be sorted. Based on this and in conjunction with our theoretical considerations, we were able to rule out not only covariate shifts but also concept shifts.

The explanatory power of these results is, however, limited by the high empirically estimated expected deviation $\mathbb{E}(\sqrt{JS_{DV}(P_t - P_{t+1})}) \approx 0.17$ between successive estimates of the predictor density. This can partly be attributed to the high dimensionality of the data and thus to the curse of dimensionality, which presents an intrinsic challenge. At the same time, future work should investigate to what extent the assumption underlying Silverman's rule of thumb[Sil86] for determining the bandwidth matrix in KDE — namely, approximately Gaussian-distributed data[Sil86] — holds for the predictors of Insertion Sort.

Another assumption that should be examined in future work is the transferability of the results for Insertion Sort to other comparison-based sorting algorithms in the taxonomy shown in

Table 2.1. Here, we have implicitly assumed that changes in the input distribution also lead to covariate shifts in these algorithms and grounded the selection of respective domain generalization methods on this assumption.

Reflections on Experiment II

As part of the second experiment, we evaluated, in addition to the problem-specific method DIBE, the two domain generalization approaches MLDG and FISH in the context of algorithm learning, specifically for comparison-based sorting algorithms. As a baseline, we used the minimization of empirical risk via gradient descent.

The selection of MLDG and FISH was based on the analysis of the problem structure presented in Chapter 4, as well as on the widely accepted taxonomy of domain generalization methods by Wang et al.[WLL+23a]. The choice of these specific methods is therefore grounded both in theoretical considerations and in their adoption by the research community, as indicated by their citation counts.

The evaluation methodology aims to strike a balance between resource consumption and the significance of the results. Accordingly, for each architecture–hyperparameter combination, training was repeated five times. These repetitions were used to compute the corresponding expected accuracy values, both within individual domains and across domains.

A comparison of the accuracy values computed in this way across the evaluated methods reveals that the differences are often minor. Accordingly, future work should consider the use of statistical significance tests, or at the very least, report the variances to better assess the reliability of the observed results.

A key observation in this work is the substantial decline in cross-domain accuracy with increasing dimensionality of the training data. This effect is not limited to specific training methods such as DIBE, but is observed across different approaches. This stands in clear contrast to our previous work, where all considered sorting algorithms achieved an accuracy of at least $\approx 77\%$. Currently, two explanations are considered: (1) In earlier studies, accuracy was estimated solely on data whose distribution matched that of the training set. In contrast, the test set in this work includes additional, previously unseen domains. (2) The training data used in this work is considerably more heterogeneous in structure, which may exceed the generalization capabilities of the neural network architectures under consideration. Both hypotheses should be examined in future research. In particular, the first explanation would call into question the generalization ability of the machine learning models trained in this manner.

The hypothesis that the domain generalization methods MLDG, FISH, and DIBE achieve higher accuracy compared to ERM under changing predictor distributions is supported, in the majority of cases, by the results presented in Tables B.2, B.5, B.8, and B.11 for MLDG.

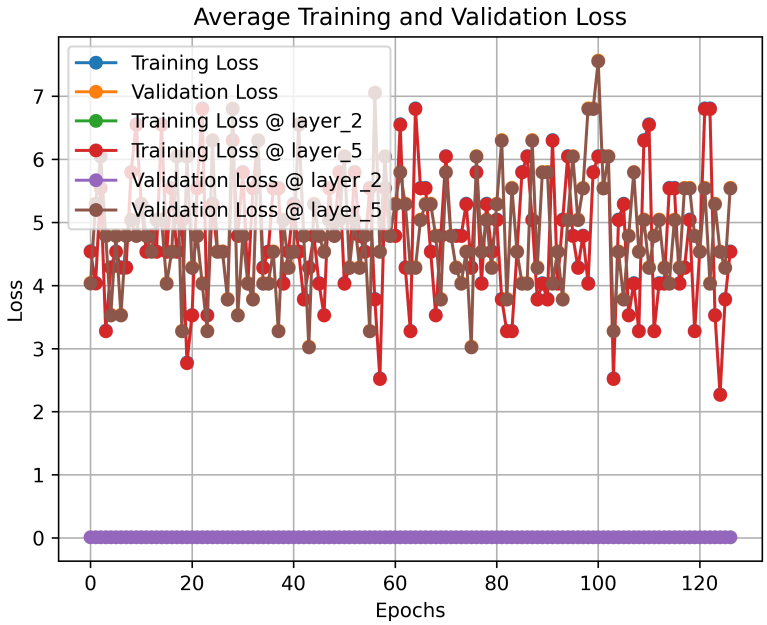

Figure 10.1: Exemplary progression of the loss curves for DIBE during fine-tuning. The green and purple curves represent the computed contrastive loss for the training and validation data, respectively. A constant value of zero raises concerns regarding the effectiveness of the contrastive loss function.

In the case of DIBE, the results are considerably more nuanced. While the method performs convincingly for Insertion Sort (see Table B.2), the observed contrastive loss values — illustrated, for example, in Figure 10.1 — raise substantial concerns about the effectiveness of cluster formation and, consequently, the fine-tuning step. Future work should therefore focus particularly on analyzing the cluster structure within the latent feature space.

10.3 Abstraction

We initially defined abstraction along two dimensions: (1) as the compression of structurally equivalent computations, and (2) as the ability to derive fixed logical or symbolic rules from empirical data.

With the introduction of structural abstraction in Chapter 9, we formally defined the idea of a semantic and structure preserving compression of a computation based on computation graphs over homogeneous algebras. Combined with the Three-Level Architecture, this led to a conceptual framework that enables a systematic analysis of this form of abstraction. While previous work consistently assumed a fixed level of abstraction, the Three-Level Architecture allows for a flexible examination of varying degrees of abstraction—provided the control flow remains unchanged. In particular, the topmost layer serves as an interface for highly condensed integration with machine learning methods, without requiring explicit rule-based or modular abstraction.

If the function $classify$ is implemented using inductive probabilistic methods, a single misclassification may suffice to significantly distort the control flow of the learned algorithm. In contrast, once rules derived through abstraction are known and verified as correct, they enable precise execution—regardless of the number of repetitions or the diversity of input. This idea is addressed through the introduction of modular abstraction, also discussed in Chapter 9. The formation of modules allows certain parts of the control flow to be treated as reliably functioning black boxes.

On the other hand, our undecidability result concerning modular abstraction — while perhaps unsurprising in light of Rice's Theorem [HMU01] — demonstrates that precisely this type of rule derivation is inherently difficult for algorithms to access.

However, it is important to emphasize that the non-existence of a general decision procedure for determining whether two representations constitute modular abstractions should not be conflated with the existence of constructive procedures for specific cases. Furthermore, approximate solutions remain a viable avenue for future exploration.

The third research question was only touched upon by the introduction of the Three-Level Architecture. Further investigations are necessary, particularly addressing the open question of state isolation.

Bibliography

[ABGL20] M. Arjovsky, L. Bottou, I. Gulrajani, D. Lopez-Paz. *Invariant Risk Minimization*. 2020. arXiv: 1907.02893 [stat.ML]. URL: https://arxiv.org/abs/1907.02893 (cit. on p. 63).

[All70] F. E. Allen. "Control flow analysis". In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, pp. 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479 (cit. on p. 95).

[Alp25a] Alphabet. *Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization @ Googel Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1911.08731 (visited on 01/27/2025) (cit. on p. 63).

[Alp25b] Alphabet. *Domain Adaptive Ensemble Learning*. 2025. URL: https://scholar.google.no/citations?user=gRIejugAAAAJ&hl=vi (visited on 01/27/2025) (cit. on p. 63).

[Alp25c] Alphabet. *Domain Generalization for Medical Image Analysis: A Review @ Google Scholar*. 2025. URL: https://scholar.google.com/citations?user=YjaJ5qQAAAAJ&hl=de&oi=sra (visited on 01/27/2025) (cit. on p. 56).

[Alp25d] Alphabet. *Domain Generalization for Object Recognition with Multi-task Autoencoders*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1508.07680 (visited on 01/27/2025) (cit. on p. 63).

[Alp25e] Alphabet. *Domain Generalization through Meta-Learning: A Survey @ Google Scholar*. 2025. URL: https://scholar.google.com/citations?user=QANF-5kAAAAJ&hl=de&oi=sra (visited on 01/27/2025) (cit. on p. 56).

[Alp25f] Alphabet. *Domain Generalization via Invariant Feature Representation @ Google Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1301.2115 (visited on 01/27/2025) (cit. on p. 63).

[Alp25g] Alphabet. *Domain Generalization with Domain-Specific Aggregation Modules @ Googel Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1809.10966 (visited on 01/27/2025) (cit. on p. 63).

[Alp25h] Alphabet. *Domain Generalization: A Survey @ Google Scholar*. 2025. URL: https://scholar.google.com/citations?user=gRIejugAAAAJ&hl=de&oi=sra (visited on 01/27/2025) (cit. on p. 56).

[Alp25i] Alphabet. *Domain-Adversarial Training of Neural Networks*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1508.07680 (visited on 01/27/2025) (cit. on p. 63).

[Alp25j] Alphabet. *Generalizing to Unseen Domains: A Survey on Domain Generalization @ Google Scholar*. 2025. URL: https://scholar.google.com/citations?user=hBZ_tKsAAAAJ&hl=de&oi=sra (visited on 01/27/2025) (cit. on p. 56).

[Alp25k] Alphabet. *Gradient Matching for Domain Generalization @ Googel Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=2104.09937 (visited on 01/27/2025) (cit. on pp. 62, 63).

[Alp25l] Alphabet. *Invariant Risk Minimization @ Google Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1907.02893 (visited on 01/27/2025) (cit. on p. 63).

[Alp25m] Alphabet. *Learning to Generalize: Meta-Learning for Domain Generalization @ Googel Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1710.03463 (visited on 01/27/2025) (cit. on p. 63).

[Alp25n] Alphabet. *Learning to Generalize: Meta-Learning for Domain Generalization @ Googel Scholar @ Googel Scholar*. 2025. URL: https://scholar.google.com/citations?user=0I2qH0oAAAAJ&hl=en (visited on 01/27/2025) (cit. on p. 63).

[Alp25o] Alphabet. *omain Generalization in Computational Pathology: Survey and Guidelines @ Google Scholar*. 2025. URL: https://scholar.google.com/citations?user=xbEURXEAAAAJ&hl=de&oi=sra (visited on 01/27/2025) (cit. on p. 56).

[Alp25p] Alphabet. *Out-of-Distribution Generalization via Risk Extrapolation (REx) @ Googel Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=2003.00688 (visited on 01/27/2025) (cit. on p. 64).

[Alp25q] Alphabet. *Scatter Component Analysis: A Unified Framework for Domain Adaptation and Domain Generalization@ Google Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1510.04373 (visited on 01/27/2025) (cit. on p. 63).

[Alp25r] Alphabet. *Self-Challenging Improves Cross-Domain Generalization @ Googel Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=2007.02454 (visited on 01/27/2025) (cit. on p. 63).

[Alp25s] Alphabet. *SelfReg: Self-supervised Contrastive Regularization for Domain Generalization*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=2104.09841 (visited on 01/27/2025) (cit. on p. 64).

[Alp25t] Alphabet. *Unified Deep Supervised Domain Adaptation and Generalization@ Google Scholar*. 2025. URL: https://scholar.google.com/scholar_lookup?arxiv_id=1709.10190 (visited on 01/27/2025) (cit. on p. 63).

[Arv20] K. Arvai. *kneed: Knee-point detection in Python*. Accessed: 2025-06-04. 2020. URL: https://github.com/arvkevi/kneed (cit. on p. 83).

[BBBK11] J. Bergstra, R. Bardenet, Y. Bengio, B. Kégl. "Algorithms for Hyper-Parameter Optimization". In: *Advances in Neural Information Processing Systems*. Ed. by J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, K. Weinberger. Vol. 24. Curran Associates, Inc., 2011. URL: https://proceedings.neurips.cc/paper_files/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf (cit. on p. 84).

[BBCP06] S. Ben-David, J. Blitzer, K. Crammer, F. Pereira. "Analysis of Representations for Domain Adaptation". In: *Advances in Neural Information Processing Systems*. Ed. by B. Schölkopf, J. Platt, T. Hoffman. Vol. 19. MIT Press, 2006. URL: https://proceedings.neurips.cc/paper_files/paper/2006/file/b1b0432ceafb0ce714426e9114852ac7-Paper.pdf (cit. on p. 58).

[BLS11] G. Blanchard, G. Lee, C. Scott. "Generalizing from several related classification tasks to a new unlabeled sample". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., 2011, pp. 2178–2186. ISBN: 9781618395993 (cit. on pp. 30, 31, 66).

[BLZ23] G. Bai, C. Ling, L. Zhao. *Temporal Domain Generalization with Drift-Aware Dynamic Neural Networks*. 2023. arXiv: 2205.10664 [cs.LG]. URL: https://arxiv.org/abs/2205.10664 (cit. on p. 57).

[Bon20] V. Bonnici. *Kullback-Leibler divergence between quantum distributions, and its upper-bound*. 2020. arXiv: 2008.05932 [cs.LG]. URL: https://arxiv.org/abs/2008.05932 (cit. on p. 44).

[BSC18] Y. Balaji, S. Sankaranarayanan, R. Chellappa. "MetaReg: towards domain generalization using meta-regularization". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 1006–1016 (cit. on p. 63).

[CCM92] J. H. Cross, E. J. Chikofsky, C. H. May. "Reverse Engineering**Based, in part, on "Reverse Engineering and Design Recovery: A Taxonomy" by E. J. Chikofsky and J. H. Cross II, which appeared in IEEE Software 7(1), 13-17. Copyright IEEE 1990." In: ed. by M. C. Yovits. Vol. 35. Advances in Computers. Elsevier, 1992, pp. 199–254. DOI: https://doi.org/10.1016/S0065-2458(08)60596-3. URL: https://www.sciencedirect.com/science/article/pii/S0065245808605963 (cit. on p. 37).

[CDB+19] F. M. Carlucci, A. D'Innocente, S. Bucci, B. Caputo, T. Tommasi. *Domain Generalization by Solving Jigsaw Puzzles*. 2019. arXiv: 1903.06864 [cs.CV]. URL: https://arxiv.org/abs/1903.06864 (cit. on pp. 30, 64).

[CHH02] M. Campbell, A. Hoane, F.-h. Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291 (cit. on p. 15).

[Cho+15] F. Chollet et al. *Keras*. https://keras.io. Accessed: 2025-06-04. 2015 (cit. on p. 73).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844 (cit. on p. 33).

[DC18] A. D'Innocente, B. Caputo. "Domain Generalization with Domain-Specific Aggregation Modules". In: *CoRR* abs/1809.10966 (2018). arXiv: 1809.10966. URL: http://arxiv.org/abs/1809.10966 (cit. on p. 63).

[DFO20] M. P. Deisenroth, A. A. Faisal, C. S. Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020 (cit. on pp. 48, 49, 127, 128).

[EKSX96] M. Ester, H.-P. Kriegel, J. Sander, X. Xu. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231 (cit. on p. 70).

[ES03] D. Endres, J. Schindelin. "A new metric for probability distributions". In: *IEEE Transactions on Information Theory* 49.7 (2003), pp. 1858–1860. DOI: 10.1109/TIT.2003.813506 (cit. on p. 44).

[FAL17] C. Finn, P. Abbeel, S. Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *CoRR* abs/1703.03400 (2017). arXiv: 1703.03400. URL: http://arxiv.org/abs/1703.03400 (cit. on pp. 30, 31, 59).

[FLL] W. Feng, L. Liu, T. Liu. "On Deterministically Approximating Total Variation Distance". In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1766–1791. DOI: 10.1137/1.9781611977912.70. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977912.70. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611977912.70 (cit. on p. 45).

[For17] O. Forster. *Analysis 2: Differentialrechnung im $\mathbb{R}^n$, Gewöhnliche Differentialgleichungen*. 11., erw. Auflage. Wiesbaden: Springer Spektrum, 2017. ISBN: 978-3-658-19410-9 (cit. on pp. 43, 68, 70).

[GBC16] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 65, 69, 70, 79, 80, 84).

[GBKZ16] M. Ghifary, D. Balduzzi, W. B. Kleijn, M. Zhang. *Scatter Component Analysis: A Unified Framework for Domain Adaptation and Domain Generalization*. 2016. arXiv: 1510.04373 [cs.CV]. URL: https://arxiv.org/abs/1510.04373 (cit. on p. 63).

[GF22] N. Ghassemi, E. Fazl-Ersi. *A Comprehensive Review of Trends, Applications and Challenges In Out-of-Distribution Detection*. 2022. arXiv: 2209.12935 [cs.LG]. URL: https://arxiv.org/abs/2209.12935 (cit. on p. 56).

[GKZB15] M. Ghifary, W. B. Kleijn, M. Zhang, D. Balduzzi. *Domain Generalization for Object Recognition with Multi-task Autoencoders*. 2015. arXiv: 1508.07680 [cs.CV]. URL: https://arxiv.org/abs/1508.07680 (cit. on p. 63).

[GPM+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML]. URL: https://arxiv.org/abs/1406.2661 (cit. on p. 58).

[GPS17] S. Gulwani, O. Polozov, R. Singh. "Program Synthesis". In: *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pp. 1–119. ISSN: 2325-1107. DOI: 10.1561/2500000010. URL: http://dx.doi.org/10.1561/2500000010 (cit. on p. 37).

[GUA+16] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, V. Lempitsky. *Domain-Adversarial Training of Neural Networks*. 2016. arXiv: 1505.07818 [stat.ML]. URL: https://arxiv.org/abs/1505.07818 (cit. on p. 63).

[Hen24] N. Henze. *Stochastik für Einsteiger: Eine Einführung in die faszinierende Welt des Zufalls*. German. 14th ed. eBook Packages: Life Science and Basic Disciplines (German Language). Springer Spektrum Berlin, Heidelberg, 2024, pp. XVIII, 530. ISBN: 978-3-662-67729-2. DOI: 10.1007/978-3-662-67729-2. URL: https://doi.org/10.1007/978-3-662-67729-2 (cit. on pp. 43, 46, 47).

[HMU01] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. Vol. 32. 1. New York, NY, USA: ACM Press, 2001, pp. 60–65. DOI: http://doi.acm.org/10.1145/568438.568455 (cit. on p. 116).

[HSM+00] R. Hahnloser, R. Sarpeshkar, M. Mahowald, R. Douglas, H. Seung. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". English (US). In: *Nature* 408.6815 (Dec. 2000). ISSN: 0028-0836. DOI: 10.1038/35050018 (cit. on p. 84).

[HSW89] K. Hornik, M. Stinchcombe, H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208 (cit. on p. 68).

[HTF09] T. Hastie, R. Tibshirani, J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer, 2009. URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/ (cit. on pp. 26, 27, 31, 33, 41, 48, 59, 68, 84, 85).

[Huy22] C. Huyen. *Designing Machine Learning Systems: An Iterative Process for Production-ready Applications*. Illustrated, Revised Edition. O'Reilly Media, Incorporated, 2022, p. 367. ISBN: 9781098107963 (cit. on p. 28).

[HWXH20] Z. Huang, H. Wang, E. P. Xing, D. Huang. *Self-Challenging Improves Cross-Domain Generalization*. 2020. arXiv: 2007.02454 [cs.CV]. URL: https://arxiv.org/abs/2007.02454 (cit. on p. 63).

[IEE25a] IEEE. *ACM Digital Library*. https://dl.acm.org/. Accessed: 2025-05-30. 2025 (cit. on p. 55).

[IEE25b] IEEE. *Domain Generalization with Adversarial Feature Learning @ IEEE*. 2025. URL: https://ieeexplore.ieee.org/document/8578664 (visited on 01/27/2025) (cit. on p. 63).

[IEE25c] IEEE. *IEEE Transactions on Knowledge and Data Engineering*. https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=69. Accessed: 2025-05-30. 2025 (cit. on p. 55).

[IEE25d] IEEE. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=34. Accessed: 2025-05-30. 2025 (cit. on p. 55).

[IEE25e] IEEE. *IEEE Xplore*. https://ieeexplore.ieee.org/Xplore/home.jsp. Accessed: 2025-05-30. 2025 (cit. on p. 55).

[Int23a] International Conference on Learning Representations. *ICLR 2023 Conference Website*. https://iclr.cc/Conferences/2023. Accessed: 2025-05-30. 2023 (cit. on p. 55).

[Int23b] International Conference on Machine Learning. *ICML 2023 Conference Website*. https://icml.cc/Conferences/2023. Accessed: 2025-05-30. 2023 (cit. on p. 55).

[Int24a] International Conference on Learning Representations. *ICLR 2024 Conference Website*. https://iclr.cc/Conferences/2024. Accessed: 2025-05-30. 2024 (cit. on p. 55).

[Int24b] International Conference on Machine Learning. *ICML 2024 Conference Website*. https://icml.cc/Conferences/2024. Accessed: 2025-05-30. 2024 (cit. on p. 55).

[JRX+23] M. Jahanifar, M. Raza, K. Xu, T. Vuong, R. Jewsbury, A. Shephard, N. Zamanitajeddin, J. T. Kwak, S. E. A. Raza, F. Minhas, N. Rajpoot. *Domain Generalization in Computational Pathology: Survey and Guidelines*. 2023. arXiv: 2310.19656 [eess.IV]. URL: https://arxiv.org/abs/2310.19656 (cit. on p. 56).

[KB15] D. P. Kingma, J. Ba. "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. 2015. URL: https://arxiv.org/abs/1412.6980 (cit. on pp. 75, 79).

[KCJ+21] D. Krueger, E. Caballero, J.-H. Jacobsen, A. Zhang, J. Binas, D. Zhang, R. L. Priol, A. Courville. *Out-of-Distribution Generalization via Risk Extrapolation (REx)*. 2021. arXiv: 2003.00688 [cs.LG]. URL: https://arxiv.org/abs/2003.00688 (cit. on p. 64).

[KGA24] P. Kunz, I. Georgievski, M. Aiello. "Towards a Framework for Learning of Algorithms: The Case of Learned Comparison Sorts". In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*. Ed. by K. Larson. Main Track. International Joint Conferences on Artificial Intelligence Organization, Aug. 2024, pp. 4353–4360. DOI: 10.24963/ijcai.2024/481. URL: https://doi.org/10.24963/ijcai.2024/481 (cit. on pp. 15, 19–27, 33–35, 37, 39, 43, 45, 46, 54, 66, 73, 83, 84, 89, 95, 106, 108, 111).

[KPKL21] D. Kim, S. Park, J. Kim, J. Lee. *SelfReg: Self-supervised Contrastive Regularization for Domain Generalization*. 2021. arXiv: 2104.09841 [cs.CV]. URL: https://arxiv.org/abs/2104.09841 (cit. on p. 64).

[KT17] C. P. Kitsos, T. L. Toulias. "Hellinger Distance Between Generalized Normal Distributions". In: *Journal of Advances in Mathematics and Computer Science* 21.2 (Mar. 2017), pp. 1–16. DOI: 10.9734/BJMCS/2017/32229. URL: https://journaljamcs.com/index.php/JAMCS/article/view/229 (cit. on p. 45).

[KVK21] A. Kristo, K. Vaidya, T. Kraska. *Defeating duplicates: A re-design of the LearnedSort algorithm*. 2021. arXiv: 2107.03290 [cs.DS]. URL: https://arxiv.org/abs/2107.03290 (cit. on p. 38).

[KW22] D. P. Kingma, M. Welling. *Auto-Encoding Variational Bayes*. 2022. arXiv: 1312.6114 [stat.ML]. URL: https://arxiv.org/abs/1312.6114 (cit. on p. 58).

[KYF24] A. G. Khoee, Y. Yu, R. Feldt. *Domain Generalization through Meta-Learning: A Survey*. 2024. arXiv: 2404.02785 [cs.LG]. URL: https://arxiv.org/abs/2404.02785 (cit. on p. 56).

[LHS20] P. H. Le-Khac, G. Healy, A. F. Smeaton. "Contrastive Representation Learning: A Framework and Review". In: *CoRR* abs/2010.05113 (2020). arXiv: 2010.05113. URL: https://arxiv.org/abs/2010.05113 (cit. on p. 69).

[Lim25] S. N. Limited. *Springer Link*. https://link.springer.com/. Accessed: 2025-05-30. 2025 (cit. on p. 55).

[Lis87] B. Liskov. “Data Abstraction and Hierarchy”. In: *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)*. OOPSLA ’87. Orlando, Florida, USA: Association for Computing Machinery, 1987, pp. 17–34. ISBN: 0897912667. DOI: 10.1145/62138.62141. URL: https://doi.org/10.1145/62138.62141 (cit. on p. 95).

[Llo82] S. Lloyd. “Least squares quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489 (cit. on pp. 70, 83).

[LPWK18] H. Li, S. J. Pan, S. Wang, A. C. Kot. “Domain Generalization with Adversarial Feature Learning”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 5400–5409. DOI: 10.1109/CVPR.2018.00566 (cit. on p. 63).

[LWS18] Z. Lipton, Y.-X. Wang, A. Smola. “Detecting and Correcting for Label Shift with Black Box Predictors”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by J. Dy, A. Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 3122–3130. URL: https://proceedings.mlr.press/v80/lipton18a.html (cit. on p. 28).

[LWZ+24] Y. Li, X. Wang, R. Zeng, P. K. Donta, I. Murturi, M. Huang, S. Dustdar. *Federated Domain Generalization: A Survey*. 2024. arXiv: 2306.01334 [cs.LG]. URL: https://arxiv.org/abs/2306.01334 (cit. on p. 56).

[LYSH17a] D. Li, Y. Yang, Y. Song, T. M. Hospedales. “Learning to Generalize: Meta-Learning for Domain Generalization”. In: *CoRR* abs/1710.03463 (2017). arXiv: 1710.03463. URL: http://arxiv.org/abs/1710.03463 (cit. on p. 31).

[LYSH17b] D. Li, Y. Yang, Y.-Z. Song, T. M. Hospedales. *Learning to Generalize: Meta-Learning for Domain Generalization*. 2017. arXiv: 1710.03463 [cs.LG]. URL: https://arxiv.org/abs/1710.03463 (cit. on pp. 30, 59, 62, 63).

[MAS+24] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, M. Farajtabar. *GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models*. 2024. arXiv: 2410.05229 [cs.LG]. URL: https://arxiv.org/abs/2410.05229 (cit. on p. 15).

[MBS13] K. Muandet, D. Balduzzi, B. Schölkopf. *Domain Generalization via Invariant Feature Representation*. 2013. arXiv: 1301.2115 [stat.ML]. URL: https://arxiv.org/abs/1301.2115 (cit. on pp. 30, 63).

[Met22] C. Metz. “The New Chatbots Could Change the World. Can You Trust Them?” In: *The New York Times* (Dec. 2022). Accessed: 2025-05-22. URL: https://www.nytimes.com/2022/12/10/technology/ai-chat-bot-chatgpt.html (cit. on p. 15).

[MP69] M. Minsky, S. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969 (cit. on p. 67).

[MPAD17] S. Motiian, M. Piccirilli, D. A. Adjeroh, G. Doretto. *Unified Deep Supervised Domain Adaptation and Generalization*. 2017. arXiv: 1709.10190 [cs.CV]. URL: https://arxiv.org/abs/1709.10190 (cit. on p. 63).

[Mur22] K. P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL: http://probml.github.io/book1 (cit. on p. 44).

[Mur23] K. P. Murphy. *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023. URL: http://probml.github.io/book2 (cit. on p. 43).

[Neu23] NeurIPS Conference. *NeurIPS 2023 Conference Website*. https://neurips.cc/Conferences/2023. Accessed: 2025-05-30. 2023 (cit. on p. 55).

[Neu24] NeurIPS Conference. *NeurIPS 2024 Conference Website*. https://neurips.cc/Conferences/2024. Accessed: 2025-05-30. 2024 (cit. on p. 55).

[NS18] F. Nielsen, K. Sun. *Guaranteed Deterministic Bounds on the Total Variation Distance between Univariate Mixtures*. 2018. arXiv: 1806.11311 [cs.LG]. URL: https://arxiv.org/abs/1806.11311 (cit. on p. 45).

[NS22] K. Nozawa, I. Sato. "Evaluation Methods for Representation Learning: A Survey". In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. Ed. by L. D. Raedt. Survey Track. International Joint Conferences on Artificial Intelligence Organization, July 2022, pp. 5556–5563. DOI: 10.24963/ijcai.2022/776. URL: https://doi.org/10.24963/ijcai.2022/776 (cit. on p. 30).

[Ope24] OpenAI. *ChatGPT*. https://chat.openai.com. Large language model (GPT-3.5) accessed via https://chat.openai.com. 2024 (cit. on p. 15).

[PB76] F. E. Petry, A. W. Biermann. "Reconstruction of Algorithms from Memory Snapshots of Their Execution". In: *ACM Annual Conference*. 1976, pp. 530–534 (cit. on p. 37).

[RM22] H. Rahimian, S. Mehrotra. "Frameworks and Results in Distributionally Robust Optimization". In: *Open Journal of Mathematical Optimization* 3 (July 2022), pp. 1–85. ISSN: 2777-5860. DOI: 10.5802/ojmo.15. URL: http://dx.doi.org/10.5802/ojmo.15 (cit. on p. 60).

[Rou87] P. J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. URL: https://www.sciencedirect.com/science/article/pii/0377042787901257 (cit. on p. 83).

[RPY+25] Y. Ren, J. Pu, Z. Yang, J. Xu, G. Li, X. Pu, P. S. Yu, L. He. "Deep Clustering: A Comprehensive Survey". In: *IEEE Transactions on Neural Networks and Learning Systems* 36.4 (2025), pp. 5858–5878. DOI: 10.1109/TNNLS.2024.3403155 (cit. on p. 69).

[Rüs14] L. Rüschendorf. *Mathematische Statistik*. German. 1st ed. Masterclass. 36 b/w illustrations. Part of the eBook Packages: Life Science and Basic Disciplines (German Language). Springer Spektrum Berlin, Heidelberg, 2014, pp. XI, 427. ISBN: 978-3-642-41996-6. DOI: 10.1007/978-3-642-41997-3. URL: https://doi.org/10.1007/978-3-642-41997-3 (cit. on p. 45).

[SAIR11] V. Satopaa, J. Albrecht, D. Irwin, B. Raghavan. "Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior". In: *Proceedings of the 31st International Conference on Distributed Computing Systems Workshops*. IEEE. 2011, pp. 166–171. DOI: 10.1109/ICDCSW.2011.20 (cit. on p. 83).

[SHM+16] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961 (cit. on p. 15).

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Monographs on Statistics and Applied Probability. Chapman and Hall, 1986 (cit. on pp. 48, 49, 113, 127).

[SKHL20] S. Sagawa, P. W. Koh, T. B. Hashimoto, P. Liang. *Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization*. 2020. arXiv: 1911.08731 [cs.LG]. URL: https://arxiv.org/abs/1911.08731 (cit. on p. 63).

[SM16] C. Sang-Hun, J. Markoff. "Master of Go Board Game Is Walloped by Google Computer Program". In: *The New York Times* (Mar. 2016). Accessed: 2025-05-22. URL: https://www.nytimes.com/2016/03/10/world/asia/google-alphago-lee-se-dol.html (cit. on p. 15).

[SS95] H. Siegelmann, E. Sontag. "On the Computational Power of Neural Nets". In: *Journal of Computer and System Sciences* 50.1 (1995), pp. 132–150. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1995.1013. URL: https://www.sciencedirect.com/science/article/pii/S0022000085710136 (cit. on p. 37).

[SST+21] Y. Shi, J. Seely, P. H. S. Torr, N. Siddharth, A. Y. Hannun, N. Usunier, G. Synnaeve. "Gradient Matching for Domain Generalization". In: *CoRR* abs/2104.09937 (2021). arXiv: 2104.09937. URL: https://arxiv.org/abs/2104.09937 (cit. on pp. 31, 32, 59, 63, 77).

[Ten25] TensorFlow Developers. *TensorFlow API Documentation*. Accessed: 2025-06-04. 2025. URL: https://www.tensorflow.org/api_docs (cit. on pp. 73, 81).

[TP98] S. Thrun, L. Y. Pratt. "Learning to Learn: Introduction and Overview". In: *Learning to Learn*. 1998. URL: https://api.semanticscholar.org/CorpusID:60987823 (cit. on pp. 29, 59).

[Tur36] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.1 (1936), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230 (cit. on pp. 20, 22, 107).

[Uni25] C. University. *arxiv*. https://arxiv.org. Accessed: 2025-05-30. 2025 (cit. on p. 55).

[Vil09] C. Villani. *Optimal Transport: Old and New. Old and New*. 1st ed. Vol. 338. Grundlehren der mathematischen Wissenschaften. Springer-Verlag Berlin Heidelberg 2009. Springer Berlin, Heidelberg, 2009, pp. XXII + 976. ISBN: 978-3-662-50180-1. DOI: 10.1007/978-3-540-71050-9 (cit. on p. 44).

[VM19] R. Volpi, V. Murino. "Addressing model vulnerability to distributional shifts over image transformation sets". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019 (cit. on p. 61).

[Web97] B. Weber. "Swift and Slashing, Computer Topples Kasparov". In: *The New York Times* (May 1997). Accessed: 2025-05-22. URL: https://www.nytimes.com/1997/05/12/nyregion/swift-and-slashing-computer-topples-kasparov.html (cit. on p. 15).

[WLL+23a] J. Wang, C. Lan, C. Liu, Y. Ouyang, T. Qin, W. Lu, Y. Chen, W. Zeng, P. S. Yu. "Generalizing to Unseen Domains: A Survey on Domain Generalization". In: *IEEE Transactions on Knowledge and Data Engineering* 35.8 (2023), pp. 8052–8072. DOI: 10.1109/TKDE.2022.3178128 (cit. on pp. 30, 113, 114).

[WLL+23b] J. Wang, C. Lan, C. Liu, Y. Ouyang, T. Qin, W. Lu, Y. Chen, W. Zeng, P. S. Yu. "Generalizing to Unseen Domains: A Survey on Domain Generalization". In: *IEEE Transactions on Knowledge and Data Engineering* 35.8 (2023), pp. 8052–8072. DOI: 10.1109/TKDE.2022.3178128 (cit. on pp. 31, 43, 55–60).

[WYKN20] Y. Wang, Q. Yao, J. T. Kwok, L. M. Ni. "Generalizing from a Few Examples: A Survey on Few-shot Learning". In: *ACM Comput. Surv.* 53.3 (June 2020). ISSN: 0360-0300. DOI: 10.1145/3386252. URL: https://doi.org/10.1145/3386252 (cit. on p. 30).

[YOS+24] J. S. Yoon, K. Oh, Y. Shin, M. A. Mazurowski, H.-I. Suk. "Domain Generalization for Medical Image Analysis: A Review". In: *Proceedings of the IEEE* 112.10 (Oct. 2024), pp. 1583–1609. ISSN: 1558-2256. DOI: 10.1109/jproc.2024.3507831. URL: http://dx.doi.org/10.1109/JPROC.2024.3507831 (cit. on p. 56).

[YZDP20] Q. Yang, Y. Zhang, W. Dai, S. J. Pan. "Introduction". In: *Transfer Learning*. Cambridge University Press, 2020, pp. 3–22 (cit. on p. 29).

[ZLQ+23] K. Zhou, Z. Liu, Y. Qiao, T. Xiang, C. C. Loy. "Domain Generalization: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.4 (2023), pp. 4396–4415. DOI: 10.1109/TPAMI.2022.3195549 (cit. on pp. 27, 30, 31, 43, 54–56).

[ZY22] Y. Zhang, Q. Yang. "A Survey on Multi-Task Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 34.12 (2022), pp. 5586–5609. DOI: 10.1109/TKDE.2021.3070203 (cit. on p. 30).

[ZYQX20] K. Zhou, Y. Yang, Y. Qiao, T. Xiang. "Domain Adaptive Ensemble Learning". In: *IEEE Transactions on Image Processing* 30 (2020), pp. 8008–8018. URL: https://api.semanticscholar.org/CorpusID:212725587 (cit. on p. 63).

[ZZL+25] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, J.-R. Wen. *A Survey of Large Language Models*. 2025. arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223 (cit. on p. 15).

All links were last followed on June 4, 2025.

A Mathematical Background

A.1 KDE and SVD

In the context of Kernel Density Estimation (KDE) [Sil86], we formally justify the necessity of applying a Singular Value Decomposition (SVD)[DFO20] when the predictor matrix $P \in \mathbb{R}^{n \times m}$ is not invertible. Since only square matrices can be invertible [DFO20], we assume $n = m$ in the following discussion. In cases where $n \neq m$, or P is otherwise non-invertible, the use of SVD becomes a necessary preprocessing step. Notably, this step remains valid and applicable regardless of whether P is square or not [DFO20].

Theorem A.1
Let $M := [\mu_\mathbf{1}|...|\mu_n]^T \in \mathbb{R}^{n \times n}, \mu_\mathbf{i} \in \mathbb{R}^n$ *and* $\hat{M} := [\hat{\mu}|...|\hat{\mu}]^\top \in \mathbb{R}^{n \times n}$*, such that:*

(A.1) $\hat{\mu} = \frac{1}{n} \sum_{j=1}^{n} \mu_j$

Then it holds:

(A.2) M^{-1} *exists.* $\implies$ $(M - \hat{M})^{-1}$ *exists.*

Proof. $(M - \hat{M})^{-1}$ exists. $\iff$ $[\mu_\mathbf{1} + \hat{\mu}|...|\mu_n + \hat{\mu}]$ is linearly independent[DFO20].

$\iff$ $\sum_{i=1}^{n} \alpha_i(\mu_i + \hat{\mu}) = 0 \implies \alpha_1 = \cdots = \alpha_n = 0$

We assume M^{-1} exists such that $[\mu_\mathbf{1}|...|\mu_n]$ is linearly idependent and forms a basis of the row space $Row(M)$[DFO20]. On this basis, we continue examining Equation x in the following:

(A.3) $\sum_{i=1}^{n} \alpha_i(\mu_i + \hat{\mu})$

(A.4) $\iff \sum_{i=1}^{n} \alpha_i\mu_i + (\sum_{i=1}^{n} \alpha_i)\hat{\mu} = 0$

(A.5) $\iff \sum_{i=1}^{n} \alpha_i\mu_i = -(\sum_{i=1}^{n} \alpha_i)\hat{\mu}$

(A.6) $\iff \sum_{i=1}^{n} \alpha_i\mu_i = -(\sum_{i=1}^{n} \alpha_i)\frac{1}{n}\sum_{j=1}^{n} \mu_j$

$$\text{(A.7)} \quad \iff \sum_{i=1}^{n} \alpha_i \mu_i = \sum_{j=1}^{n} -(\frac{1}{n} \sum_{i=1}^{n} \alpha_i) \mu_j$$

$$\text{(A.8)} \quad \overset{\text{Basis Row(M)}}{\iff} \forall i : \alpha_i = -(\frac{1}{n} \sum_{i=1}^{n} \alpha_i)$$

As a consequence of Equation A.8 it holds:

$$\text{(A.9)} \quad \alpha_1 = \cdots = \alpha_n$$

We distinguish the following cases:

1. $a_i < 0 \implies a_i = -(\frac{1}{n} \sum_{i=1}^{n} \alpha_i) > 0$
2. $a_i > 0 \implies a_i = -(\frac{1}{n} \sum_{i=1}^{n} \alpha_i) < 0$

Both cases result in a contradiction implying $\alpha_1 = \cdots = \alpha_n = 0$. Consequently, $[\mu_\mathbf{1} + \hat{\mu} | ... | \mu_n + \hat{\mu}]$ is linearly independent and, therefore[DFO20], $(M - \hat{M})^{-1}$ exists.

Theorem A.2
Let $M := [\mu_\mathbf{1}|...|\mu_n]^T \in \mathbb{R}^{n \times n}, \mu_\mathbf{i} \in \mathbb{R}^n$ *and* $\hat{M} := [\hat{\mu}|...|\hat{\mu}]^\top \in \mathbb{R}^{n \times n}, \alpha \in \mathbb{R}_{\neq 0}$*, such that:*

$$\text{(A.10)} \quad \hat{\mu} = \sum_{i=1}^{n} \mu$$

$$\text{(A.11)} \quad H := \alpha (M - \hat{M})^\top (M - \hat{M})$$

Then it holds:

$$\text{(A.12)} \quad M^{-1} \textit{ exists.} \implies H^{-1} \textit{ exists.}$$

Proof. According to Deisenroth et al.[DFO20] and Theorem A.1 the following is true for any matrix $N \in \mathbb{R}^{n \times n}$:

1. N is invertible. $\implies$ $(N - \hat{N})$ is invertible.
2. N is invertible. $\iff$ $det(N) \neq 0$
3. $det(\alpha N) = \alpha^n det(N)$
4. $det(N) = det(N^\top)$

Consequently, the followings holds:

$$\text{(A.13)} \quad det(H) = det(\alpha (M - \hat{M})^\top (M - \hat{M})) = \alpha^n det(M - \hat{M})^2 \neq 0 \implies H^{-1} \text{ exists.}$$

Theorem A.2 applies in particular to cases where M is a predictor matrix and α is defined, in accordance with the principles of KDE, as follows:

$$\text{(A.14)} \quad \alpha := \frac{1}{n-1} \left(\frac{4}{d+2} \right)^{\frac{2}{d+4}} n^{-\frac{2}{d+4}} \in \mathbb{R}_{\neq 0}$$

B Results of Experiment II: Learning Algorithms under Distribution Shifts

B.1 Insertion Sort

Method	Neuron Count	Activation Functions	α	β	γ	A_{opt}
MLDG (8096)	10, 512, 512, 2	r, r, s, σ	10^{-2}	1	10^{-1}	$\approx 95.57\%$
DIBE (16384)	10, 256, 129, 3, 32, 32, 2	$r, s, s\ r, r, r, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 94.03\%$

Table B.1: A layer-wise overview of the architecture-hyperparameter combinations that achieved the highest cross-domain accuracy A_{opt} for Insertion Sort across varying domain-specific sample sizes, $n \in \{8096, 16384\}$. The activation functions are encoded as follows: r stands for $relu$, s stands for $sigmoid$ and the softmax-function is represented by σ.

Method	Neuron Count	Activation Functions	α	β	γ	A_{min}
MLDG (8096)	10, 512, 512, 2	r, r, s, σ	10^{-2}	1	10^{-1}	$\approx 92.24\%$
MLDG (8096)	10, 512, 512, 2	r, r, s, σ	1	1	10^{-2}	$\approx 91.83\%$
ERM (8096)	10, 512, 512, 512, 2	$s, r, s\ s, \sigma$	1	-	-	$\approx 91.39\%$
MLDG (8096)	10, 1024, 1024, 1024, 2	$r, s, s\ s, \sigma$	10^{-2}	1	10^{-1}	$\approx 91.27\%$
MLDG (8096)	10, 128, 128, 128, 2	$r, r, r\ s, \sigma$	10^{-2}	1	10^{-1}	$\approx 89.64\%$
DIBE (16384)	10, 256, 129, 3, 32, 32, 2	$r, s, s\ r, r, r, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 89.61\%$
DIBE (16384)	10, 512, 257, 3, 32, 32, 2	$r, s, s\ r, r, s, \sigma$	1	1	1	$\approx 86.87\%$
DIBE (16384)	10, 256, 129, 3, 64, 64, 2	$r, r, s\ s, s, r, \sigma$	1	10^{-4}	10^{-4}	$\approx 84.42\%$
DIBE (16384)	10, 256, 129, 3, 32, 32, 2	$r, r, s\ r, r, s, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 84.17\%$
DIBE (16384)	10, 256, 129, 3, 64, 64, 2	$r, r, s\ s, s, r, \sigma$	10^{-4}	10^{-4}	10^{-4}	$\approx 81.84\%$

Table B.2: A layer-wise analysis of the top five architectures and hyperparameter configurations that achieved the highest minimum domain-specific accuracy A_{min} for Insertion Sort, evaluated across two sample sizes per domain, $n \in \{8096, 16384\}$. MLDG shows superior performance at the smaller sample size, while DIBE proves more effective when the sample size is larger. The parameter names for MLDG and DIBE are consistent with Section 2.4.2, whereas the parameter names for FISH are aligned with Algorithm 2.

Method	Rank
n = 8096	
ERM	3
MLDG	1
FISH	33
DIBE	13
n = 16384	
ERM	290
MLDG	298
FISH	311
DIBE	1

Table B.3: The table presents the rankings of domain generalization approaches - MLDG, FISH, and DIBE - along with ERM as a baseline, evaluated using Insertion Sort. A higher rank corresponds to a lower ability to generalize across varying domains, as measured by smaller values of A_{min}.

B.2 Quick Sort

Method	Neuron Count	Activation Functions	α	β	γ	A_{opt}
MLDG (8096)	20, 512, 512, 6	r, r, s, σ	1	10^{-2}	10^{-1}	$\approx 52.47\%$
ERM (16384)	20, 1024, 1024, 6	s, r, s, σ	1	-	-	$\approx 52.54\%$

Table B.4: A layer-wise overview of the architecture-hyperparameter combinations that achieved the highest cross-domain accuracy A_{opt} for Quick Sort across varying domain-specific sample sizes, $n \in \{8096, 16384\}$. The activation functions are encoded as follows: r stands for $relu$, s stands for $sigmoid$ and the softmax-function is represented by σ.

Method	Neuron Count	Activation Function	α	β	γ	A_{min}
MLDG (8096)	20, 512, 6	s, r, σ	1	1	10^{-2}	$\approx 47.83\%$
FISH (8096)	20, 512, 512, 512, 6	s, r, r, s, σ	10^{-1}	10^{-1}	-	$\approx 47.44\%$
MLDG (8096)	20, 1024, 1024, 1024, 6	$s, s, s\ s, \sigma$	1	1	10^{-2}	$\approx 47.35\%$
ERM (8096)	20, 512, 512, 512, 6	$s, r, s\ s, \sigma$	1	-	-	$\approx 47.31\%$
DIPE (8096)	20, 256, 129, 3, 32, 32, 6	$s, s, s, r, r\ r, \sigma$	10^{-4}	1	10^{-4}	$\approx 47.29\%$
MLDG (16384)	20, 256, 256, 256, 6	$r, r, r\ s, \sigma$	10^{-1}	1	10^{-4}	$\approx 47.31\%$
MLDG (16384)	20, 128, 128, 128, 6	$r, s, s\ s, \sigma$	1	1	10^{-1}	$\approx 47.31\%$
ERM (16384)	20, 1024, 1024, 6	s, r, s, σ	1	-	-	$\approx 47.31\%$
MLDG (16384)	20, 128, 128, 128, 6	$s, r, r\ s, \sigma$	10^{-1}	1	10^{-1}	$\approx 47.31\%$
MLDG (16384)	20, 256, 256, 256, 6	$s, r, r\ s, \sigma$	10^{-4}	1	10^{-2}	$\approx 47.31\%$

Table B.5: A layer-wise analysis of the top five architectures and hyperparameter configurations that achieved the highest minimum domain-specific accuracy A_{min} for Quick Sort, evaluated across two sample sizes per domain, $n \in \{8096, 16384\}$. MLDG shows superior performance at the smaller sample size, while DIBE proves more effective when the sample size is larger. The parameter names for MLDG and DIBE are consistent with Section 2.4.2, whereas the parameter names for FISH are aligned with Algorithm 2.

Method	Rank
n = 8096	
ERM	4
MLDG	1
FISH	2
DIBE	5
n = 16384	
ERM	3
MLDG	1
FISH	26
DIBE	50

Table B.6: The table presents the rankings of domain generalization approaches - MLDG, FISH, and DIBE - along with ERM as a baseline, evaluated using Quick Sort. A higher rank corresponds to a lower ability to generalize across varying domains, as measured by smaller values of A_{min}.

B.3 Heap Sort

Method	Neuron Count	Activation Function	α	β	γ	A_{opt}
MLDG (8096)	22, 128, 128, 128, 24	s, r, s, s, σ	10^{-4}	1	1	$\approx 38.13\%$
ERM (16384)	22, 256, 256, 256, 24	$r, r, s, s\ \sigma$	10^{-1}	-	-	$\approx 30.08\%$

Table B.7: A layer-wise overview of the architecture-hyperparameter combinations that achieved the highest cross-domain accuracy A_{opt} for Heap Sort across varying domain-specific sample sizes, $n \in \{8096, 16384\}$. The activation functions are encoded as follows: r stands for $relu$, s stands for $sigmoid$ and the softmax-function is represented by σ.

Method	Neuron Count	Activation Function	α	β	γ	A_{min}
MLDG (8096)	22, 128, 128, 128, 24	s, r, s, s, σ	10^{-4}	1	1	$\approx 34.33\%$
MLDG (8096)	22, 128, 128, 24	r, r, s, σ	10^{-4}	1	10^{-4}	$\approx 32.75\%$
MLDG (8096)	22, 128, 128, 128, 24	r, r, s, σ	1	1	10^{-4}	$\approx 32.37\%$
MLDG (8096)	22, 128, 128, 128, 24	r, s, s, s, σ	1	1	1	$\approx 29.27\%$
MLDG (8096)	22, 128, 128, 128, 24	s, r, s, s, σ	10^{-4}	1	1	$\approx 28.80\%$
MLDG (16384)	22, 128, 24	r, s, σ	10^{-2}	1	1	$\approx 29.85\%$
MLDG (16384)	22, 256, 256, 256, 24	r, r, s, s, σ	10^{-1}	1	10^{-4}	$\approx 29.40\%$
MLDG (16384)	22, 128, 128, 24	s, r, s, σ	10^{-4}	1	10^{-4}	$\approx 26.36\%$
FISH (16384)	22, 256, 24	r, s, σ	10^{-1}	10^{-1}	-	$\approx 22.43\%$
MLDG (16384)	22, 256, 256, 24	r, s, s, σ	10^{-4}	1	10^{-4}	$\approx 21.27\%$

Table B.8: A layer-wise analysis of the top five architectures and hyperparameter configurations that achieved the highest minimum domain-specific accuracy A_{min} for Heap Sort, evaluated across two sample sizes per domain, $n \in \{8096, 16384\}$. MLDG shows superior performance at the smaller sample size, while DIBE proves more effective when the sample size is larger. The parameter names for MLDG and DIBE are consistent with Section 2.4.2, whereas the parameter names for FISH are aligned with Algorithm 2.

Method	Rank
n = 8096	
ERM	4
MLDG	1
FISH	2
DIBE	5
n = 16384	
ERM	6
MLDG	1
FISH	4
DIBE	21

Table B.9: The table presents the rankings of domain generalization approaches - MLDG, FISH, and DIBE - along with ERM as a baseline, evaluated using Heap Sort. A higher rank corresponds to a lower ability to generalize across varying domains, as measured by smaller values of A_{min}.

B.4 Top-Down Merge Sort

Method	Neuron Count	Activation Function	α	β	γ	A_{opt}
MLDG (8096)	26, 256, 256, 22	r, s, s, σ	10^{-2}	1	1	$\approx 25.13\%$
MLDG (16384)	26, 256, 256, 22	r, s, s, σ	10^{-2}	1	1	$\approx 22.18\%$

Table B.10: A layer-wise overview of the architecture-hyperparameter combinations that achieved the highest cross-domain accuracy A_{opt} for Top-Down Merge Sort across varying domain-specific sample sizes, $n \in \{8096, 16384\}$. The activation functions are encoded as follows: r stands for $relu$, s stands for $sigmoid$ and the softmax-function is represented by σ.

Method	Neuron Count	Activation Function	α	β	γ	A_{min}
MLDG (8096)	26, 256, 256, 22	r, s, s, σ	10^{-2}	1	1	$\approx 21.2\%$
MLDG (8096)	26, 256, 256, 256, 22	r, r, s, s, σ	10^{-2}	1	1	$\approx 19.49\%$
MLDG (8096)	26, 128, 128, 22	r, r, s, σ	1	10^{-2}	1	$\approx 19.23\%$
MLDG (8096)	26, 128, 128, 22	s, r, s, σ	1	10^{-2}	10^{-1}	$\approx 19.21\%$
MLDG (8096)	26, 128, 128, 128, 22	r, r, s, s, σ	1	10^{-2}	10^{-1}	$\approx 17.83\%$
MLDG (16384)	26, 256, 256, 22	r, r, s, σ	10^{-2}	1	10^{-1}	$\approx 20.19\%$
MLDG (16384)	26, 1024, 1024, 256, 22	s, s, s, s, σ	10^{-4}	1	1	$\approx 19.68\%$
DIPE (16384)	26, 128, 11, 64, 64, 22	$s, r, s, r, s\ \sigma$	1	10^{-4}	10^{-4}	$\approx 17.97\%$
ERM (16384)	26, 512, 512, 22	r, s, s, σ	10^{-2}	1	1	$\approx 17.90\%$
FISH (16384)	26, 512, 512, 512, 22	s, r, s, s, σ	10^{-2}	10^{-1}	10^{-1}	$\approx 17.37\%$

Table B.11: A layer-wise analysis of the top five architectures and hyperparameter configurations that achieved the highest minimum domain-specific accuracy A_{min} for Top-Down Merge Sort, evaluated across two sample sizes per domain, $n \in \{8096, 16384\}$. MLDG shows superior performance at the smaller sample size, while DIBE proves more effective when the sample size is larger. The parameter names for MLDG and DIBE are consistent with Section 2.4.2, whereas the parameter names for FISH are aligned with Algorithm 2.

Method	Rank
n = 8096	
ERM	9
MLDG	1
FISH	17
DIBE	35
n = 16384	
ERM	4
MLDG	1
FISH	5
DIBE	3

Table B.12: The table presents the rankings of domain generalization approaches - MLDG, FISH, and DIBE - along with ERM as a baseline, evaluated using Top-Down Merge Sort. A higher rank corresponds to a lower ability to generalize across varying domains, as measured by smaller values of A_{min}.

C Acknowledgements

I hereby declare that generative AI was used exclusively for the revision and editing of self-written text.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature