FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DOCTORAL THESIS

Multi-Agent based Neural Networks

Ing. Martin Bulín, MSc.

This doctoral thesis is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Ph.D.)

in the field of

Cybernetics

Supervisor
Prof. Ing. Josef Psutka, CSc.

Department of Cybernetics

Pilsen 2024

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Ing. Martin Bulín, MSc.

"The human brain is capable of only one strong emotion at a time, and if it be filled with curiosity or hope or love, then there is no room for fear."

Sir Arthur Conan Doyle

Acknowledgements

I would like to express my sincere gratitude to Ing. Jan Švec, Ph.D. for his invaluable mentorship and enthusiastic support throughout this journey. His guidance and encouragement have been instrumental in shaping my research and academic career.

I am also deeply thankful to Doc. Ing. Pavel Ircing, Ph.D. for his meticulous proofreading of my thesis and for providing valuable feedback. His insightful comments have greatly contributed to the refinement of my work.

I extend my appreciation to Prof. Ing. Josef Psutka, CSc. for his supervision, expertise, and patience throughout my studies.

Finally, I extend my deepest gratitude to my family for their unwavering support and encouragement during my academic journey. Their constant encouragement and belief in me have been the primary source of motivation.

Abstract

In the realm of computer science and information technology, scientific methodologies can be categorized into two overarching paradigms: analytical approaches and strategies grounded in machine learning or optimization techniques. In the context of machine learning, the massive employment of neural networks contrasts with the limited explainability of their internal processes. The Universal Approximation Theorem formulated as early as the 1990s also suggests room for improvement in recent trends, particularly in the design of network architectures. This study delves into fundamental neural principles and presents a new perspective on working with them by introducing an original methodology for designing neural network architectures tailored to address the classification problems at hand. At the core of the method lies the fusion of a genetic algorithm, attention mechanism, and reinforcement learning utilized for training agents within a multi-agent system. The developed algorithm is evaluated on five classification scenarios, ranging from simple, interpretable 2D tasks to complex multi-dimensional and multi-class problems. A minimal network architecture is generated for each task. Experimental evaluation indicates promising scalability in handling larger parameter sets, hints at partial explainability in generated network architectures and unveils new directions for future research and exploration.

Anotace

Vědecké postupy a metody v oblasti výpočetní techniky a informačních technologií lze z širšího úhlu pohledu rozdělit na analytické přístupy a strategie založené na strojovém učení či optimalizačních technikách. V rámci strojového učení hraje klíčovou roli využití neuronových sítí, jejichž výsostné postavení kontrastuje s omezenými možnostmi vysvětlení jejich vnitřních procesů. Všeobecná aproximační věta formulovaná již v 90. letech minulého století také naznačuje prostor pro vylepšení dnešních trendů, především pak postupu při návrhu architektur sítí. Tato práce nabízí alternativní pohled na využití neurálních principů představením vlastní metody pro návrh architektury neuronové sítě vedoucí k řešení zadaného klasifikačního problému. Základním stavebním kamenem práce je kombinace genetického algoritmu, attention mechanismu a reinforcement learningu pro učení agentů v multiagentním systému. Navržený algoritmus je vyhodnocen na pěti klasifikačních problémech, které zahrnují jednoduché úlohy vykreslitelné ve 2D prostoru, ale i komplexní problémy o více dimenzích a třídách. Pro každou úlohu je vygenerována ukázková minimální struktura sítě. Zpracované výsledky naznačují další možnosti využití a otevírají širokou škálu nových směrů pro budoucí výzkum.

Table of contents

List of tables **ix**

List of figures **x**

Nomenclature **xiv**

1 Introduction **1**

1.1 Thesis Objectives . . . 3

1.2 Thesis Outline . . . 4

2 Neural Networks **5**

2.1 Biological Analogy . . . 11

2.2 Feedforward Architectures . . . 13

2.2.1 Multi-Layer Perceptron (MLP) . . . 13

2.2.2 Time Delay Network (TDNN) . . . 15

2.2.3 Convolutional Network (CNN) . . . 17

2.2.4 Residual Network . . . 20

2.3 Recurrent Architectures . . . 21

2.3.1 Long Short-Term Memory (LSTM) . . . 23

2.3.2 Gated Recurrent Unit (GRU) . . . 25

2.3.3 Bidirectional Network (BRNN) . . . 26

2.3.4 Time-Distributed Layers . . . 27

2.3.5 Special Recurrent Structures . . . 27

2.4 Transformers . . . 30

2.4.1 Attention Mechanism . . . 30

2.4.2 Original Transformer . . . 36

2.4.3 Encoder-Only Models . . . 38

2.4.4 Decoder-Only Models . . . 40

2.4.5 Encoder-Decoder Models . . . 41

2.5 Special Architectures 42
2.6 Learning Algorithm 44
2.6.1 Loss Functions 46
2.6.2 Limitations of Backpropagation 47
2.6.3 Design Choices and Optimization Tools 48
2.7 Neural Architecture Search 51
2.7.1 Hyper-parameter Tuning 51
2.7.2 Pruning Methods 53
2.7.3 Building Methods 54
2.8 Frameworks for Neural Networks 56

3 Related Principles and Technologies 57
3.1 Reinforcement Learning 57
3.1.1 Formulation of a RL problem 58
3.1.2 Introduction to Optimization Techniques 63
3.1.3 Q-Learning 69
3.1.4 SARSA 70
3.1.5 Deep Q-Networks 70
3.1.6 Policy Gradient Algorithms 72
3.1.7 Actor-Critic 74
3.1.8 Reinforcement Learning Frameworks 76
3.2 Multi-Agent Systems 77
3.2.1 Multi-Agent Reinforcement Learning 78
3.2.2 MARL Frameworks 80
3.3 Genetic Algorithms 81
3.4 Human-in-the-Loop 82

4 Multi-Agent based Neural Networks 83
4.1 Problem Formulation 83
4.2 Key Concepts 86
4.2.1 No Layers 86
4.2.2 Network Architecture Representation 87
4.2.3 Multi-Agent System Analogy 89
4.2.4 Self-Attention of Multi-Agent Systems 89
4.2.5 Employing Genetic Principles 91
4.3 The Algorithm 92
4.3.1 Initialization 94

4.3.2 Fitness Function 94
4.3.3 Monitor - Graphical Interface 95
4.3.4 Termination Condition 96
4.3.5 Multi-Agent based Mutation 96
4.3.6 New Generation Selection 101
4.3.7 Collecting Results 101
4.4 Potential Extensions 102

5 Experimental Evaluation 103
5.1 Setup and Metrics 103
5.1.1 Default Configuration 105
5.2 Examples 105
5.2.1 Fundamental 2D Problems 106
5.2.2 A Multi-class 2D Problem - Continents 108
5.2.3 A Multidimensional Problem - Michalski's Trains 109
5.2.4 Intent Classification 112
5.3 Summary of Results 114

6 Discussion 115
6.1 Future Research Directions 116

7 Conclusion 117

References 118

Appendix A Notation Conventions 127

Appendix B Project API 130
B.1 Framework Implementation 130
B.2 Structure of the Workspace 132
B.3 Configuration File Example 133

List of tables

3.1 Parametrization of model-free RL algorithms. 66

5.1 Default configuration settings used in the experiments. 105
5.2 Basic 2D problem. 106
5.3 XOR problem. 106
5.4 Features describing the trains (table overtaken from [20]). 109
5.5 Trains: encoded features. 109
5.6 Samples of the intent classification dataset sorted into classes and subsets. . 112
5.7 Results summary for all presented examples. 114

List of figures

2.1 A biological neural cell [130] 11
2.2 An artificial neuron. 13
2.3 Example of a feedforward (MLP) architecture. 14
2.4 Common transfer (activation) functions. 14
2.5 Single TDNN cell connected to the input layer. 15
2.6 A time-shifted window over the input data stream for a TDNN network. . . 16
2.7 The TDNN principle with subsampling (red) and without subsampling (red + gray). 17
2.8 The CNN (1D data) principle. 18
2.9 CNN principle (2D data). The yellow-marked $w \times h$ filter in the input layer corresponds to the little yellow cube in the hidden layer. 19
2.10 CNN (3D data). For example a video (width, height, time). 19
2.11 Principle of the max-pooling method. 20
2.12 Residual Network (ResNet) – skipping connection – principle. 20
2.13 Sequence data - task types. Figure is inspired by [69]. 22
2.14 Default RNN - cell body and data flow. 22
2.15 Long Short-Term Memory (LSTM) cell. 24
2.16 Gated Recurrent Unit (GRU) cell. 25
2.17 Bidirectional Recurrent Neural Network (BRNN) - the purple cells can be e.g. LSTM or GRU. 26
2.18 Illustration of the time-distributed wrapper. 27
2.19 A Hopfield network of three units. 27
2.20 A (Restricted) Boltzmann Machine example. 29
2.21 Simple recurrent networks (SRN). 29
2.22 Example of Self-Attention. Darker colors represent higher attention scores. 31
2.23 Basic self-attention – processing vector x_2 and yielding vector y_2, where $f()$ stands for the *softmax* function as given in Eq. 2.22. 32

2.24 Trainable version of self-attention with *query*, *key* and *value* transformations. 33
2.25 Fictive example of attention scores. Darker colors indicate higher attention. 34
2.26 Multi-head attention - projection to lower dimensionality and back. 35
2.27 Illustration of the attention mechanism applied on image pixels (red circle $\sim$ focus). 35
2.28 Transformer architecture – decomposition into *Encoder* and *Decoder* parts. 36
2.29 Transformer block - *Encoder*. 36
2.30 Transformer - Decoder block. 37
2.31 Encoder-only architecture employed for sentiment classification in movie reviews. 39
2.32 Wav2Vec 2.0 - original figure from [7]. Illustration of learning representations. 40
2.33 T5: Example of a training sample. 41
2.34 The Generative Adversarial Network concept. 42
2.35 Autoencoders (handling nonlinear relations). 43
2.36 Kohonen's Self-Organizing Map example. 43
2.37 BPTT unfolding an RNN through time. 45
2.38 Learning rate: (A) too low; (B) optimal; (C) too high (red) / way too high (orange). 48
2.39 Example of a dropped-out network. 49
2.40 The Neural Architecture Search with RL principle [137]. 52
2.41 The principle of network pruning. 54
2.42 The inner and outer learning loops in the Badger architecture [110]. 55

3.1 Placement of reinforcement learning within the machine learning family. . . 57
3.2 A single step of the reinforcement learning procedure (single agent). 58
3.3 Example of three independent trajectories of episodic RL tasks. 59
3.4 Example of a Markov chain (process) of two states. 60
3.5 Environment model: (a) observable (rare), (b) hidden (typical case). 61
3.6 Influence of the discount factor $\gamma \in [0,1]$ in Eq. 3.2. 62
3.7 Difference between *value* (a) and *Q-value* (b). 63
3.8 Categories of optimization algorithms. Green: RL algorithms [34]. 63
3.9 Illustration of the power of Bellman equation (Eq. 3.4). 64
3.10 Deriving optimal policy from optimal value with Q_1 and Q_2 being the Q-values of corresponding actions (a_1 and a_2). 67
3.11 Deriving optimal policy from optimal value [34]. 67
3.12 The ε-greedy strategy using a decay based on Eq. 3.10. 68
3.13 Q-Learning: Example of the target action selection. 69

3.14 Q-Learning vs. Deep Q-Networks. 70
3.15 Deep Q-Network reinforcement learning data flow. 71
3.16 Policy Gradient methods have a probability distribution on their output. . . 72
3.17 The data flow of the policy gradient method. 72
3.18 Actor-Critic principle. 75
3.19 Multi-agent system - N agents interacting with each other and with the environment. 77
3.20 MARL: (a) shared policy vs. (b) individual policies. 79
3.21 MARL: Joint vs. individual actions, states and rewards. 79
3.22 MARL: Multiple variants of agents synchronization. 80
3.23 Diagram of the Genetic Algorithm workflow. 82
3.24 The human-in-the-loop (HITL) principle. 82

4.1 Depiction of the proposed method's focus within machine learning (highlighted in yellow). Reinforcement learning principles (purple) are leveraged as a supportive tool. 83
4.2 An illustration of a pruned (and partially explainable) network, overtaken from [20]. 84
4.3 Number of architecture variants (right y-axis / red) vs. number of hidden neurons. 85
4.4 Elimination of layers: a) standard layer-wise approach; b) proposed approach. 86
4.5 The concept of a *Grid* defining the target network capacity. 87
4.6 The overall network architecture representation using a matrix embedding. . 88
4.7 Neural network viewed as a multi-agent system. 89
4.8 Idea of employing the *Attention mechanism* to map agents relations: (a) Attending embeddings of agents analogically to a NLP task; (b) Attention scores $\sim$ agents relations. 90
4.9 Idea of applying genetic algorithms principles for neural architecture search. 91
4.10 An overall view of the developed framework. 92
4.11 The main loop of the algorithm based on the GA principle. 93
4.12 Algorithm initialization methods - (b) example: $N = 2$ and $M = 1$ leads to $p = 3$. 94
4.13 Fitness function value across different β configurations. 94
4.14 Screenshot of the graphical interface. 95
4.15 The *mutation* process within a single episode of the GA outer loop. 96
4.16 A single epoch of the A4C learning (no policy updates included). 97
4.17 Actor (agent) and critic models structures sharing the attended input. 98

4.18 Two proposed variants of the agent embedding (θ_i being training samples). 99
4.19 The prior probability of activating an agents is based on its current suroundings. 100
4.20 Determining if entity A is better than entity B (different runs). 102

5.1 Fundamental 2D tasks of two classes - illustration in the 2D space. 106
5.2 Minimal structures, 2D problems: a) Basic-2D; b1) XOR ($p = 7$); b2) XOR ($p = 9$). 107
5.3 Algorithm performance on 2D tasks - mean and standard dev. of 10 realizations. 107
5.4 Generated networks - fundamental 2D tasks (corresponding with targets - Fig. 5.2). 108
5.5 The multi-class 2D task and a partially explainable network architecture solving it. 108
5.6 The Michalski's Trains task [92]. 109
5.7 Resulting generated network architecture for the Michalski's trains problem. 110
5.8 The Michalski's trains problem: trained attention scores and best entity statistics. 110
5.9 Probability of agents taking action *ON* (activating themselves). 111
5.10 Creation of the intent classification dataset. 112
5.11 Intent classification: Performance of the algorithm (mean of 10 realizations). 113
5.12 Resulting generated network architecture for the intent classification task. . 113

B.1 Implementation - structure of the framework. 130

Nomenclature

Machine Learning Acronyms

A2C	**A**dvantage **A**ctor **C**ritic
A3C	**A**synchronous **A**dvantage **A**ctor **C**ritic
A4C	**A**ttended **A**synchronous **A**dvantage **A**ctor **C**ritic
AI	**A**rtificial **I**ntelligence
ANN	**A**rtificial **N**eural **N**etwork
ASR	**A**utomatic **S**peech **R**ecognition
BERT	**B**idirectional **E**ncoder **R**epresentations from **T**ransformers
BPTT	**B**ack**P**ropagation **T**hrough **T**ime
BRNN	**B**i-directional **R**ecurrent **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
DL	**D**eep **L**earning
DQN	**D**eep **Q**-**N**etwork
GA	**G**enetic **A**lgorithm
GAN	**G**enerative **A**dversarial **Network**
GDA	**G**radient **D**escent **A**lgorithm
GOFAI	**G**ood **O**ld-**F**ashioned **AI**
GPT	**G**enerative **P**re-trained **T**ransformer

GPU	**G**raphics **P**rocessing **U**nit
HITL	**H**uman-**i**n-**t**he-**L**oop
LLaMA	**L**arge **L**anguage **M**odel **M**eta AI
LLM	**L**arge **L**anguage **M**odel
LSTM	**L**ong **S**hort-**T**erm **M**emory
MARL	**M**ulti-**A**gent **R**einforcement **L**earning
MAS	**M**ulti-**A**gent **S**ystem
MDP	**M**arkov **D**ecision **P**rocess
ML	**M**achine **L**earning
MLP	**M**ulti-**L**ayer **P**erceptron
MNIST	**M**ixed **N**ational **I**nstitute of **S**tandards and **T**echnology Database
NLP	**N**atural **L**anguage **P**rocessing
NN	**N**eural **N**etwork
OCR	**O**ptical **C**haracter **R**ecognition
PCA	**P**rincipal **C**omponent **A**nalysis
PPO	**P**roximal **P**olicy **O**ptimization
ReLU	**Re**ctified **L**inear **U**nit
ResNet	**Res**idual **Net**work
RL	**R**einforcement **L**earning
RNN	**R**current **N**eural **N**etwork
SARSA	**S**tate-**A**ction-**R**eward-**S**tate-**A**ction
SO(F)M	**S**elf-**O**rganizing (**F**eature) **M**ap
SotA	**S**tate **o**f **t**he **A**rt
SVM	**S**upport **V**ector **M**achine

T5	**T**ext-**t**o-**t**ext **T**ransfer **T**ransformer
TDNN	**T**ime-Delay **N**eural **N**etwork
TRPO	**T**rust **R**egion **P**olicy **O**ptimization

Symbols - Neural Networks

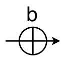
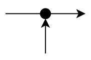
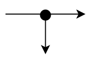
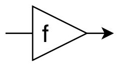
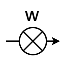
$\Delta^{(i)}$	r-by-p matrix of errors on r neurons in i^{th} layer for all samples
τ	number of time-steps (context-dependent structures)
$A^{(i)}$	r-by-p matrix of activities of r neurons in i^{th} layer for all samples
$a^{<t>}$	activity of a (recurrent) neuron at time t
$a_k^{(i)}$	activity of k^{th} neuron in i^{th} layer
$B^{(i)}$	vector of r biases for r neurons in i^{th} layer
$b_k^{(i)}$	bias connected to k^{th} neuron in i^{th} layer
$c^{<t>}$	state of a (recurrent) neuron at time t
$f'(\cdot)$	derivative of the transfer function $f(\cdot)$
$f(\cdot)$	transfer function
M	number of classes
N	problem dimension (sample size; number of features)
$o^{<t>}$	output of a (recurrent) neuron at time t
q	number of hidden layers
S	number of samples
U	m-by-p matrix: desired network output (targets) for all samples
$W^{(i)}$	r-by-s matrix of weights for synapses connecting s neurons in $(i-1)^{th}$ layer to r neurons in i^{th} layer
$w_{k,l}^{(i)}$	weight of synapse from l^{th} neuron in $(i-1)^{th}$ layer to k^{th} neuron in i^{th} layer
X	n-by-p matrix: network input (samples)

Y	m-by-p matrix: predicted network output for all samples ($Y = A^{(q)}$)
$Z^{(i)}$	r-by-p matrix of activations for r neurons in i^{th} layer for all samples
$z_k^{(i)}$	activation of k^{th} neuron in i^{th} layer

Symbols - Reinforcement Learning and Multi-Agent Systems

γ	discount factor, $\gamma \in [0,1]$
π	policy
$a^{<t>}$	action at time t
$Q^{(\pi)}(s,a)$	Q-value of state s under policy π when taking action a
$R^{<t>}$	Return at time t
$r^{<t>}$	reward at time t
$s^{<t>}$	state at time t
$v^{(\pi)}(s)$	value of state s under policy π

Graphical Symbols

🔵	hidden feedforward node
🟣	hidden recurrent node
🟩	input node
▷	output node
	addition (signal plus b)
	concatenate two signals
	copy a signal
	function of the signal: $f(\cdot)$
	multiplication (signal times w)

Chapter 1

Introduction

The remarkable journey of technological progress, spanning from the invention of the Turing machine in 1936 to the widespread adoption of personal computers around the 1980s, and culminating in the rapid advancements in hardware over the past two decades, has profoundly shaped both our daily lives and society at large. In tandem with the advances in hardware development, the ongoing exploration of theoretical principles continually pushes the boundaries of what is achievable with computers. This trend is beautifully characterized by Captain Jonathan Archer from the Star Trek series, highlighting the inherent human curiosity and desire to explore the uncharted territories.

> *"It's the unknown that defines our existence. We are constantly searching, not just for answers to our questions, but for new questions."*

Driven by both enthusiasm and market demands, technology is gradually integrated to be more and more in contact with humans. Bound together by their shared dependence on computational power, technological methods can be divided into two broad categories:

- *Deterministic methodologies.* These systems handle tasks that can potentially be resolved analytically or can be represented by limited state spaces, ensuring predictable and guaranteed results. They encompass all IT systems used in sectors such as banking, healthcare information, industrial control, or public administration.

- *Optimization strategies and machine learning.* In contrast, the majority of real-world problems are so complex that even the most advanced computers cannot exhaustively explore all possible combinations to find a solution. These encompass combinatorial NP problems, as well as tasks like weather forecasting or self-driving cars development. Here we use optimization techniques and machine learning models that are nowadays almost exclusively grounded in the principle of *neural networks*.

Neural networks, aiming to replicate the workings of the most powerful machine ever invented – human brain, have emerged as the dominant machine learning technique. Their influence reached a milestone in 2022 with the release of ChatGPT by OpenAI, marking a pivotal moment in bringing the AI phenomenon to the forefront of public consciousness. It's challenging to identify shortcomings in these large language models, as they operate almost seamlessly, fulfilling the expectations one might have for an AI tool and some in the community speculate that further advancements are hindered by hardware limitations only. On the other hand, there are three fundamental aspects inherent to LLMs that cannot be resolved solely by scaling up:

- *Hallucination.* This phenomenon in LLMs refers to the model's ability to generate plausible but false information. It occurs due to the model's attempt to generate text that fits the context, even if the information it generates is not grounded in reality.
- *Explainability.* Explaining the decisions made by neural networks in general (not only LLMs) poses significant challenges due to their complex architectures and vast numbers of parameters. Ensuring trust in a model's application relies on its transparency. Thus, the idea of scaling up its size seems like a move in the opposite direction.
- *Adaptivity.* Fine-tuning neural networks allows for parameter adaptation to new observations. However, for real-time applications requiring model updates based on new real-world data, minimizing (not scaling up) the number of parameters becomes essential. Adaptation is also closely linked to the explainability of the model, as optimal adaptation requires pinpointing where to make targeted changes within the network.

In this work, the aim is not to challenge the capabilities of LLMs, but rather to explore the application of fundamental neural principles from a different perspective. Rather than scaling up the size of models, we focus on designing architectures of very tiny models. The concept is supported by the *Universal Approximation Theorem* [26], initially proposed by G. Cybenko in 1989 and later complemented by subsequent works such as [57]. The theorem states the following.

> *A feedforward neural network with a single hidden layer, containing a finite number of neurons, can approximate any continuous function on a compact subset of Euclidean space to any desired degree of accuracy, given a sufficiently large number of neurons.*

The ongoing research in this field is evident in recent studies such as [44] from 2018 and the latest publication [62] in 2023, which demonstrates that a three-layer neural network can represent any multivariate function. Despite this potential, there are two key challenges.

1. *Training algorithm:* Despite advancements, the Backpropagation algorithm from the 1970s remains a cornerstone in training neural networks.

2. *Architecture design:* Building optimal neural network architectures, due to the enormous number of potential combinations, remains a challenge.

In this work, the second challenge is addressed by introducing a novel approach for designing neural networks tailored to arbitrary classification problems. In contrast to conventional neural architecture search methods, which typically involve tuning the meta-parameters based on Bayesian optimization [12, 32, 38, 67], reinforcement learning [136, 102, 63], or evolutionary algorithms [81, 108], in this work, the state space is directly formed by the variants of network architecture, particularly its components. We leverage the analogy between neural networks and multi-agent systems, integrating a multi-agent reinforcement learning algorithm enhanced with a specialized attention mechanism. The reinforcement learning policy is trained to modify the neural network architecture, acting as the mutation mechanism within a broader loop that integrates genetic algorithm principles. This seemingly complicated approach is described in detail, illustrating how seamlessly each individual method and component intertwine and complement one another.

The developed algorithm is evaluated on five classification scenarios, ranging from simple, interpretable 2D tasks to complex multi-dimensional and multi-class problems. The evaluation indicates promising scalability in handling larger parameter sets and hints at partial explainability in generated network architectures. In this sense, this work opens a portion of research questions and offers multiple directions for future research and exploration.

1.1 Thesis Objectives

The primary aim of this study is to explore fundamental principles and propose an innovative methodology in the realm of designing neural network architectures for arbitrary classification problems. Throughout this endeavour, several incremental steps have been identified and subsequently addressed:

1. Provide an in-depth examination of the evolution of neural network architectures throughout history. Discuss their underlying principles and assess their suitability for tackling various ML problems. Additionally, include a overview of widely used optimization techniques for the learning algorithm.

2. Work out a summary of currently available techniques addressing the problem of neural architecture search (NAS).

3. Outline the principles and technologies employed in the proposed methodology.
4. Present the developed methodology for constructing neural network architectures.
   - Formulate the problem and state the motivation for addressing it.
   - Present the key ideas and the developed algorithm.
   - Define research questions and specify the experiments to be evaluated.
5. Select appropriate testing examples and propose an experimental setup. Evaluate the effectiveness of the proposed algorithm.
6. Discuss the obtained results and elaborate on potential future research directions.

1.2 Thesis Outline

The thesis introduction provides a historical context and outlines the primary motivation behind the research. It offers a glimpse of the hypotheses proposed and the results obtained, serving as a trailer for what is to come in the thesis. Finally, it articulates the objectives of the study, setting the stage for the subsequent chapters.

Chapter 2 offers an in-depth exploration of neural networks, tracing their historical evolution from the mid-20th century to the present day. It delves into the development of various network architectures over time, discusses the learning algorithms employed, and provides insights into the current state-of-the-art in the field of neural architecture search.

Chapter 3 extensively covers the principles supporting the proposed method. It provides detailed explanations of reinforcement learning algorithms, focusing particularly on their application. Additionally, the chapter explores multi-agent systems and discusses the application of RL within them. Finally, it briefly touches upon the concept of genetic algorithms and the human-in-the-loop principle.

Chapter 4 introduces the developed method. It begins with a thorough problem formulation and hypothesis establishment. Subsequently, key concepts are presented, followed by an in-depth explanation of the overarching algorithm. Finally, the chapter concludes with suggestions for future research directions.

Chapter 5 presents the experimental evaluation of the proposed methodology. It starts by introducing the experimental setup and metrics used in the evaluation. Next, the chapter provides several examples demonstrating the deployment of the developed algorithm.

The study is discussed in Chapter 6 and subsequently concluded in Chapter 7. In the appendices, readers can find a summary of the notation conventions used and the user manual, including implementation details.

Chapter 2

Neural Networks

The original model of an artificial neuron, trying to imitate key features of a biological neural cell, dates back to 1943, when W. S. McCulloch and W. Pitts came with a highly simplified version [85]. A step-by-step elaboration over the years led to teachable systems nowadays known as artificial neural networks. These systems, the way we have been using them recently, are capable of learning and performing a human-like behaviour when solving one particular task. In the realm of specific tasks, particularly those that are not highly complex, methods based on artificial neural networks consistently yield fascinating results. Consequently, these methods are rightfully regarded as state-of-the-art classifiers, regressors, and, more recently, especially effective in the domain of generative models.

The term *Artificial Neural Network* (ANN) includes various methods that share a neural basis but differ primarily in their architectures and the types of data structures they can handle. Many issues across different domains can be formulated as machine learning problems. The initial step towards creating a successful ML system lies in an accurate problem formulation, data representation and preprocessing. Subsequently, the optimal artificial neural network architecture is typically selected based on the task definition and the nature of the data to be processed. In the following sections, we delve into a description of the most frequently used network architectures categorizing them into three groups. The first group encompasses architectures that rely on the *static* states of their cells. By referencing Eq. 2.1, where $z_i^{<t>}$ denotes the state of the i^{th} cell (neuron) at time t, we differentiate these from architectures featuring *dynamic* cells, where the current state is a function of the previous one.

$$\text{architecture} \sim \begin{cases} \text{static}, & \text{if} \quad z_i^{<t+1>} \neq function(z_i^{<t>}) \quad \forall\, i\, \forall\, t \\ \text{dynamic}, & \text{if} \quad \exists\, i,t : z_i^{<t+1>} = function(z_i^{<t>}) \end{cases} \tag{2.1}$$

Static systems (architectures) are commonly known as *Feedforward architectures* (see Sec. 2.2). On the other hand, dynamic systems, as determined by Eq. 2.1 and typically depicted with loops, fall under the category of *Recurrent architectures* (Sec. 2.3). Additionally, despite their potential inclusion in the first static group, we introduce a third distinct category for *Transformer architectures* (Sec. 2.4) due to their unique data processing characteristics.

Another approach to categorize ANNs is to explore the most significant breakthroughs chronologically, which is well documented in [72]. As already said, the first attempts date back to the middle of the 20th century, while the first real opening came in 1958 with the idea of a perceptron by Frank Rosenblatt (detailed in Sec. 2.2.1). At that time, the foundations of the majority of today's neural networks were established. Architectures primarily composed of derivatives of these perceptrons have proven effective and continue to be utilized extensively to this day.

The perceptron's promising capability of learning the basic OR/AND/NOT functions was further extended into a multi-category classifier presented as the ADALINE structure [121]. However, the enthusiasm of having a tool to solve complex AI problems was suppressed shortly thereafter, as it turned out perceptrons are not able to solve linearly inseparable tasks, such as the XOR problem. Today we know that those tasks are solvable using multiple non-linear layers (i.e., hidden layers), but at that time the way of making multilayer perceptrons learn had not been yet invented. This epoch is known as the *AI winter*, as especially skeptical conclusions of the Minsky's work *Perceptrons* [88] caused a freeze to funding and publications in AI.

The key *Backpropagation* learning algorithm (described in Sec. 2.6) based on the chain rule was firstly derived and implemented to run on computers by Finnish student S. Linnainmaa [79] in 1970 and in 1974 proposed to be used for neural networks after analyzing it in depth in [129]. This author was, interestingly, loosely inspired by Sigmund Freud's psychological theories about modelling the human mind with the concept of a backward flow of credit assignment. Even though the math had been already derived and the algorithm discovered, mostly because of the lack of academic interest and the loss of the faith in tackling problems pointed out in *Perceptrons*, the approach was popularized more than a decade later in [113]. Finally, the mathematical proof that multiple layers allow neural networks to theoretically implement any function, and certainly XOR, was given in [58]. Since then, ANNs have become popular again and started to be applied to real-world applications, such as the *Handwritten Zip Code Recognition* problem [74].

In response to the evolving demands of different tasks, numerous network architectures and optimization methods have been introduced over the years. Here's a brief chronological list of some of the most significant breakthroughs:

1982 *Hopfield network* [56]; The architecture is not actually related to the backpropagation learning and even dates back earlier. The Hopfield network was considered a recurrent structure, however, not really in the manner we imagine recurrent networks today. It served as a content-addressable *associative memory* system and it is described in more detail in Sec. 2.3.5.

- *SOM – Self-organizing maps* [70]; Introduced by Finnish professor T. Kohonen, SOMs produce a low-dimensional (typically a two-dimensional) discretized representation of the input space and is therefore a method to do dimensionality reduction using unsupervised learning (more in Sec. 2.5).

1986 *Boltzmann machine* [52]; This approach can be seen as a stochastic and generative counterpart of the Hopfield network. The restricted version (RBM) is being used in deep learning for weights pre-training till today (detailed in Sec. 2.3.5).

1987 *TDNN – Time Delay Neural Network* [127]; Mainly motivated by the speech recognition task, there was a need to consider context dependencies in data. The time-delay network is a special version of a multilayer feedforward neural network with the ability of context modeling and classification of patterns with shift-invariance (more in Sec. 2.2.2).

- *Autoencoders* [16]; Based on the neural principles, ANN structures started to be used for compression and data encoding tasks (see Sec. 2.5).

1990 *Backpropagation through time* [36]; The key idea for using backpropagation on recurrent neural networks lies in unrolling loops into several networks connecting one to another and limiting the number of time steps (see Sec. 2.3).

- Application in robotics, control engineering and games [93]; At that time, ANNs started to be used as decision makers in another branch of machine learning - *reinforcement learning* (see Sec. 3.1). The research in [78] showed a successful application to tasks like wall following or door passing as well as to playing logical games. Those programs soon reached their limits though and were not even close to the well-known Alpha Go or Chess artificial players we know today.

1993 *Siamese (twin) network* [19]; The idea of using the same weights for two models working in tandem was highly popularized in the era of deep learning, especially for *computer vision* tasks, however, the original idea is much older (more in Sec. 2.5).

1995 *Wake-sleep algorithm* [51]; G. E. Hinton and his team kept working on some extra tricks for a slightly different belief net setup, which was later on called *The Helmholtz Machine* [27]. It basically allowed the training of Boltzmann Machines to be done much faster.

- Other (not ANN) methods; With the idea of the kernel trick [25], *Support Vector Machines (SVMs)* became a mathematically optimal way of training an equivalent to a two layer neural network and started to be seen as superior to neural nets. Moreover, also other methods, notable *Random Forests* [54] proved to be very effective while having a neat mathematical theory behind them.

1997 *LSTM – Long Short-Term Memory* [55]; At that time, the key invention for sequential data modeling, capable of learning the long-term dependences in data, was published in 1997, however, its full power was also reached later with deep learning. The method is detailed in Sec. 2.3.1.

- *BRNN – Bidirectional Recurent Neural Network* [119]; In this approach, two recurrent layers of opposite directions of the data flow are connected to the same output. Calling it a generative deep learning, the output layer can get information from past (backwards) and future (forward) states simultaneously (see Sec. 2.3.3 for details).

1998 *CNN – Convolutional Neural Network* [75]; One of the most important ideas in the field of ANNs was published in 1998, when Yan Lecun, inspired by the weight-sharing mechanism in TDNNs, used a similar principle for positional-dependent features (especially useful for images) and invented convolutional layers (see Sec. 2.2.3).

2002 Restricted Boltzmann Machines in deep learning; With the failure of backpropagation in learning of deep structures, the early 2000s were a dark time for neural networks research again. The restricted version of a Boltzmann machine (see Fig. 2.20b) was initially invented under the name *Harmonium* in 1986 [120], however, in 2002, G. Hinton and his team came with the idea to use RBMs for weights initialization in networks with many layers [49], which led to a fast learning algorithm and significantly influenced the birth of deep learning.

2006 A fast learning algorithm for deep belief nets [10]; This algorithm meant a breakthrough significant enough to rekindle the interest in neural nets again. The movement in deep learning started with this paper and the idea that neural networks with many layers could be trained well, if the weights are initialized in a clever way rather than randomly. Since then the deep learning has been here and no winter is in sight.

2013 *Deep Q-Learning* [90]; Using deep ANNs in RL tasks was firstly published in 2013 and subsequently got even more attention in 2017 with so-called *DQfD* incorporating demonstrations (expert knowledge) into deep reinforcement learning, enhancing sample efficiency and accelerating learning (see Sec. 3.1.5).

- *Self-supervised learning* [87]; While not explicitly using the exact term, the idea of training models on large amounts of unlabeled data to learn useful representations is a foundational concept that aligns with the principles of self-supervised learning.

2014 *GRU – Gated Recurrent Unit* [24]; Alongside the LSTM cell, GRU is the other gating mechanism that were commonly used in recurrent structures at that time. The GRU cell has fewer parameters and seems to be more efficient and faster, while LSTMs are generally more accurate on datasets with longer sequences. Details in Sec. 2.3.2.

- *Attention mechanism* [8]; This key innovation revolutionized neural networks by allowing them to selectively focus on specific elements within input sequences. Inspired by human visual attention, it greatly improved the performance of models in tasks like machine translation and image captioning. The mechanism was firstly used in [8] for sequence-to-sequence learning and later massively incorporated into all Transformer-based applications. The concept is detailed in Sec. 2.4.1.

- *GAN – Generative Adversarial Network* [42]; GANs are considered one of the most interesting recent ideas in deep learning. There is a generator part producing fake samples with respect to the given dataset and trying to fool the second part - a discriminator, which is trying to learn boundaries between real and fake samples. There are many real-world applications (more in Sec. 2.5).

2015 *Batch normalization* [61]; This very basic but still significant optimization gained widespread adoption after its effectiveness in accelerating training and improving generalization was demonstrated.

2017 *Transformer - Attention is All You Need* [126]; This groundbreaking contribution has arguably been the most impactful advancement in the field of AI, revolutionizing sequence-to-sequence tasks in NLP. Departing from RNN or CNN structures, Transformers rely on self-attention mechanisms to process input sequences in parallel, enabling more efficient training on large datasets. This innovation has become a cornerstone in deep learning, with applications extending beyond NLP to various domains, such as computer vision or reinforcement learning. The Transformer's impact is evident in subsequent models like BERT, GPT, and T5, highlighting its enduring significance in the field.

- *Capsule networks* [115]; Proposed as an alternative to traditional convolutional neural networks, capsule networks aim to address issues related to viewpoint variation and hierarchical feature learning.

2018 *BERT – Bi-directional Encoder Representations from Transformers* [31]; A Transformer-based encoder-only approach that revolutionized natural language processing by providing high-quality text embeddings (details in Sec. 2.4.3).

2019 *GPT – Generative Pre-trained Transformer* [105]; This approach introduced by OpenAI, is a Transformer-based decoder-only model designed for unsupervised pre-training on vast amounts of text data. GPT has revolutionized natural language processing by learning contextualized language representations, enabling its adaptation to various downstream tasks through fine-tuning (see Sec. 2.4.4 for details).

2020 *T5 – Text-to-Text Transfer Transformer* [106]; This encoder-decoder Transformer-based approach, introduced by Google, is a versatile NLP model that frames all tasks in a text-to-text format. The unified approach simplifies various tasks and has proven effective for tasks like translation and summarization. Pre-trained on large datasets and fine-tuned, T5 demonstrates the power of a unified architecture for diverse language understanding tasks (see Sec. 2.4.5).

2021 *DALL-E* [96] This model from OpenAI revolutionized the world by introducing a model capable of generating diverse and high-quality images from textual descriptions.

2022 *ChatGPT* [95] As the release of the Transformer architecture in 2017 marked a significant breakthrough for the AI community, the introduction of OpenAI's public version of conversational AI chatbot in 2022 became a game changer for the rest of the world. Based on the GPT (Generative Pre-trained Transformer) architecture, it showcases the application of extensive unsupervised learning for natural and dynamic dialogue generation.

2023 *Large Language Model Meta AI (LLaMA)* [125] This is an open language model released by Meta (Facebook), free to be fine-tuned on custom datasets. It is a Meta's response to OpenAI's GPT models.

A complete list of important ANN techniques described in more detail is presented in sections below, sorted out based on the purpose rather than the year of invention.

2.1 Biological Analogy

The human brain is historically considered the most sophisticated machine ever observed. Its capability of solving complex tasks, learning new skills and actually being somehow responsible for the human consciousness and the way we perceive the world around us is shrouded in mystery. Due to its complicated structure, it is one of the last, if not the only one, human organ that we cannot accurately describe and explain its functionality. Both, the enormous computational power as well as the curiosity to reveal the mystery, make us try to mimic its behaviour artificially.

Let's sum up the facts related to the purpose of this work. As far as we know, the human brain consists of approximately 100 bilion neural cells and each of these cells can have up to 15,000 connections with other neurons via synapses. The neurons are capable of generating electrical signals called *action potentials*, which allows them to transmit information quickly [11]. The work of a single neuron consists of three basic functions that are being processed in three main parts of a cell (see Fig. 2.1):

1. *dendrites* - receive signals (or information) from outside;
2. *soma* - processes the incoming signals and determines whether or not to pass the information along;
3. *axon* - communicates the signals to other cells.

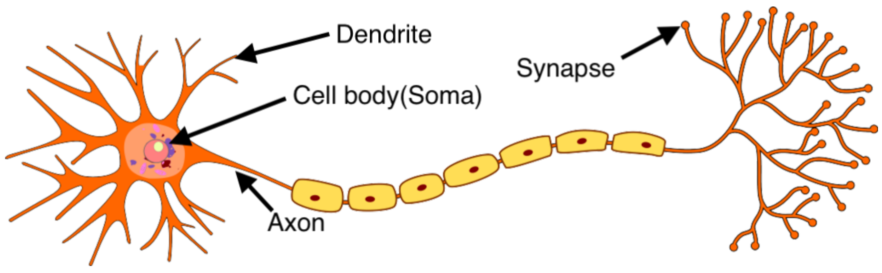


Fig. 2.1 A biological neural cell [130]

The single cell itself does not seem that complicated and therefore, what produces the behaviour solemnly called *inteligence*, must be the enormous amount of the cells and virtually an infinite number of combinations of connecting them. Out of the many, there are several facts about the human brain that are interesting for this work [30]:

- *Multitasking is impossible*. Should it look like that from the outside, we are actually super-quickly switching context instead.

- *Brain is a powerful machine.* The speed of information flow is about 250 mph. It is capable of about 1,000 processes per second. The capacity of the memory is challenging to quantify precisely, but estimates suggest the brain can store around 2.5 petabytes of information.

- *High energy consumption*: Despite representing only about 2% of the body's weight, the brain consumes approximately 20% of the body's energy. Compared to the human brain's average energy consumption of 20 watts, the power usage of LLMs during inference may vary, possibly ranging from tens to hundreds of watts.

- *Asynchronous processing and fault tolerance*: Minor failures will not result in memory loss. The architecture is decentralized.

- *Unique fingerprint*: Each brain anatomy is unique. The patterns of neural connections are distinct, contributing to individual differences in cognition and behavior.

- *Neuroplasticity* - in a lifetime, the brain is shaped partly by genes and largely by experience. The size is tripled the first year of life, stops developing in our late 40s and gets smaller as we get older. However, there is no evidence that the brain size matters. More importantly, neurons as well as synapses can die and new ones can be born and reorganized during a process called *neurogenesis*. Once a new neuron is born, it moves (is guided by chemical signals) to its final location. The final step of *neurogenesis* is the *differentiation* step, when the neuron settles and starts to communicate with its neighbours.

- *Brain areas* - there are different circuits in the brain responsible for different tasks. For example, reading aloud uses different pathways than reading silently.

- *Short term memory* lasts about 20-30 seconds. Most people hold memory for numbers or letters around 7 seconds and can store up to 7 digits in the working memory.

- *Dreaming mystery*: The purpose and mechanisms of dreaming remain a mystery to scientists. Dreaming occurs during the rapid eye movement (REM) stage of sleep, and it's thought to play a role in memory consolidation.

The complexity of a complete structure of the biological brain is incalculable and therefore, there is not the only general design of ANNs being used. Instead, several highly simplified architectures have been developed over the years, each of them designed for a specific task type in machine learning. Those tasks are defined by the nature of the problem to be artificially solved.

2.2 Feedforward Architectures

There are no loops in feedforward architectures and the work of a single cell is defaultly based on the principle of a *perceptron* [111]. In Fig. 2.2, with the reference to the biological template in Fig. 2.1, there are *dendrites* of k^{th} neuron in i^{th} layer carrying signals $a_{k,1}^{(i)},...,a_{k,j}^{(i)}$ that are being adjusted by parameters $b_k^{(i)}, w_{k,1}^{(i)},...,w_{k,j}^{(i)}$. Then the *soma* modelled as the blue part and finally the *axon* holding the output signal $a_k^{(i)}$ (see notation conventions in App. A).

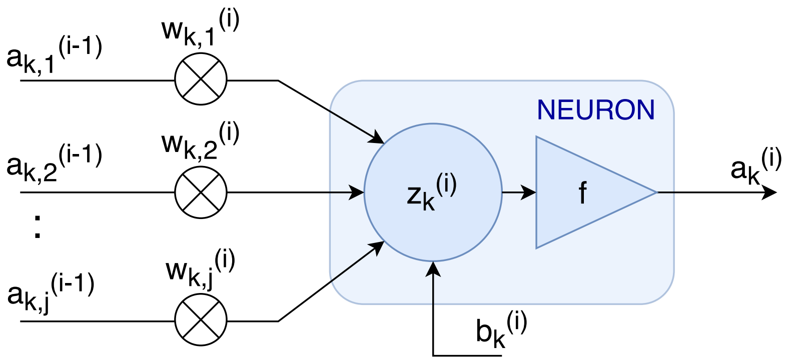


Fig. 2.2 An artificial neuron.

The process of *firing* the neuron consists of two steps. At first, assuming j being the number of input synapses (dendrites), the *activation* of the neuron $z_k^{(i)}$ is computed (Eq. 2.2).

$$z_k^{(i)} = \sum_{l=1}^{j} [a_l^{(i-1)} \cdot w_{k,l}^{(i)}] + b_k^{(i)} \tag{2.2}$$

with $a^{(0)} = x$ being the network input. Then we apply a chosen transfer function (see Sec. 2.2.1) to get the neuron *activity* (Eq. 2.3).

$$a_k^{(i)} = f(z_k^{(i)}) \tag{2.3}$$

2.2.1 Multi-Layer Perceptron (MLP)

The default (vanilla) neural network consists of multiple nodes arranged into layers. In Fig. 2.3, there is an example of such a structure with 2 input nodes (green), 1 output node (red) and one hidden layer of 3 nodes.

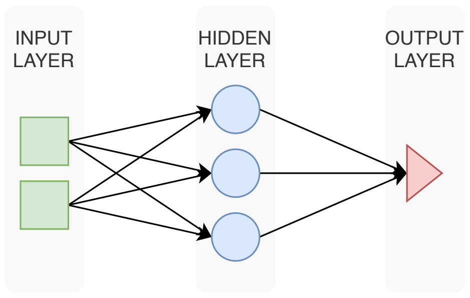


Fig. 2.3 Example of a feedforward (MLP) architecture.

In general, the size of the input layer is given by the problem dimension n and the size of the output layer, assuming a classification problem, is determined by the number of classes m. By default, the structure is considered fully-connected, so for each node there is a synapse to all nodes in the following layer. Arranging neurons into layers is one of the many deviations of the artificial approach from the biological template, however, it enables fast matrix computations for inference and learning procedures. Including a hidden layer makes the classifier capable of solving linearly inseparable tasks and in [58], it was proven that multiple layers can theoretically implement any function. However, despite the proof of a theoretical possibility, the optimal way of initialization and training is not known. As basic as this structure is, it is still widely used under names *dense*, *feedforward*, *MLP* or *fully-connected*. The network is trained by tuning its parameters (weights and biases) using the *backpropagation* algorithm explained in Sec. 2.6.

Transfer (activation) functions.[1] The learning algorithm (Sec. 2.6) needs the activation function to be (easily) differentiable. The most common activation functions are *hyperbolic tangent*, *sigmoid*, *ReLu* and recently also *swish* [107] (Fig. 2.4).

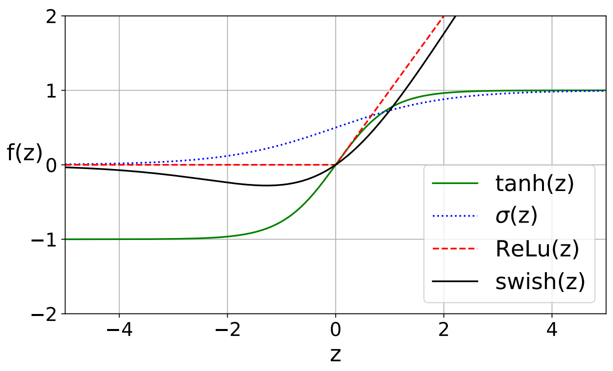


Fig. 2.4 Common transfer (activation) functions.

[1]The depicted transfer functions find applications also in other architectures.

Softmax function The softmax function, also referred to as the Boltzmann or Gibbs distribution in physics and statistical mechanics, was initially used in 1868 and later adopted in the field of deep learning [28]. Its purpose is to transform a vector of m real values into another vector of m real values, ensuring that the resulting values sum up to 1, and thereby allowing them to be interpreted as probabilities. The Softmax function is defined by Eq. 2.4, where z_i denotes the i-th element of the input vector z with a size of m.

$$\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \tag{2.4}$$

2.2.2 Time Delay Network (TDNN)

This approach is a special version of the MLP with the capability of classification of temporal patterns with shift invariance, such as speech for example. The shift invariant classification means that there is no explicit segmentation required prior to classification.

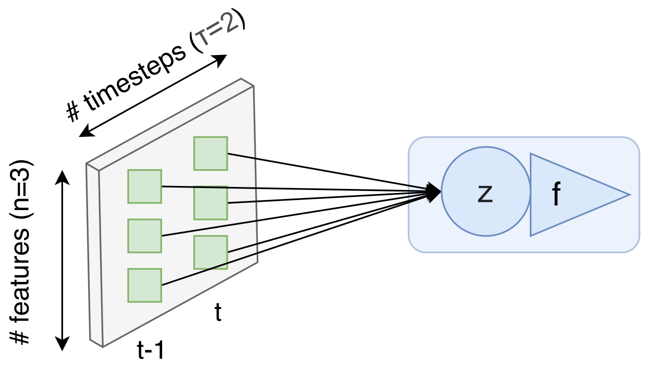


Fig. 2.5 Single TDNN cell connected to the input layer.

The idea was firstly presented in [127][2] on the task of phoneme recognition. In Fig. 2.5, there is an example of a single TDNN cell and the way it is connected to features in the input layer.

There is no change in the functionality of the cell body (see the perceptron in Fig. 2.2), but there is a difference in the arrangement of its inputs. As shown graphically in Fig. 2.5 and mathematically in Eq. 2.5, there are additionally past (time-delayed) features included. There is a window of τ time-steps shifted over a stream of context-dependent features and the cell activation z is then computed as a weighted sum of all the features from the contextual window taken from the input sequence of features.

[2] The notation in the original paper is different to the one in this work.

$$z = \sum_{i=1}^{n} \sum_{j=0}^{\tau-1} [x_i^{<t-j>} \cdot w_{i,j}] + b \tag{2.5}$$

For two-dimensional signals (such as time-frequency patterns or images) there is a 2D context window. Typically, multiple TDNN units (designated as ψ in Fig. 2.6) are organized into layers within the architecture. Layers closer to the input generally capture finer details, while those further from the input tend to model more abstract features, as they aggregate inputs from wider context windows.

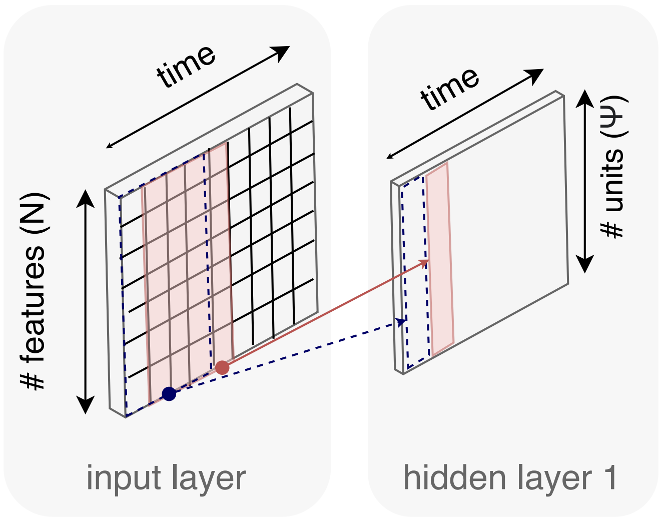


Fig. 2.6 A time-shifted window over the input data stream for a TDNN network.

The number of weights for each unit is given by the number of features and the window size. The key idea is based on sharing the weights, as the contextual window moves along the input sequence. In the backpropagation training (see Sec. 2.6), the weight update is then computed as an average of suggested updates for all window positions and thus the shift-invariance is achieved.

In Fig. 2.7, there are two design choices illustrated: 1) for time t, the contextual window may include future time-steps as well as past time-steps (here <t-6, t+3>); 2) the number of time-steps can be subsampled in order to reduce the number of operations [100].

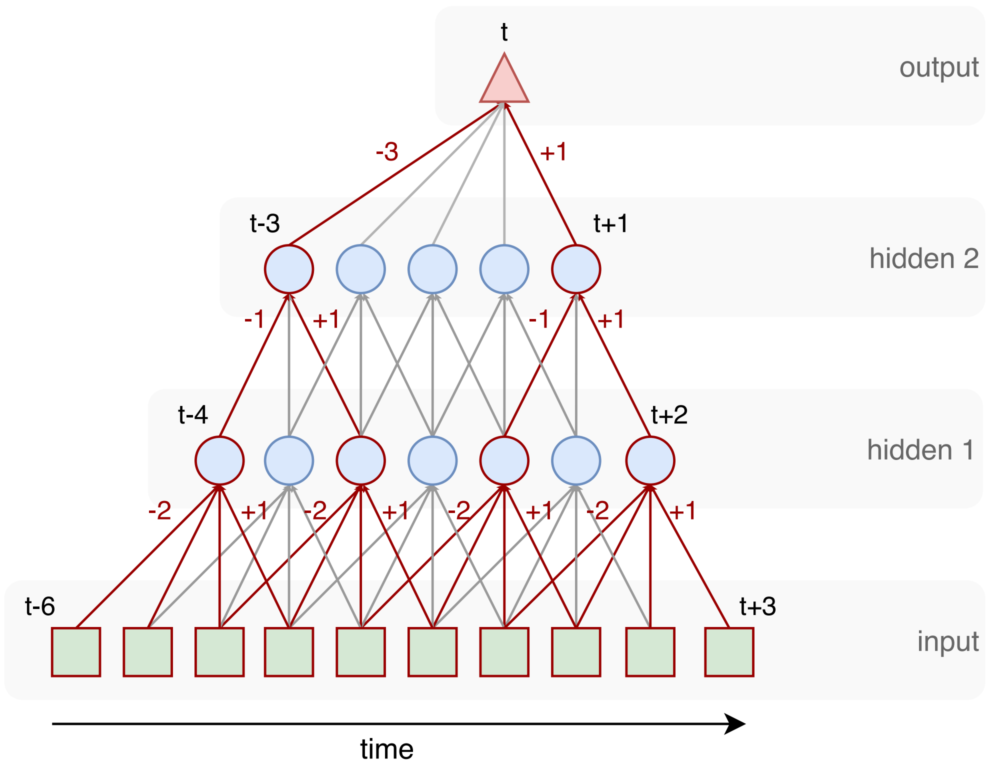


Fig. 2.7 The TDNN principle with subsampling (red) and without subsampling (red + gray).

2.2.3 Convolutional Network (CNN)

Following the theory of receptive fields in the human visual cortex [59], there was the idea of so-called *neocognitron* in [41], which can be considered the origin of the CNN architecture. However, the standard reference is from [75], when a pioneering 7-level CNN was applied to classify handwritten digits on bank checks in the USA. The developement was inspired by the TDNN theory (Sec. 2.2.2) and even the principle is identical with certain settings[3].

The most common application is the visual imagery analysis and the motivation for using CNNs instead of the MLP (Sec. 2.2.1) has two points:

- *Parameters reduction* - using the MLP, a typical 256x256 image on the input results in $56,000 \cdot \psi$ parameters, where ψ is the number of units in the following hidden layer. In CNN the weights are cleverly shared and thus their quantity is significantly reduced.
- *Consideration of contextual dependencies* - there is clearly a relationship between space and pixels in images. Two nearby pixels are much more correlated than two distant pixels and the CNN approach takes this fact into account.

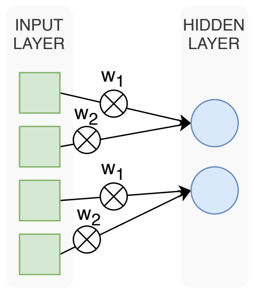
[3]The approach is identical to the TDNN (Sec. 2.2.2) in case of stride $= 1$ (1D data).

Compared to the MLP approach, many synapses are actually removed and the decision which synapses remain is based on our understanding of the space-importance property.

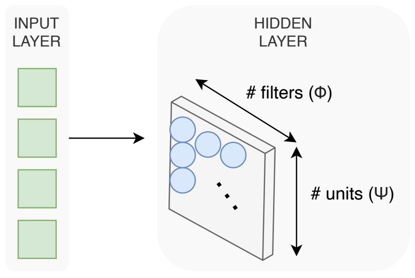
Fig. 2.8 explains the principle of the convolution on a 1D input. There are several hyper-parameters to be set when a CNN is designed:

- *Filter size* - the set of shared weights is called a *filter*. In Fig. 2.8a, the filter consists of w_1 and w_2 and thus its size equals 2.
- *Stride* - the step size when moving the filter. In Fig. 2.8a, stride = 2.
- *Padding* - optionally, the input space can be padded by zeros around its boundaries. There is no padding in the example in Fig. 2.8a.
- *Number of filters* - adding more filters is illustrated in Fig. 2.8b. Each filter is then defined by its own set of weights and is shared by connections to ψ units in the following layer.

The number of units ψ in the following layer depends on stride and padding.



(a) CNN (1D data): a single filter.



(b) CNN (1D data): multiple filters

Fig. 2.8 The CNN (1D data) principle.

The most common usage of the CNN architecture is for the classification (or generally the analysis) of images. An image of width W and height H is defined by a 2D matrix and thus a 2D convolution is applied. The concept is very similar to the 1D case, here we just have 2D filters ($w \times h$) and as a result there is a 3D shaped hidden layer generated (width ψ_1, height ψ_2, number of filters ϕ).

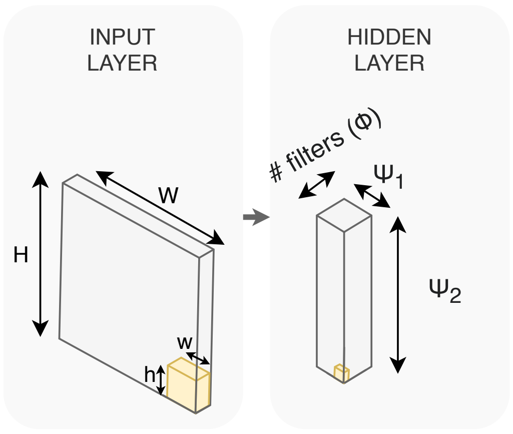


Fig. 2.9 CNN principle (2D data). The yellow-marked $w \times h$ filter in the input layer corresponds to the little yellow cube in the hidden layer.

Besides the size, images are often described by several channels (usually R, G, B) as well. This might seem to be the third dimension on the input, however, as the filter is not slided along the channels (it is only slided in the width and height dimensions), the convolution is still considered 2D.

The 3D convolution can be applied to a video for example. Alongside the width and height dimensions of the frames there is additionally the *time* dimension. Unlike the channels case, ordering in time has a meaning for the network to capture, therefore we slide the filter in three dimensions and the hidden layer is 4-dimensional here (width ψ_1, height ψ_2, time ψ_3 and number of filters ϕ). As in the 1D case, the number of filters is chosen and the rest depend on the *stride* and *padding* parameters.

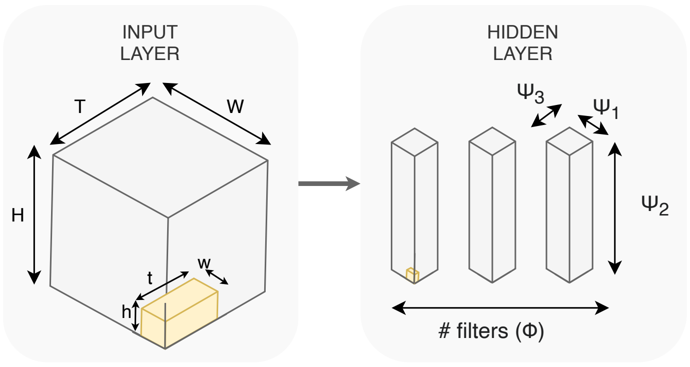


Fig. 2.10 CNN (3D data). For example a video (width, height, time).

Convolutional layers are commonly combined with the *max-pooling* mechanism and finished by standard feedforward layers (Sec. 2.2.1).

Pooling [132].

This method is typically applied consequently to convolutional layers (see Sec. 2.2.3) in CNNs, in order to reduce the dimensions of the feature maps. The most popular type is called max-pooling and its principle is illustrated in Fig. 2.11.

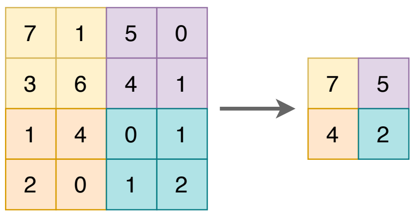


Fig. 2.11 Principle of the max-pooling method.

2.2.4 Residual Network

The *Residual Network (ResNet)* architecture, commonly referred to as *skipping connections* [45], was initially introduced to address the *vanishing gradient problem*. In deep structures with numerous hidden layers, gradient updates may exponentially decrease, leading to poor updates in early layers. The ResNet design tackles this issue by incorporating alternative paths for the backpropagation algorithm through added connections that skip subsequent layers.

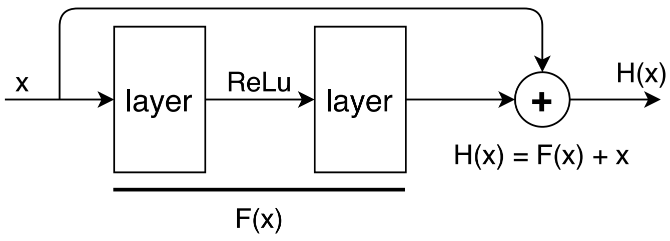


Fig. 2.12 Residual Network (ResNet) – skipping connection – principle.

Later, this approach demonstrated its effectiveness in the forward pass and is widely utilized, such as in Transformer architectures combined with Attention layers (refer to Sec. 2.4). The underlying concept is illustrated in Fig. 2.12. The addition operation is a form of skip connection, also known as a shortcut connection.

2.3 Recurrent Architectures

Unlike the restricted direction of the information flow in feedforward structures, recurrent networks have loops - outputs of units with a specific state can generally be used as inputs to cells in the same or even previous layers.

Regarding the goals of this work, there are several interesting architectures (special kinds of RNNs) described below, but first, we start with the common and nowadays mostly used RNN approach based on the idea from 1986 [113]. Since then, the crucial breakthroughs related to recurrent neural networks and context modeling are:

- *the backpropagation-through-time algorithm* [36] - capability of learning using the standard backpropagation algorithm (Sec. 2.6);
- *the LSTM cell* [55] - capability of learning long-term dependencies (Sec. 2.3.1);
- *using RBM for weights initialization* [49] - capability of learning for deep RNNs;
- *Transformer* [126] - This novel approach to sequence processing has outperformed recurrent neural networks across all domains (further discussed in Section 2.4).

In a form of a directed graph along a (usually temporal) sequence, such a network is capable of dealing with contextual dependencies in data. In contrast to the TDNN approach (Sec. 2.2.2), the ability of processing input sequences of a variable length is done in a much more sophisticated way. The loops allow to work with an internal state (memory) for each cell. Typical RNN tasks generally differ one from each other as illustrated in Fig. 2.13:

(a) *one-to-one* - a fixed-sized input to a fixed-sized output, no need of RNN (e.g. image classification);

(b) *one-to-many* - sequence output (e.g. image captioning - an image is taken as the input and the systems outputs a sentence);

(c) *many-to-one* - sequence input (e.g. sentiment analysis - a sentence is classified to be of a positive or negative sentiment);

(d) *many-to-many* - sequence input and output (e.g. machine translation);

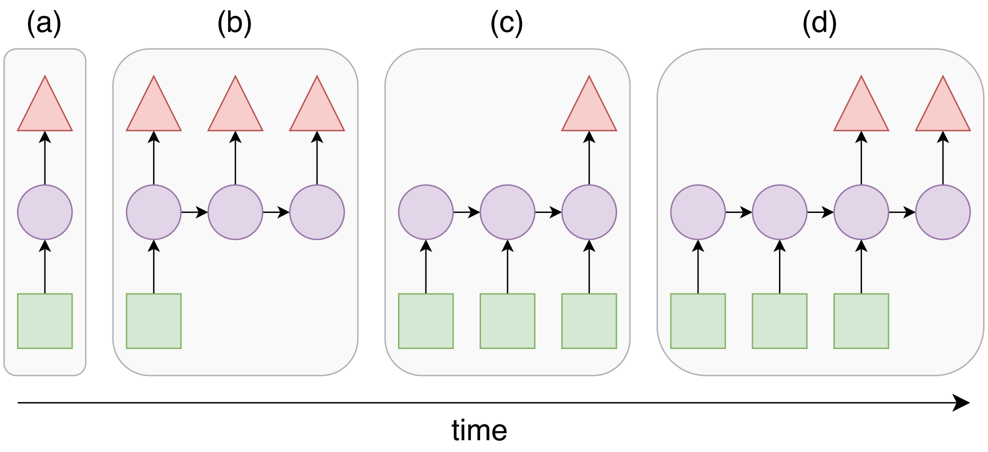


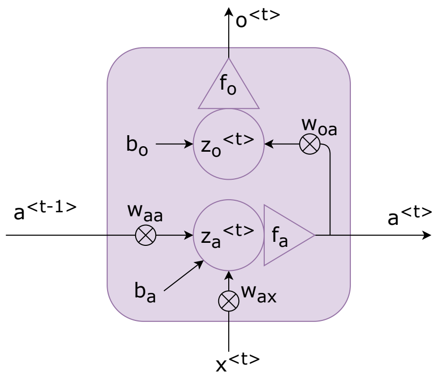
Fig. 2.13 Sequence data - task types. Figure is inspired by [69].

Assuming that the processed sequences are temporal (context-dependent over time) and so sequential samples are indexed by <t>, the default RNN cell is illustrated in Fig. 2.14a [4]. For each timestep t, the cell activation $a^{<t>}$ and the output $o^{<t>}$ are computed as follows:

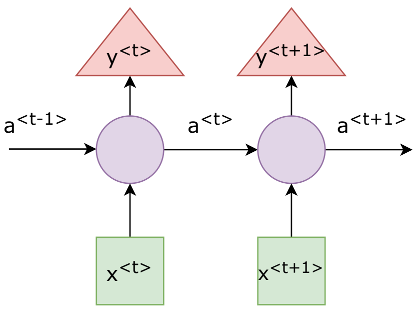
$$a^{<t>} = f_a(w_{aa} \cdot a^{<t-1>} + w_{ax} \cdot x^{<t>} + b_a) \tag{2.6}$$

$$o^{<t>} = f_o(w_{ya} \cdot a^{<t>} + b_o) \tag{2.7}$$

where w_{aa}, w_{ax}, w_{ya}, b_a, b_o are parameters that are shared over time and f_a, f_o are chosen transfer functions (see Sec. 2.2.1).



(a) Default RNN - cell body.



(b) Information flow in the default RNN.

Fig. 2.14 Default RNN - cell body and data flow.

In general, RNN models are mostly used in the fields of natural language processing and speech recognition. A particular model is designed for different applications like for example machine translation, phoneme recognition or sentiment analysis. The nature of the RNN approach allows processing inputs of any length and as the weights are shared over time, the model size does not increase with the input size. However, several special versions have been developed over the years (Sec. 2.3.1 - 2.3.3) in order to deal with the main drawbacks of RNNs in general:

- RNNs suffer from the vanishing and exploding gradient problems, which occur during the training process. When backpropagating through time, gradients can either become extremely small (vanish) or extremely large (explode). This makes it difficult for RNNs to learn long-range dependencies in sequences. This has been addressed by the idea of gates in the cell body (LSTM - Sec. 2.3.1 and GRU - Sec. 2.3.2).
- Each time step depends on the previous time step, leading to a sequential bottleneck that can slow down training and inference, especially for long sequences.
- The default RNN version cannot consider any future input for the current state - a *bidirectional* version (BRNN - Sec. 2.3.3) can.
- The computation is notably sluggish, posing challenges in training with extensive datasets, and the computational time scales directly with the length of the sequence.

2.3.1 Long Short-Term Memory (LSTM)

With respect to the number of layers, the multiplicative gradient can be exponentially decreasing/increasing. This phenomena is known as the vanishing/exploding gradient problem (see Sec. 2.6.2) and it makes the default RNN incapable of capturing long term dependencies in the data sequence.

Originally introduced in [55], there are so-called *gates* inside the cell body that filter the information passing through. In the LSTM cell (Fig. 2.15), there is a cell state $c^{<t>}$ working like a conveyor belt that affects the activation $a^{<t>}$ and is regulated by these gates:

- *forget gate* $g_f^{<t>}$ - decides what information is thrown away from the cell state using the sigmoid transfer function (Fig. 2.4), $f_f(\cdot) = \sigma(\cdot)$;

$$g_f^{<t>} = \sigma(w_f \cdot [a^{<t-1>}, x^{<t>}] + b_f) \tag{2.8}$$

- *input gate* $g_i^{<t>}$ - decides what information is stored in the cell state using the sigmoid transfer function (Fig. 2.4), $f_i(\cdot) = \sigma(\cdot)$;

$$g_i^{<t>} = \sigma(w_i \cdot [a^{<t-1>}, x^{<t>}] + b_i) \tag{2.9}$$

- *candidate gate* $g_c^{<t>}$ - creates new candidate values that could be added to the cell state and so together with the *input* gate decides about the update of the cell state using the hyperbolic tangent as the transfer function (Fig. 2.4), $f_c(\cdot) = \tanh(\cdot)$;

$$g_c^{<t>} = \tanh(w_c \cdot [a^{<t-1>}, x^{<t>}] + b_c) \tag{2.10}$$

- *output gate* $g_o^{<t>}$ - decides what information is sent to the output using the sigmoid transfer function (Fig. 2.4, $f_i(\cdot) = \sigma(\cdot)$) and combined with the cell state generates the activation of the cell (Eq. 2.13);

$$g_o^{<t>} = \sigma(w_o \cdot [a^{<t-1>}, x^{<t>}] + b_o) \tag{2.11}$$

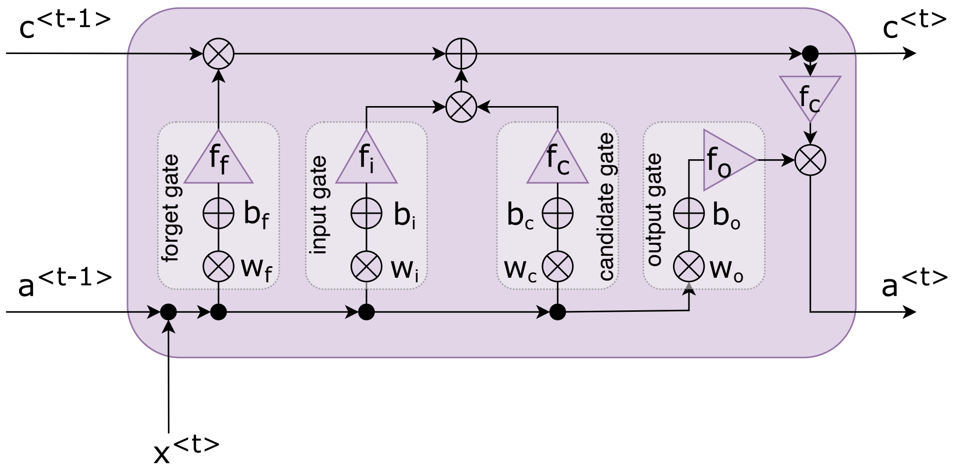


Fig. 2.15 Long Short-Term Memory (LSTM) cell.

Finally, a new cell state $c^{<t>}$ and a new activation value $a^{<t>}$ are expressed as follows:

$$c^{<t>} = g_f^{<t>} \times c^{<t-1>} + g_i^{<t>} \times g_c^{<t>} \tag{2.12}$$

$$a^{<t>} = g_o^{<t>} \times \tanh(c^{<t>}) \tag{2.13}$$

2.3.2 Gated Recurrent Unit (GRU)

There have been several experiments over the years slightly adjusting the body of the LSTM cell - a good comparison of those versions is provided in [43]. The most popular modified version of the general LSTM template is called GRU [24]. It combines the forget and input gate into a single *update* gate and merges the cell state with the hidden activation state. As illustrated in Fig. 2.16, the gates are:

- *reset gate* $g_r^{<t>}$ - decides how much of the past information is forgotten using the sigmoid transfer function (Fig. 2.4), $f_r(\cdot) = \sigma(\cdot)$;

$$g_r^{<t>} = \sigma(w_r \cdot [a^{<t-1>}, x^{<t>}] + b_r) \tag{2.14}$$

- *update gate* $g_u^{<t>}$ - the update to the activation of the cell is expressed as follows ($f_u(\cdot) = \sigma(\cdot)$):

$$g_u^{<t>} = \sigma(w_u \cdot [a^{<t-1>}, x^{<t>}] + b_u) \tag{2.15}$$

- *candidate gate* $g_c^{<t>}$ - the new candidate values are given as follows ($f_c(\cdot) = \tanh(\cdot)$):

$$g_c^{<t>} = \tanh(w_c \cdot [g_r^{<t>} \times a^{<t-1>}, x^{<t>}] + b_u) \tag{2.16}$$

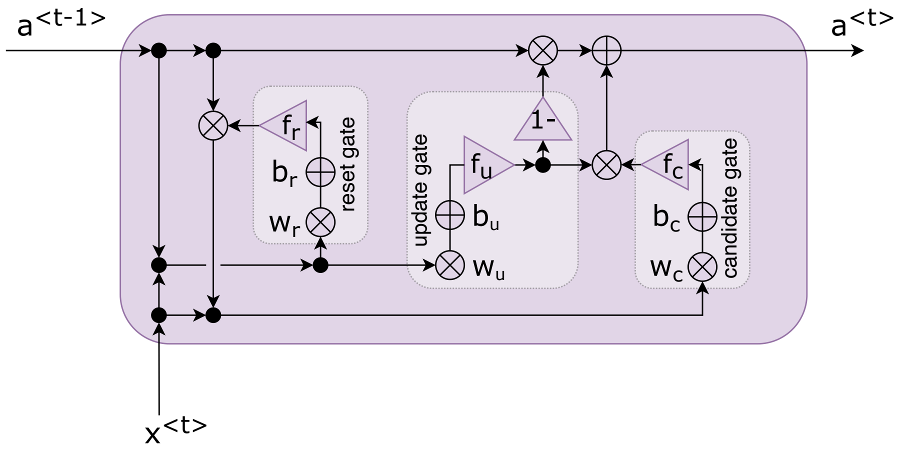


Fig. 2.16 Gated Recurrent Unit (GRU) cell.

Finally, the new activation value is expressed as follows:

$$a^{<t>} = (1 - g_u^{<t>}) \times a^{<t-1>} + g_u^{<t>} \times g_c^{<t>} \tag{2.17}$$

Both, LSTM and GRU versions, have been widely used in parallel. In general, the LSTM is believed to work better for larger datasets, while the GRU is simpler and so usually faster, but those conclusions might differ for specific problems. The general learning procedure for RNNs is described in Sec. 2.6.

Finding a way of learning deep RNN networks (based on RBM pre-training - Sec. 2.3.5) was the key step to make them the SoTA in sequential learning. The next significant improvements came with including the *attention* mechanism (Sec. 2.4.1), using RNNs as a part of Generative Adversarial Networks (GANs - Sec. 2.5) and also using the *bidirectional* architecture.

2.3.3 Bidirectional Network (BRNN)

Even though it is not natural from the human point of view, as it is not possible for us to learn from future events, artificial systems can take advantage of it as long as they use the standard learning procedure based on offline datasets (all data collected beforehand).

The theory published in [119] can be applied to all previously described RNN cell types (default, LSTM, GRU). As shown in Fig. 2.17, there are forward (fed in a normal time order) and backward (fed in a reverse order) layers combined into a single network. The outputs of the two layers are concatenated (or summed - depends on the implementation) at each time step and so the network has both backward and forward information about the sequence.

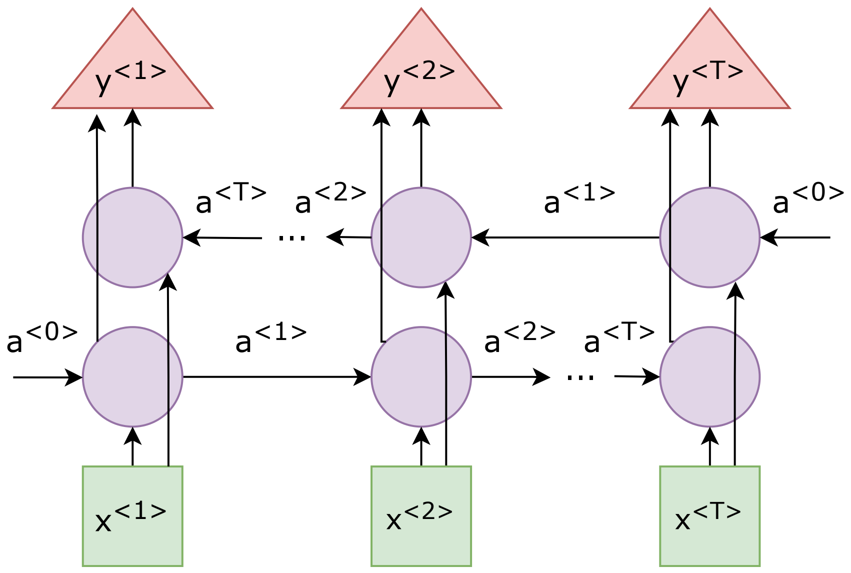


Fig. 2.17 Bidirectional Recurrent Neural Network (BRNN) - the purple cells can be e.g. LSTM or GRU.

2.3.4 Time-Distributed Layers

A time-distributed layer [124] is a type of layer in recurrent neural networks (RNNs) or 1D convolutional neural networks (CNNs) that applies the same layer configuration to every temporal slice of the input sequence independently. This means that the layer's parameters are shared across different time steps. Time-distributed layers are often used when dealing with sequential data, such as time series or sequences of text. Basically, the time-distributed layer serves as a wrapper.

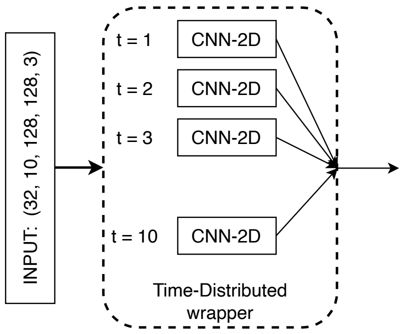


Fig. 2.18 Illustration of the time-distributed wrapper.

2.3.5 Special Recurrent Structures

Besides the standard RNN architectures based on the default RNN cell and its composition into a network (Fig. 2.14), there are several special methods that can be considered recurrent. Regarding the goals of this work, learning their structures and functionalities can be useful.

Hopfield Network [56]

In the Hopfield network, neurons are connected to every other neuron. There are no layers, as the neurons are considered input before the training, hidden during it and output afterwards.

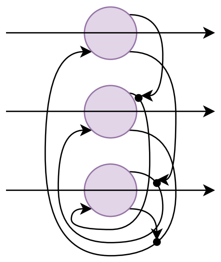


Fig. 2.19 A Hopfield network of three units.

If the connections are symmetric ($w_{ij} = w_{ji}$), there is so-called global *energy* function E (Eq. 2.18) and each configuration of the network is mapped to a certain energy value.

$$E = -\sum_{i<j} s_i \cdot s_j \cdot w_{ij} - \sum_i b_i \cdot s_i \tag{2.18}$$

where $s_i \in \{-1, 1\}$ is the binary output of i^{th} unit, b_i is its bias and w_{ij} is the weight of its connection to the j^{th} unit. The weight update is performed by the *Hebbian rule*: $\Delta w = s_i \cdot s_j$ [47] and is usually done asynchronously (can be done synchronously in theory). It is proven that as the network learns a pattern, its energy decreases and always settles in a local minima of the energy function. This feature makes the Hopfield network capable of memorizing patterns and even of reconstructing the learned pattern when given just a part of it. Therefore, it can be used as a content-addressable (associative) memory with the capacity limited to $0.15N$ for N being the number of units.

Boltzmann Machine [2]

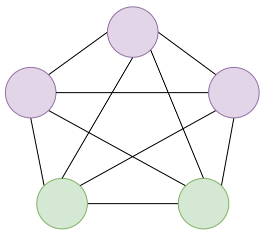
The structure of the Boltzmann machine (Fig. 2.20a) is identical to the *Hopfield* network, however, the units decisions about whether to be on or off are stochastic [50]. This makes the algorithm possibly capable of escaping from a poor local optima while searching for good solutions. The energy function of state vector v is defined as in Eq. 2.18 ($E(v) = E$) and the probability of the *Boltzmann equilibrium* (or stationary distribution) is given as the energy relative to energies of all possible binary state vectors:

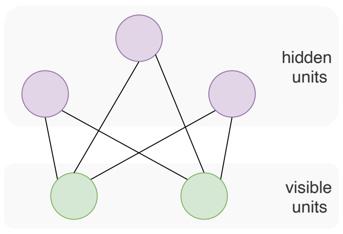
$$P(v) = \frac{e^{E(v)}}{\sum_u e^{-E(u)}} \tag{2.19}$$

Boltzmann machines are used for two different computational problems:

1. a *search* problem - weights remain fixed and represent the cost function of the optimization problem;

2. a *learning* problem - weights are adjusted (using $\partial E(v)/\partial w_{ij} = -s_i^v \cdot s_j^v$) so that a set of binary data vectors is a good solution to the optimization problem defined by the weights.


(a) Boltzmann Machine



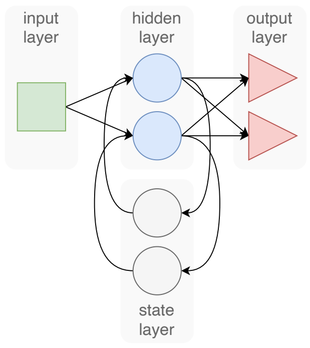
(b) Restricted Boltzmann Machine

Fig. 2.20 A (Restricted) Boltzmann Machine example.

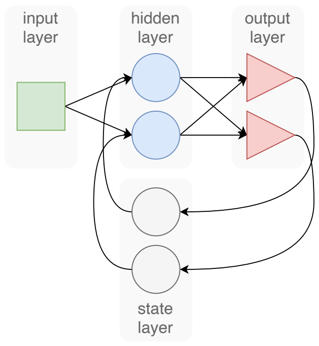
The restricted version - RBM [120], shown in Fig. 2.20b, consists of the visible layer and the hidden layer with no connections between units of the same layer. During the learning phase [49], visible and hidden units are iteratively (layer by layer) updated until the reconstruction of the visible units is close enough to the original. Then the output of the hidden layer can be used as the input to another Boltzmann machine. Learning one hidden layer at a time is a very effective way of getting suitable weights initialisation for deep neural networks, as highest level features are typically much more useful for classification than raw data vectors.

Elman/Jordan Network [36], [65]

These two structures are commonly known as simple recurrent networks (SRN). As shown in Fig. 2.21, they include a state layer containing context nodes that maintain memory of the prior values and thus the application to sequential data is allowed [64]. In the case of the Elman network (Fig. 2.21a) the state layer is fed from the hidden layer and in the case of the Jordan network (Fig. 2.21b), the output layer is stored into the state layer. Multiple state layers can possibly be subsequently added and the learning is done by the backpropagation algorithm (the BPTT version, Sec. 2.6).



(a) Elman network.



(b) Jordan network.

Fig. 2.21 Simple recurrent networks (SRN).

2.4 Transformers

The concept of Transformers has become a prominent paradigm in machine learning, particularly in Natural Language Processing (NLP). Introduced in the groundbreaking paper *Attention is All You Need* [126], this architecture relies on the *attention* mechanism to deliver remarkable improvements in the performance of deep learning models (not only) for text-related applications. *Attention* had been previously employed on top of RNN layers, however, this paper demonstrated an increase in performance when implemented independently and directly processing the input sequences. The primary benefits over previous approaches are:

- *Parallelization and efficient processing of extensive datasets:* By employing the practical matrix multiplication provided by the *attention* mechanism, Transformers enable efficient parallel processing of input sequences. This scalability makes Transformers highly efficient at handling extensive datasets and training on powerful hardware.
- *Long-range dependencies:* Unlike traditional sequential models like RNNs, Transformers can capture long-range dependencies in input sequences effectively. The *self-attention* mechanism enables each output to depend on all positions in the input sequence, overcoming the vanishing gradient problem associated with RNNs.
- *Transfer learning:* Due to the robust embedding capabilities of the *attention* mechanism, a model trained on one task can effortlessly adapt to a second, related task. This intelligent weight initialization greatly enhances the learning process for the target task, particularly when labeled data for the target task is limited.

2.4.1 Attention Mechanism

As outlined in the preceding section listing the advantages of the Transformer architecture, obviously the key ingredient driving the ground-breaking performance is the *self-attention* mechanism. As we show later, it can be expanded to multiple domains, but it is well and intuitively descriptable on text. So, its functionality can be summarized as follows [35]:

"While processing a word, self-attention enables the model to focus on other closely related words in the input sentence."

To illustrate this with an example, let's consider two sentences:

(A) *The dog ate the cheese, because **it** was hungry.*

(B) *The dog ate the cheese, because **it** was tasty.*

In the first (A) sentence, the word **it** refers to the dog, whereas in sentence (B), **it** refers to the *cheese*. When the model processes the word **it**, *self-attention* provides the model with additional information about its meaning, enabling it to correctly associate **it** with the relevant context.

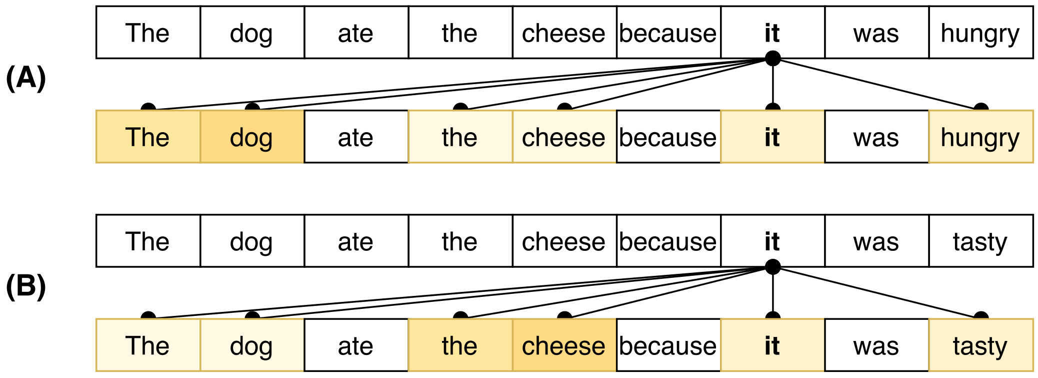

(A)
The dog ate the cheese because it was hungry
The dog ate the cheese because it was hungry
(B)
The dog ate the cheese because it was tasty
The dog ate the cheese because it was tasty

Fig. 2.22 Example of Self-Attention. Darker colors represent higher attention scores.

At this point, let's clarify some related terms while keeping the explanations within the text domain. First, we distinguish between 1) *self-attention*, which involves looking at how words within the same sentence relate to each other, and 2) *attention*, where we calculate relations of two different sentences. However, to be honest, term *attention* is frequently used as a shorthand, referring to *self-attention*. Additionally, in the following, we define terms *basic–* and *trainable– self-attention*, along with the concept of *multi-head attention.*

Basic self-attention. Self-attention operates as a sequence-to-sequence process, where a sequence of vectors $x_1, x_2, ..., x_L$ is inputted, yielding a corresponding sequence of vectors $y_1, y_2, ..., y_L$ as output [15]. Each vector in this process has a dimensionality of k. To generate the output vector y_i, the self-attention operation simply takes *a weighted average* across all input vectors, as given in Eq. 2.20.

$$y_i = \sum_j \omega_{ij} \cdot x_j \tag{2.20}$$

where j indexes the entire sequence, and the weights sum to one over all j. The weight matrix ω_{ij}, as defined in Equation 2.22, is not a parameter in the traditional neural network sense. Instead, it is derived from a function involving x_i and x_j and is denoted as the *attention score*. There are multiple options, but the simplest choice for this function is the dot product:

$$\omega'_{ij} = x_i^T \cdot x_j \tag{2.21}$$

The dot product essentially involves multiplying corresponding pairs of numbers and then summing them up. These numbers can be viewed as individual features within the word embedding. When the paired features (e.g., x_{i2} and x_{j2}) share the same polarity (both positive or both negative), the product is positive, increasing the final summation. Conversely, the summation is reduced when the two paired features have different polarities and the product is negative. Moreover, the magnitude of the numbers (i.e., the significance of the features) directly influences their contribution to the overall summation. This characteristic renders the dot product a solid score estimator, reflecting the alignment between two vectors.

As illustrated in Fig. 2.23, x_i represents the input vector at the same position as the current output vector y_i. When moving to the subsequent output vector, a completely new set of dot products is computed, resulting in a distinct weighted sum (different *attention score*).

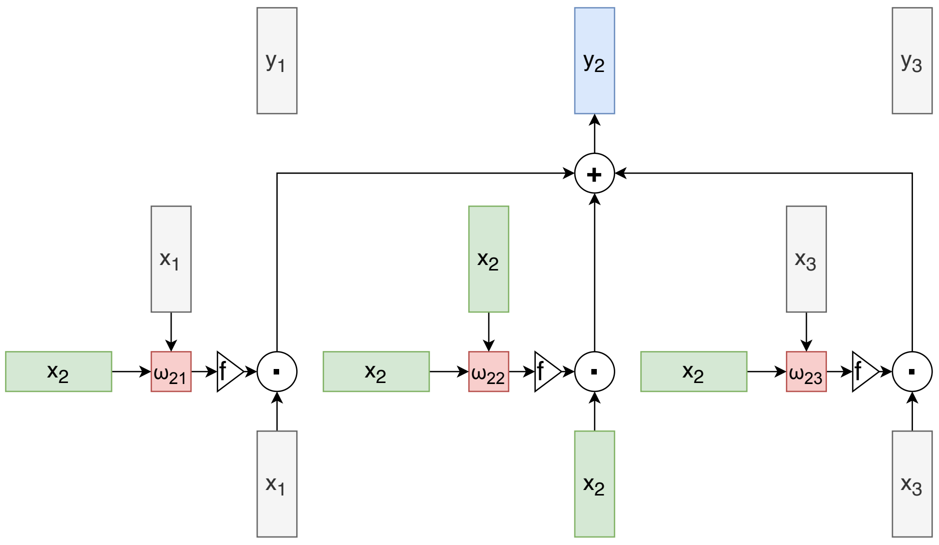


Fig. 2.23 Basic self-attention – processing vector x_2 and yielding vector y_2, where $f()$ stands for the *softmax* function as given in Eq. 2.22.

As the dot product yields a value spanning from negative to positive infinity, to confine these values within the [0, 1] range and guarantee their summation to 1 across the entire sequence, the *softmax* function is applied (Eq. 2.22).

$$\omega_{ij} = \frac{exp\omega'_{ij}}{\sum_j exp\omega'_{ij}} \tag{2.22}$$

Remarkably, the mechanism of *self-attention* is the only operation within the entire *Transformer* architecture that facilitates information exchange between vectors. Every other operation is applied individually to each vector in the input sequence, with no interactions with the others.

Trainable self-attention. The basic version might be effective if we had perfect embedding vectors that already encapsulate context dependencies and task-based relations. However, in practise, we treat individual features as parameters in our model and these embeddings are learned through training. To achieve this, three additional components, denoted as W_κ, W_q, and W_v, are introduced. Each of them represents a $k \times k$ matrix of trainable parameters, and is responsible for an independent linear transformation of vector x_i. As depicted in Fig. 2.24, these transformations play three distinct roles within the mechanism:

- *Query* $q_i = W_q \cdot x_i$ is compared to every other vector to determine the attention scores for its own output y_i;
- *Key* $\kappa_i = W_\kappa \cdot x_i$ is compared to every other vector to establish the attention scores for output y_j;
- *Value* $v_i = W_v \cdot x_i$ is used as a component of the weighted sum to compute each output vector once the attention scores have been determined.

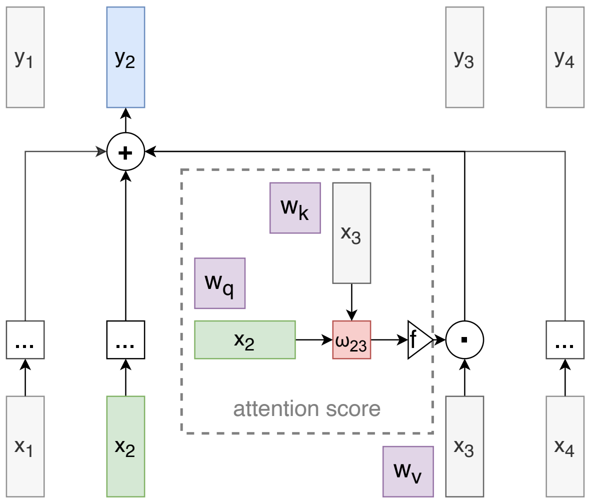


Fig. 2.24 Trainable version of self-attention with *query*, *key* and *value* transformations.

The self-attention operation is then expressed by Eq. 2.23. This gives the mechanism some controllable parameters, and allows it to modify the incoming vectors to suit the three roles they must play. Additionally, to address the sensitivity of the *softmax* function to large input values, which may impede gradient flow and slow down training, the dot product is scaled back by $\sqrt{k}$ – a factor by which the increase in dimension enhances the length of the average vectors [15].

$$y_i = \sum_j softmax(\frac{q_i^T \cdot \kappa_j}{\sqrt{k}}) \cdot v_j \tag{2.23}$$

Attention scores. In addition to a well-performing model, as a bonus after training, we obtain the attention scores. These scores provide valuable insights into how the model arrived at its decisions and learned relationships among individual input vectors.

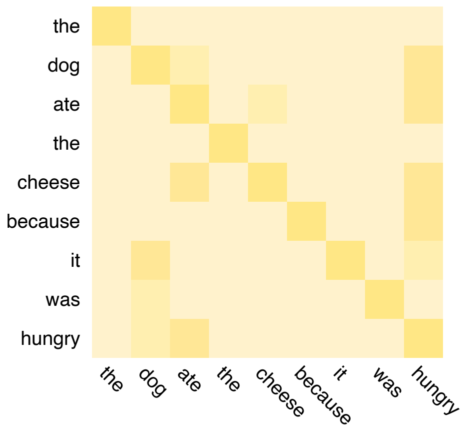


Fig. 2.25 Fictive example of attention scores. Darker colors indicate higher attention.

Returning to the example sentence from the beginning of this section, a fictive illustration of such scores is depicted in Figure 2.25. The attention matrix is always of size $L \times L$, where L is the length of the input sequence (the number of words in the sentence). Thus, the matrix provides a score for each combination of input vectors (words), with rows representing the words attending and columns representing the words receiving attention.

Multi-head self-attention. A word can have multiple relations with various meanings within a sentence. In our previous example, *dog* represents the entity eating, *cheese* denotes the object being consumed, and *hungry* explains the motivation behind the act of eating. In a single *self-attention* operation, all this information just gets summed together. Inputs corresponding to *dog* and *cheese* can influence the output for *ate* by different amounts, depending on their dot-product with *ate*, but they cannot influence it in different ways. And this is the reason why a little more flexibility is needed here [15].

To enhance the discriminative capability, multiple self-attention mechanisms can be employed in parallel, each featuring distinct matrices W_q^h, W_k^h, W_v^h, representing so-called *attention heads*. As depicted in Fig. 2.26, for input x_i, each attention head generates a unique output vector y_i^h. These outputs are concatenated and subsequently subjected to a linear transformation, using weights W_o, to change the dimensionality back to k. To implement efficient multi-head self-attention, each head receives low-dimensional keys, queries, and values. For instance, with dimensionality $k = 256$ and number of attention heads $H = 4$, the input vectors are projected to 64-dimensional sequences through a 256×64 matrices for keys (k_i^h), queries (q_i^h), and values (v_i^h).

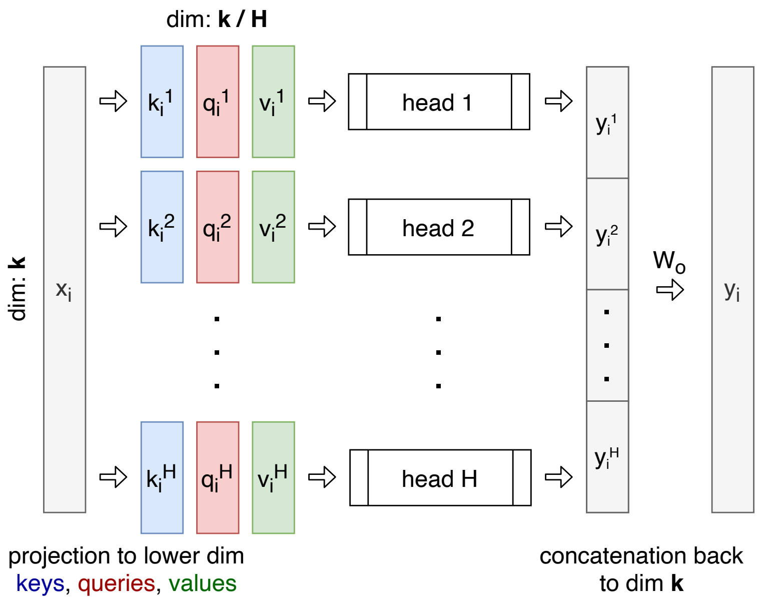


Fig. 2.26 Multi-head attention - projection to lower dimensionality and back.

This requires $(3 \times H)$ matrices, each of size $k \times \frac{k}{H}$. In total, this results in $(3 \times H) \times \frac{k \times k}{H} = 3k^2$ parameters for computing inputs to the multi-head self-attention, aligning with the parameter count of the single-head self-attention. The only difference is in using of the matrix W_o at the output of the multi-head self-attention, contributing k^2 additional parameters compared to the single-head version. Nevertheless, the necessity of employing W_o is a subject to discussion, especially in scenarios with subsequent feedforward layers.

Expansion beyond textual domain. Despite being initially illustrated using a textual example, the attention mechanism has proven beneficial across multiple domains. Fig. 2.27 showcases its utilization to embeddings based on image pixels. For an extensive exploration of attention's applications, a well-structured survey is available in [22].

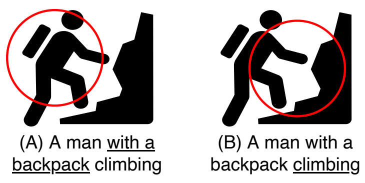

(A) A man with a backpack climbing
(B) A man with a backpack climbing

Fig. 2.27 Illustration of the attention mechanism applied on image pixels (red circle $\sim$ focus).

2.4.2 Original Transformer

The original *Transformer* architecture, initially presented in [126], can be decomposed into separate blocks, as depicted in Fig. 2.28. Each of these blocks will be explained individually in this section. Subsequently, we will delve into an exploration of the most popular architectures, categorizing them into three groups: 1) *Encoder only*, exemplified by BERT or Wav2Vec (Sec. 2.4.3); 2) *Decoder only*, embodied by the GPT and LLM family (Sec. 2.4.4); and 3) *Encoder-Decoder models*, such as T5, SpeechT5, or other multi-modal structures (Sec. 2.4.5).

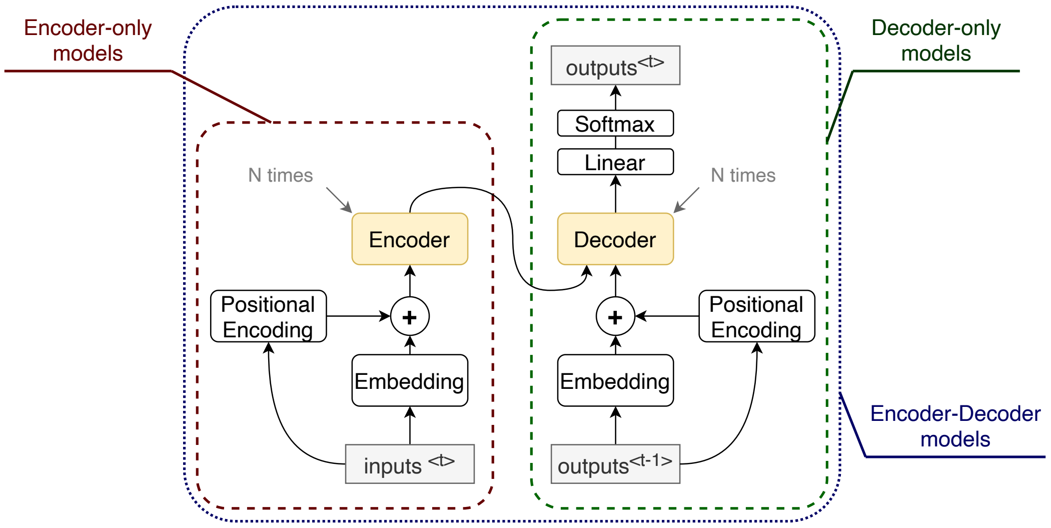


Fig. 2.28 Transformer architecture – decomposition into *Encoder* and *Decoder* parts.

Encoder. The encoder is represented by a standard Transformer block (Fig. 2.29) that typically follows this structure: self-attention layer, layer normalization, feedforward layer (applied independently to each vector), and another layer normalization.

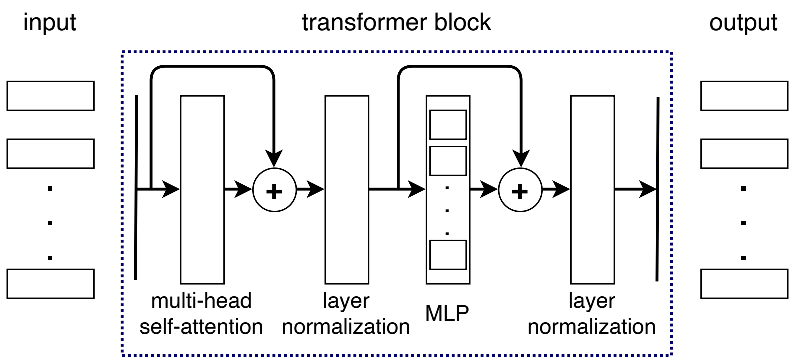


Fig. 2.29 Transformer block - *Encoder*.

Residual connections surround both the self-attention and feedforward layers before normalization. These components ensure the integration of self-attention with local feedforward operations, enhanced by normalization and residual connection. The MLP layer is sometimes substituted by a *Time-Distributed* layer (see Sec. 2.3.4).

Decoder. The decoder block closely resembles the default Transformer block, with a crucial distinction of incorporating an encoder-decoder attention mechanism. In this scenario, attention is applied to assess the relevance between two distinct inputs: 1) the encoded input at time t, utilizing keys and values, and 2) the previous model output at time $(t-1)$, employing its queries. Further details on this mechanism is discussed below.

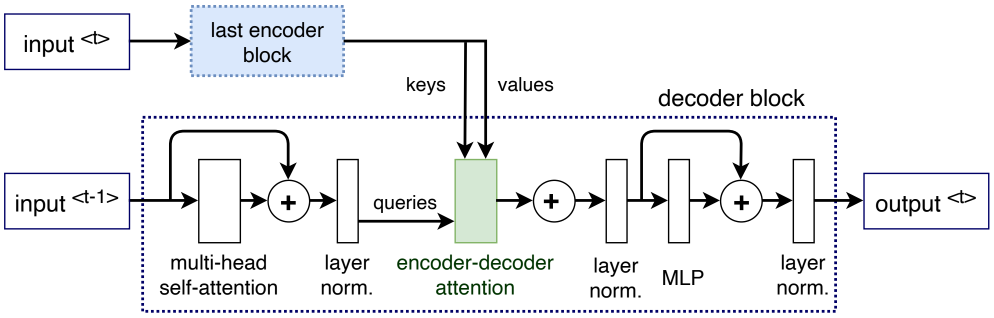


Fig. 2.30 Transformer - Decoder block.

Embedding and positional encoding. Unlike images where numerical values are inherent in pixel values, for textual inputs, each word (token) is typically represented by a single vector of fixed dimension k encapsulating its meaning. Alongside this embedding, encoding the positional information of words is needed, as self-attention views its input as a set rather than a sequence. If we permute the input sequence, the output sequence will be identical, except also permuted. This permutation-invariance and the ability to process all data in parallel is the major advantage of Transformers over RNNs. However, this characteristic also implies that position information needs to be incorporated back "manually". The authors of [126] devised a clever method involving trigonometric functions as follows:

$$PE_{(pos,2i)} = \sin(\frac{pos}{10000^{\frac{2i}{k}}}) \tag{2.24}$$

$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{\frac{2i}{k}}}) \tag{2.25}$$

Here, pos represents the position of the word in the sentence, k is the embedding dimensionality, and i is the index value in the embedding vector. This way, the *encoding* vector is formed and then typically added to the default embedding vector.

Training phase. Transformers work slightly differently during training and inference.During the training phase, assuming the utilization of both input and target sequences and still assuming the original version, the procedure is as outlined in [35]:

1. The input sequence is converted into embeddings (with position encoding) and fed to the stack of encoders.
2. The stack of encoders processes it and produces a representation of the input sequence.
3. The target sequence is prepended with a *start-of-sentence token*, converted into embeddings (with position encoding), and fed to the stack of decoders.
4. The stack of decoders processes this along with the encoded representation of the input (from encoders) to produce an encoded representation of the target sequence.
5. The output layer converts it into word probabilities and the final output sequence.
6. The loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients for backpropagation.

Inference phase. During inference, only the input sequence is available, and there is no target sequence to pass as input to the decoder. The objective is to generate the target sequence solely from the input sequence. Therefore, the output is generated iteratively in a loop, feeding the output sequence from the previous time-step $(t-1)$ to the decoder until an *end-of-sentence token* is encountered. The distinction from standard sequence-to-sequence models lies in re-feeding the entire output sequence generated at each time-step, rather than just the last generated word.

2.4.3 Encoder-Only Models

Encoder-only models focus on extracting meaningful representations from input sequences without a dedicated decoding mechanism.

Classification Transformer. The fundamental Transformer for sequence classification consists of a series of transformer blocks. The key design choices are in handling input sequences and transforming the final output sequence into a classification. The most common method for this is to employ global average pooling on the final output sequence, followed by the *softmax* function for categorization.

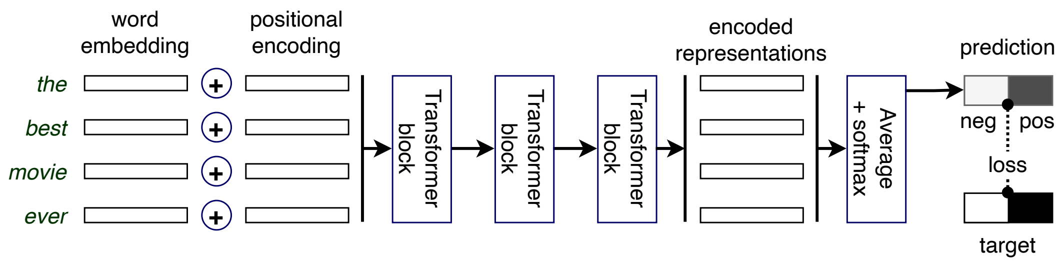


Fig. 2.31 Encoder-only architecture employed for sentiment classification in movie reviews.

Bidirectional Encoder Representation from Transformers (BERT) [31] This architecture published by *Google AI Language* team is basically a stack of Transformer encoders. Unlike traditional language models that process text in a unidirectional manner, BERT employs bidirectional context understanding, considering both the left and right contexts of each word. This allows BERT to capture richer contextual information, making it highly effective for various natural language processing tasks such as sentiment analysis, named entity recognition, and question answering.

BERT's pre-training entails predicting missing words in sentences (*masked language model - MLM*), allowing the model to acquire deep contextualized representations. Additionally, the technique of *next sentence prediction* (NSP) is employed, where the model is presented with a pair of sentences (A and B) to discern if sentence B succeeds A in the corpus, aiding in understanding the relationship between sentences. Fine-tuning on specific tasks further refines its performance and adaptability across diverse applications. Several variants have emerged from the original BERT, including:

- *RoBERTa*, short for "Robustly Optimized BERT Approach," represents an enhanced iteration of the BERT model with notable improvements. Key distinctions include the incorporation of dynamic masking, an increased volume of data points for enhanced model information, the omission of the Next Sentence Prediction (NSP) task, utilization of a larger dataset, and processing with a larger batch size.
- *AlBERT* is larger in terms of the number of parameters. Next, it uses sentence order prediction instead of next sentence prediction. Also, it utilizies parameter sharing between the layers to increase performance.

Wav2Vec 2.0 [7] This ASR framework was developed by Facebook AI. It introduces a self-supervised pre-training approach for learning representations from unlabeled speech data. The model leverages a contrastive learning objective, where it predicts masked speech representations and learns to differentiate between positive and negative samples.

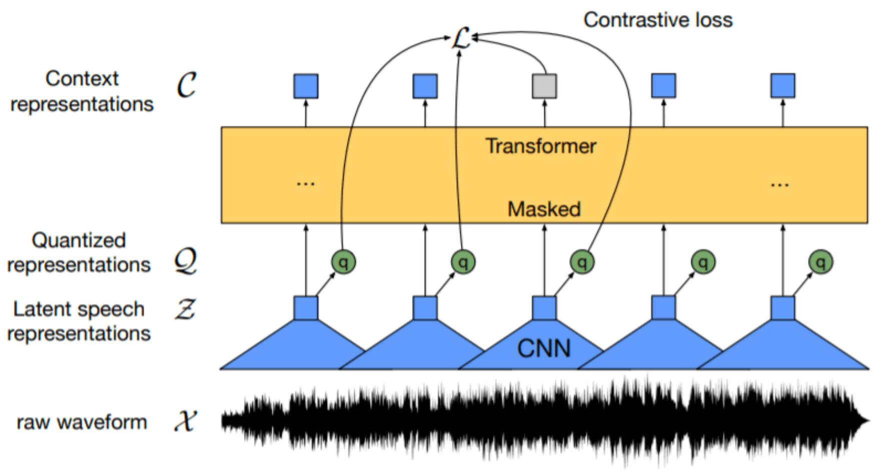


Fig. 2.32 Wav2Vec 2.0 - original figure from [7]. Illustration of learning representations.

Wav2Vec 2.0 achieves state-of-the-art results in ASR by pre-training on a vast amount of data, followed by fine-tuning on smaller, task-specific datasets. The architecture effectively captures contextual information in speech signals, making it robust and adaptable for various speech-related applications, like for example in [138].

2.4.4 Decoder-Only Models

Decoder-only Transformers, including the GPT family, are tailored for sequence generation tasks and are also termed as *auto-regressive* models. Unlike traditional transformers, they exclusively focus on decoding, generating sequences element by element.

Generative Pre-trained Transformer (GPT) [105] GPT is a series of auto-regressive language models developed by OpenAI. The evolution of GPT includes multiple versions:

GPT 2 Model introduced in 2019 with 1.5 billion parameters, solid generation capabilities.

GPT 3 A giant model with 175 billion parameters, showcasing remarkable natural language understanding and generation.

GPT 3.5 : *ChatGPT* –a variant of GPT-3, is fine-tuned for conversational purposes, enabling users to engage in natural language interactions with the model. This, when publicly released in December 2022 has been the most significant breakthrough in AI so far.

GPT 4 A large multimodal model (accepting image and text inputs, emitting text outputs) that, while less capable than humans in many real-world scenarios, exhibits human-level performance on various professional and academic benchmarks [97].

The GPT series has significantly advanced the field of AI. The models boast an expansive range of applications, offering versatility in tasks from language translation to code generation. However, a notable limitation, shared with other auto-regressive models, is the potential for so-called *hallucinations*. These instances involve the model generating outputs that may lack accuracy or contextually appropriate information, presenting a key challenge for refining the reliability of such models.

Large Language Model Meta AI (LLaMA) [125] This model has been provided in multiple version of various model sizes, ranging from 7 billion to 65 billion parameters. Unlike other large language models that are typically only available via restricted APIs, Meta AI has chosen to make LLaMA's model weights accessible under a noncommercial license.

2.4.5 Encoder-Decoder Models

This section introduces a typical encoder-decoder Transformer architecture – T5. Additionally, newer instances within this category, such as SpeechT5 [5], Whisper [104], and SeamlessM4T [3], are only listed with references for further details.

Text-to-Text Transfer Transformer (T5) [106] This popular approach adopts a Transformer-based architecture, featuring both an encoder and a decoder, encompassing 12 Transformer blocks with a cumulative parameter count of 220 million. Pre-training occurred on a substantial C4 dataset (Colossal Clean Crawled Corpus), comprising 750 GB of English text. Similar to BERT, T5 employs the Masked Language Model (MLM) approach, where it learns to predict target words for enhanced language understanding.

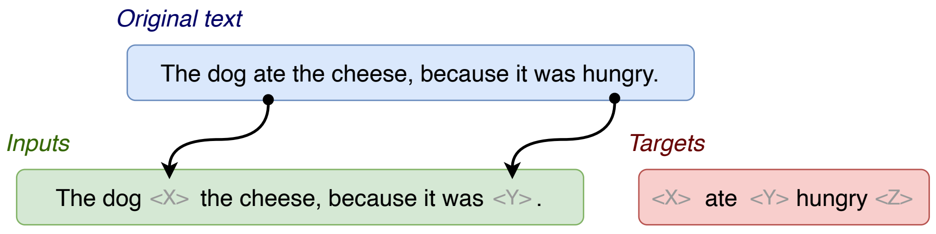

Original text
The dog ate the cheese, because it was hungry.
Inputs
The dog <X> the cheese, because it was <Y> .
Targets
<X> ate <Y> hungry <Z>

Fig. 2.33 T5: Example of a training sample.

The primary distinction between Bert and T5, apart the presence of the decoder, lies in the size of tokens (words) involved in prediction. While BERT predicts a target consisting of a single word (single token masking), T5, as illustrated in the figure above, can predict multiple words.

2.5 Special Architectures

The following methods do not belong to any of the three categories in terms of the architecture type, purpose or learning mechanism. Their backgrounds are related to this work though.

Generative Adversarial Network (GAN) [42]

As shown in Fig. 2.34, there are two neural networks contesting one with each other in terms of data distributions. The generator tries to fool the discriminator by creating fake samples of the same distribution as the real samples are. Its goal is to *maximise* the final classification error. The discriminator is trained to *minimise* the final classification error and its goal is to distinguish the real samples from the fake ones. This approach enables the model to learn in an unsupervised manner.

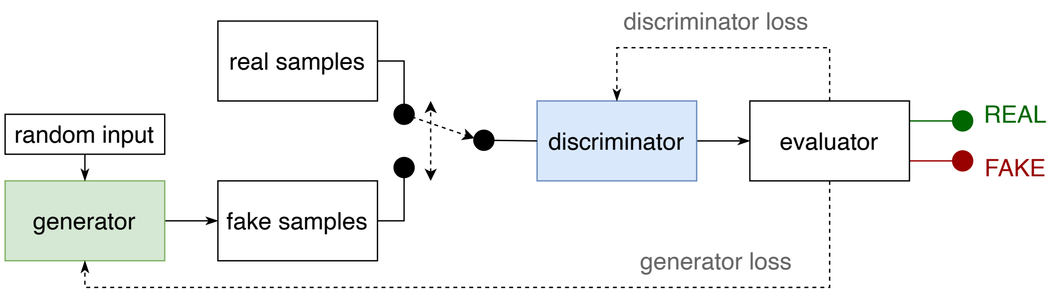


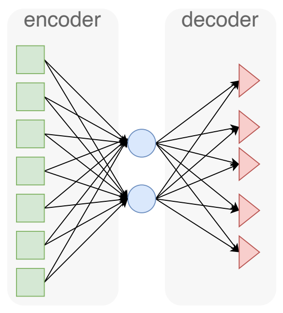
Fig. 2.34 The Generative Adversarial Network concept.

Autoencoder (AE) [16], [71]

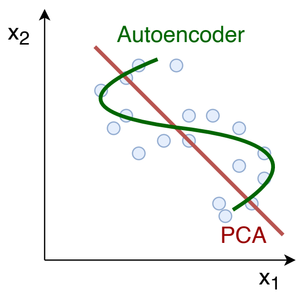
The encoder-decoder architecture is already known from Sec. 2.4.2, however, the idea is much older then Transformers are and the range of applications is wide. The first applications date back to 1980s and since then the idea has been popularized especially for its *dimensionality reduction* and *feature learning* capabilities. The structure typically consists of two parts:

- *an encoder* that maps the input into a coded representation;
- *a decoder* that reconstructs the coded representation.

The dimensionality of the (coded) hidden layer (also called a *bottleneck*; blue in Fig. 2.35a) is reduced compared to the original input layer. The goal is to make the hidden layer keep as much information as possible. Based on the nature of neural networks and apart from the standard *Principal Component Analysis (PCA)*, even nonlinear relations are handled (Fig. 2.35b).



(a) Autoencoder (a bottleneck).



(b) Autoencoder vs. PCA.

Fig. 2.35 Autoencoders (handling nonlinear relations).

Self-Organizing Map (SOM) [70]

Another unsupervised *dimensionality reduction* method is based on competitive learning (as opposed to loss-based methods) and its output is typically two-dimensional (called *a map* of size $\psi_1 \times \psi_2$ - Fig. 2.36). Partly motivated by the human cerebral cortex, the goal is to cause different parts of the output map to respond similarly to certain input patterns. The algorithm is well described in [11].

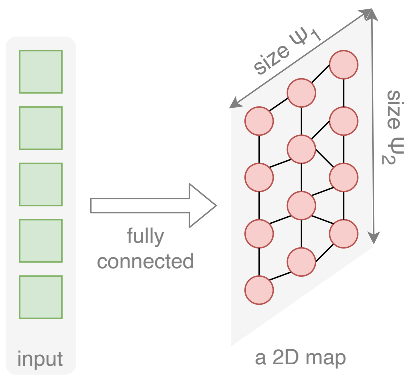


Fig. 2.36 Kohonen's Self-Organizing Map example.

Siamese network [19]

The original idea from 1993 has been popularized again with the rise of deep learning. There are two models (also called *twin* networks) sharing the same weights. They work in tandem on two different input vectors and their outputs are then compared using either the *triplet* or the *contrastive* loss. The approach is known for its application to the *face recognition* task. A detailed explanation is available in [11].

2.6 Learning Algorithm

As stated above, regarding the goals of this work, we assume a classification problem on supervised (labeled) data to demonstrate the presented methods. Therefore, the learning phase is mostly based on the well-known *backpropagation* algorithm [79] using the *Gradient Descent* iterative optimization. The following math complies with the notation listed at the beginning of this work supplied by the additions in App. A.

By default, the algorithm was derived for feedforward architectures (Sec. 2.2) and the overall procedure follows these steps:

1. *forward propagation* of a batch of samples;

$$A^{(1)} = f(W^{(1)} \cdot X + B^{(1)}) \tag{2.26}$$
$$A^{(i)} = f(W^{(i)} \cdot A^{(i-1)} + B^{(i)}) \tag{2.27}$$
$$A^{(q)} = Y = f(W^{(q)} \cdot A^{(q-1)} + B^{(q)}) \tag{2.28}$$

2. *error calculation* based on the chosen loss function $\mathscr{L}_{ff}$;

$$\mathscr{L}_{ff} = \frac{(U-Y)\times(U-Y)}{2} \tag{2.29}$$

3. *backpropagation* of the prediction error;

$$\Delta^{(q+1)} = (U-Y)\times f'[Z^{(q+1)}] \tag{2.30}$$
$$\Delta^{(i)} = \left[\left[W^{(i+1)}\right]^T \cdot \Delta^{(i+1)}\right] \times f'[Z^{(i)}] \tag{2.31}$$

4. *finding the optimal updates* - taken over from [20]; Every sample ξ has a vote $dW^{(i)}_{(\xi)}$ (resp. $dB^{(i)}_{(\xi)}$) on how the parameters $W^{(i)}$ (resp. $B^{(i)}$) should change to get the minimal error and then the result is obtained as a compromise of those votes. Index (i) indicates the layer. Consider $\Delta^{(i)}_{(\xi)}$ be the ξ^{th} column of the $\Delta^{(i)}$ matrix, which corresponds to the ξ^{th} sample. Analogically, $A^{(i-1)}_{(\xi)}$ is the ξ^{th} column of the activation matrix $A^{(i-1)}$ in the $(i-1)^{th}$ layer. Then we get the votes as:

$$dW^{(i)}_{(\xi)} = A^{(i-1)}_{(\xi)} \cdot \left[\Delta^{(i)}_{(\xi)}\right]^T \tag{2.32}$$
$$dB^{(i)}_{(\xi)} = \Delta^{(i)}_{(\xi)} \tag{2.33}$$

5. *parameters update*; The `batch_size` value states how many votes are processed together to make one update of the parameters (the learning is called sequential for `batch_size = 1`). For *batch learning* (`batch_size > 1`):

$$dW^{(i)} = \sum_{\xi}^{batch\_size} dW^{(i)}_{(\xi)} \tag{2.34}$$

The same is analogically applied to biases. The `learning_rate` value (μ), usually set $0 < \mu << 1$, is included in order to deal with GDA problems (shown below). The update of the parameters is then done as follows (`<t>` refers to a moment in time):

$$W^{(i)<t+1>} = W^{(i)<t>} + \mu \cdot dW^{(i)<t>} \tag{2.35}$$

$$B^{(i)<t+1>} = B^{(i)<t>} + \mu \cdot dB^{(i)<t>} \tag{2.36}$$

The procedure is commonly repeated over a specified number of epochs or until a required value of the error is reached. In case of recurrent architectures (Sec. 2.3), thanks to the method known as *backpropagation-through-time (BPTT)* from [36], the same procedure can be analogically applied. As shown in Fig. 2.37, the RNN layer can be unrolled over a limited number of time steps T and considered as subsequent feedforward layers.

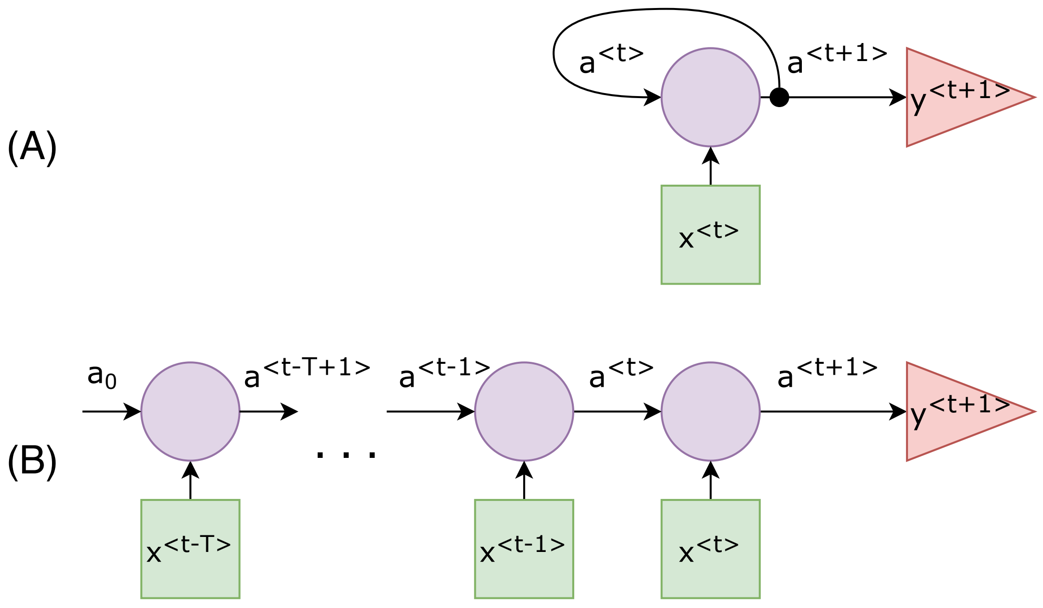


Fig. 2.37 BPTT unfolding an RNN through time.

Then the loss function and the parameters update are expressed as:

$$\mathscr{L}_{rnn} = \sum_{t=1}^{T} \mathscr{L}_{ff}^{<t>} \tag{2.37}$$

$$\frac{\partial \mathscr{L}_{rnn}}{\partial W} = \sum_{t=1}^{T} \left. \frac{\partial \mathscr{L}_{rnn}^{<t>}}{\partial W} \right|_{t} \tag{2.38}$$

2.6.1 Loss Functions

The learning process is significantly impacted by the thoughtful selection of evaluation metrics and loss functions. Broadly, loss functions can be categorized for tasks involving either *regression* or *classification*. For a more comprehensive list and detailed explanations, refer to [23].

Regression Loss Functions

- *Mean Squared Error (MSE):* Measures the average squared difference between predicted and actual values.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2.39}$$

- *Mean Absolute Error (MAE):* Computes the average absolute difference between predicted and actual values.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{2.40}$$

- *Huber Loss:* A combination of MSE for small errors and MAE for large errors, introducing a parameter δ.

$$\text{Huber Loss} = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2, & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{2.41}$$

Classification Loss Functions

- *Binary Cross-Entropy Loss:* Suitable for binary classification tasks, penalizing deviations from true class probabilities.

$$\text{Binary Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{2.42}$$

- *Categorical Cross-Entropy Loss:* Extends binary cross-entropy to multi-class classification scenarios.

$$\text{Categorical Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij}) \tag{2.43}$$

- *Sparse Categorical Cross-Entropy Loss:* Similar to categorical cross-entropy, but uses integer class labels instead of one-hot encoded vectors.

$$\text{Sparse Categorical Cross-Entropy Loss} = -\frac{1}{n}\sum_{i=1}^{n}\log(\hat{y}_i) \tag{2.44}$$

2.6.2 Limitations of Backpropagation

The *backpropagation* learning has not been overcome for more than 50 years, however, the procedure has three main shortcomings:

- *Stucking at a local minima*; Especially in case of deep structures, the number of parameters is enormous and finding the optimal solution (such parameters settings that makes the cost function minimal) is challenging. In most cases, the algorithm gets stuck in a local (not global) minima (addressed by *momentum* and ADA-based optimizers).
- *Exploding/vanishing gradient*; In case of many hidden layers, there are many derivatives multiplied together. If these derivatives are large, the gradient will increase exponentially until it eventually *explode*. Analogically, it eventually *vanishes* if many small derivatives are multiplied together.
- *Computational limits* due to the enormous number of operations in deep structures. In case of *Transformers*, current performance bottleneck resides primarily in hardware limitations. Unlike convolutions or LSTMs, the capabilities of these models are solely constrained by the size of the model that can fit into GPU memory and the volume of data that can be efficiently processed within a reasonable timeframe.

2.6.3 Design Choices and Optimization Tools

Network architectures described in the previous sections together with the learning algorithm are the baseline in the field of ANNs. Section 2.5 is then devoted to special methods that do not belong to any of the previous categories, but still are interesting in relation to this work. This section goes more into detail and focuses on related methods that support the baseline and help to overcome the learning limitations.

As stated above, the pure backpropagation algorithm suffers mainly from stucking in a local minima. Therefore, several methods adjusting the default learning equation have been proposed, in order to help the algorithm converge.

Optimizers. There are several optimization techniques for the *Gradient Descent Algorithm (GDA)*, to name some of them: *RMSProp*, *Adam*, *Nesterov*, *Adagrad*, *Adadelta*. A detailed explanation is provided in [112].

Learning rate. The learning rate (μ) is a common learning hyper-parameter. It helps to converge to the solution by little steps (Fig. 2.38). The value is usually being tuned during the training.

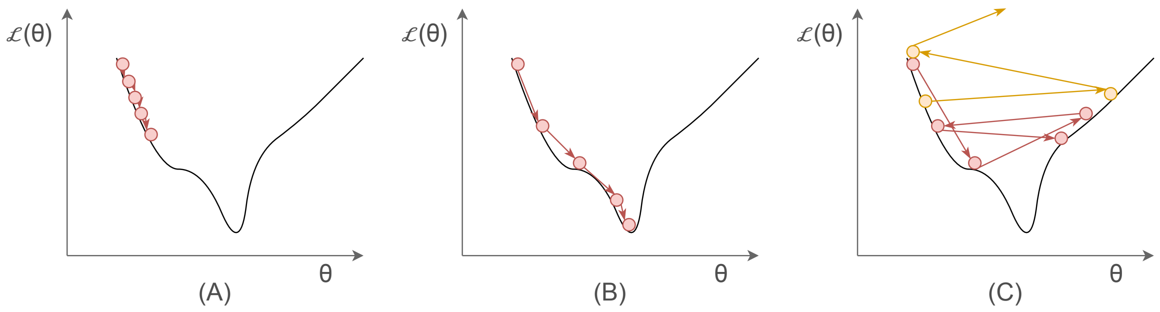


Fig. 2.38 Learning rate: (A) too low; (B) optimal; (C) too high (red) / way too high (orange).

Momentum. The momentum mechanism can be added to the step of parameters update (Eq. 2.35). The purpose is to prevent oscillations and to keep traveling in the same direction along the gradient. Assuming α to be the momentum rate, the change of Eq. 2.35 is shown in Eq. 2.45:

$$W^{(i)<t+1>} = W^{(i)<t>} + \mu \cdot [(1-\alpha)\cdot dW^{(i)<t>} + \alpha \cdot dW^{(i)<t-1>}] \tag{2.45}$$

Regularization techniques

- *L1 Regularization:* Adds the absolute values of weights as a penalty term in the loss function, promoting sparsity in the model.
- *L2 Regularization (Weight Decay):* Sometimes the weights become too specialized to the training data and cause so-called *over-fitting*. To prevent it, this method makes weights decay in proportion to their sizes. By adding the squared values of weights as a penalty term, it prevents the model from becoming too reliant on individual features. The default update formula (Eq. 2.35), λ being a decay factor, is adjusted as follows:

$$W^{(i)<t+1>} = W^{(i)<t>} + \mu \cdot (dW^{(i)<t>} - \lambda W^{(i)<t>}) \tag{2.46}$$

- *Dropout:* Apart from other regularization methods ($L1$ - Laplacian, $L2$ - Gaussian), the *dropout* mechanism [53] is another method addressing the overfitting problem.

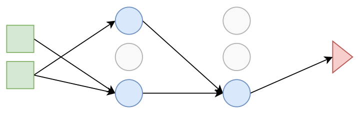


Fig. 2.39 Example of a dropped-out network.

As illustrated on the example in Fig. 2.39, selected nodes and the corresponding connections are ignored during individual iteratioins of the training phase. There is the *probability* hyper-parameter p deciding, for each node individually, about its omission. During the inference phase then, all nodes are considered as usual.

- *Drop-connect:* Extends dropout to connections rather than neurons, randomly dropping weights during training.
- *Gradient Clipping:* Limits the magnitude of gradients during backpropagation to prevent exploding gradients, often used in recurrent neural networks.

Initialization techniques Weights initialization can generally determine model's convergence and overall performance. The list below outlines some of the widely used initialization methods, each designed to address specific issues.

- *Random Initialization (Gaussian):* Initializes weights using random values drawn from a Gaussian distribution with mean 0 and small variance.

- *Xavier/Glorot Initialization:* Designed to address vanishing/exploding gradient issues, setting weights from a Gaussian distribution with mean 0 and variance $\frac{1}{\text{input size}}$.
- *He Initialization:* Similar to Xavier, but uses $\frac{2}{\text{input size}}$ for the variance, recommended for ReLU activation functions.

Some of the other methods include: *Orthogonal*, *Sparse*, *Identity* or *Zero* initialization.

Normalization techniques These methods maintain consistent input scales, preventing issues such as vanishing or exploding gradients. This helps more efficient optimization and supports the training especially of deeper models with many layers.

- *Batch Normalization:* Normalizes the input of a layer by adjusting and scaling activations within mini-batches, mitigating internal covariate shift.
- *Layer Normalization:* Similar to batch normalization, but normalizes across the entire layer's input rather than mini-batches, making it applicable in recurrent neural networks.
- *Group Normalization:* Divides channels into groups and normalizes each group independently, striking a balance between batch and layer normalization.
- *Instance Normalization:* Normalizes each channel independently for each instance in the batch, frequently used in style transfer and image-to-image translation.
- *Switchable Normalization:* Combines batch normalization, layer normalization, and instance normalization, allowing the model to learn the most suitable normalization method.

Weight constraints techniques Weight constraints are another technique for regulating the model's learning behaviour and preventing overfitting. By imposing constraints on weight vectors, these methods helps the stability during training, facilitate convergence, and contribute to making the model generalize better.

- *MaxNorm Constraint:* Constrains the maximum norm of weight vectors, limiting their magnitude during training.
- *UnitNorm Constraint:* Constrains the L2 norm of weight vectors to be 1, preserving the direction of the vectors.
- *Orthogonal Constraint:* Constrains weight matrices to be orthogonal, preserving angles between vectors and aiding in preventing overfitting.

2.7 Neural Architecture Search

Previous sections provide a comprehensive summary of various ANN architectures, together with methods mostly addressing the shortcomings of the general learning algorithm (Sec. 2.6). In most of these cases, the network structure is fixed. This section, with respect to the objectives of this work (see Sec. 1.1), is devoted to algorithms for searching the network architecture in terms of neurons, synapses and their mutual interconnections.

In [37] Neural Architecture Search (NAS) methods are classified into three dimensions: search space, search strategy, and performance estimation strategy. This approach has resulted in the development of various search algorithms catering to different aspects of NAS. Hyper-parameter tuning initially adopted Bayesian Optimization [12], [32], [38], [67], while Reinforcement Learning ([136], [102], [63]) was utilized to train agents interacting with the search space. Evolutionary algorithms ([81], [108]) encode model architectures into DNA and evolved candidate pools. Notably, ProgressiveNAS [80] employed heuristic search, gradually constructing models from simple to complex architectures. Conversely, LayerNAS [39] focused on making changes to the layers of a full complex model. A comprehensive survey of *Automated Machine Learning (AutoML)* methods is provided in [46]. In general, these algorithms are divided into three categories: 1) *hyper-parameter tuning*; 2) *top-down methods* (pruning and shrinking); and 3) *bottom-up methods* (building a network from scratch).

2.7.1 Hyper-parameter Tuning

This methodology utilizes diverse optimization tools to navigate the state space search, aiming to find optimal network hyper-parameters such as the number of layers, neurons per layer, learning rate, batch size, and other relevant factors.

Evolving neural networks (NEAT) [122]

This approach uses *evolutionary* optimization to construct deep learning architectures that are, based on the published results, more complex than the hand-made ones. It is based on searching the enormous space of hyper-parameters, components and network topologies. The researchers claim that the full potential of their approach is constrained by computational resources and the results are based more on fast-learners instead of top-performers.

The generated network architecture is initialized by a graph of chromosomes. In case of the original (NEAT) approach, each node represents a neuron. Later then, the approach was applied to deep networks, where each node represents a layer. The latest version called

Coevolution DeepNEAT [86] implements two parallel graphs of chromosomes that are combined during the fitness evaluation. Related observations:

- An arbitrary connectivity is allowed (layers not stricly fully-connected).
- Depending on the network size, elementary units are neurons or layers.
- The fitness (evaluation function) is based on how well the evolved networks can be trained (using the GDA) to perform in the given task.

Neural architecture search with reinforcement learning [136], [137]

This approach is a representative example of the architecture search algorithm. In this case, reinforcement learning is used to train an RNN, which composes the target network architecture (see Fig. 2.40) for a given task automatically. The (RNN) controller is capable of designing a CNN architecture that rivals the SotA methods on the CIFAR-10 dataset and an RNN architecture dealing with a language modeling task.

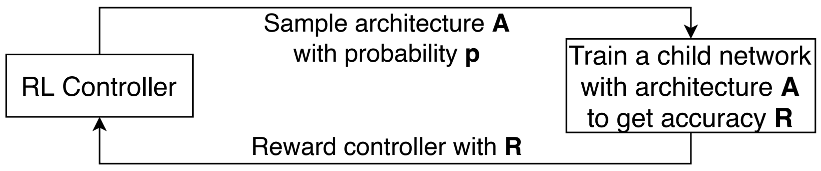

Sample architecture **A** with probability **p**
RL Controller
Train a child network with architecture **A** to get accuracy **R**
Reward controller with **R**

Fig. 2.40 The Neural Architecture Search with RL principle [137].

An adjusted version of the algorithm determined to generate models for mobile devices is called *MnasNet* [123]. This version is mainly focused on the trade-off between accuracy and inference latency. The generated network is being described by hyper-parameters, such as the filter size, stride and the number of filters (in case of CNN). Also, the controller is trained to modify the architecture of the network, for example using the *skip-connections* approach (see Sec. 2.2.4). Accuracy of the generated network is used as a reward for the RL algorithm.

LayerNAS: NAS in polynomial complexity [39]

This novel approach streamlines the multi-objective NAS problem using combinatorial optimization, significantly reducing complexity. This leads to a substantial decrease in the number of model candidates, requiring less computation for multi-trial searches and enabling the discovery of superior-performing architectures. A potential limitation of this approach lies in its reliance on the layer-wise pattern of the final network when searching for arbitrary network architectures.

MetaQNN: designing neural network architectures using RL [9]

MetaQNN, a reinforcement learning algorithm, autonomously generates high-performing Convolutional Neural Network (CNN) architectures for specific learning tasks. The agent-designed networks, utilizing standard layers, surpass existing networks and rival state-of-the-art methods with more complex layer types in image classification benchmarks. This approach strictly adheres to the layer-wise pattern and also is designed for CNNs.

NSGA-NET: NAS using multi-objective genetic algorithm [83]

NSGA-Net is introduced as an evolutionary Neural Architecture Search (NAS) approach, achieving competitive results by optimizing dual objectives of minimizing error and computational complexity. The algorithm efficiently explores potential neural network architectures through population-based search, incorporating a Bayesian Network for exploitation. NSGA-Net attains CIFAR-10 error rates similar to state-of-the-art NAS methods while utilizing significantly fewer computational resources, showcasing its promise in the realm of deep learning.

Progressive neural architecture search [80]

This method utilizes sequential model-based optimization (SMBO) to systematically explore structures in ascending complexity, while learning the structure of CNNs. Compared within the same search space, this approach is up to 5 times more efficient than the RL method from [136] in terms of evaluated models and 8 times faster in total compute.

2.7.2 Pruning Methods

These algorithms remove synapses from fully-connected networks, however, in contrast to the *dropout* optimization technique (Sec. 2.6.3), the dropped-out synapses are not turned back on for the inference phase and instead, the resulting pruned network is used for prediction. The general pruning procedure consists of these steps (corresponding to Fig. 2.41):

1. design an oversized network structure for given classification data;
2. train the network until the maximal possible accuracy is reached;
3. remove selected synapses (depending on chosen pruning measure);
4. repeat step (3) as long as the original maximal accuracy is kept.

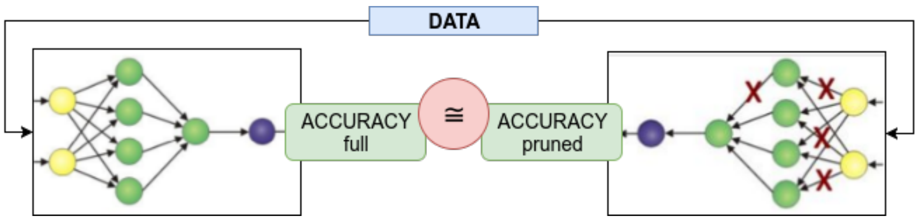


Fig. 2.41 The principle of network pruning.

A detailed study on this topic is provided in [20], where a new pruning measure is introduced. It was shown that generally more than 90% of the synapses are commonly redundant in fully-connected networks. Moreover, several experiments proved the ability of the presented algorithm to select features and to find the minimal network structure for given data. As a result, pruned networks are faster in the prediction phase and, as all remaining synapses are guaranteed to be important, the information flow can be tracked and thus parts of such a network can be demystified. In[20]), the derived algorithm is compared to these related studies:

- *Skeletonization* [92];
- *Optimal brain damage* [76];
- *Sensitivity measure* [68].

The main drawback of the *top-down* approach is the need of (in practise random) choice of the initial network. As long as the algorithm can only remove parts (and not add new ones), the result is restricted by the initial structure. Moreover, even though the resulting network is more efficient and the accuracy is kept, the accuracy never improves compared to the original.

2.7.3 Building Methods

These algorithms take elementary units (or blocks of units in some cases) and connect them into a structure for a specific purpose. This methodology is closely related to this work.

Badger Architecture [110]

The long-term goal of the Prague-based GoodAI company is to build general artificial intelligence and the *Badger* architecture - their latest project, among other related studies, is probably the closest one to this work.

As stated in the paper, they introduce a way how to adapt to new environments by "learning to learn learning algorithms". The learning procedure is illustrated in Fig. 2.42.

There is an *agent* made up of many so-called *experts* sharing a universal *expert policy*. The overall goal is to make the experts quickly adaptable, when a new environment is shown to the system.

First of all, the expert policy is trained over generations of agents on diverse environments (outer loop) and then it is fixed. Then an agent is run in a new environment and its adaptation emerges as a result of inter-expert communication (inner loop). If needed, more experts are added by cloning the old ones. At inference time, the roles of experts are assigned dynamically.

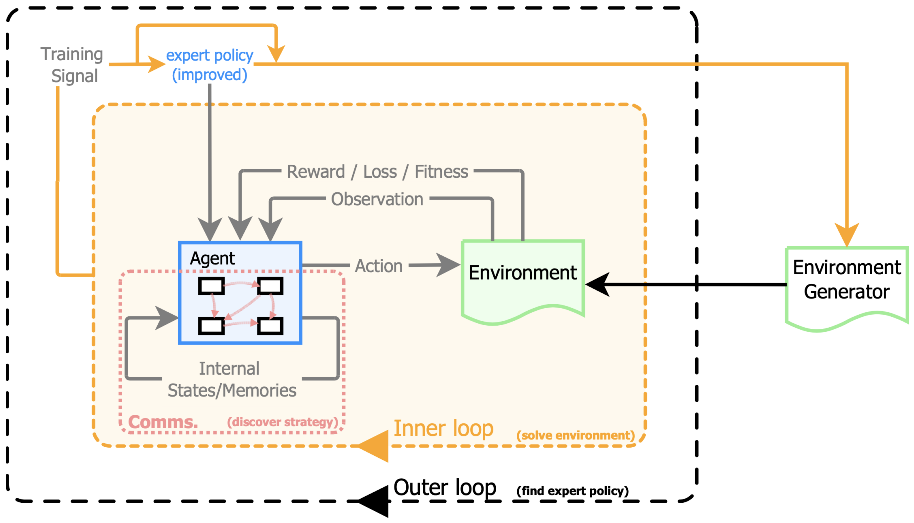


Fig. 2.42 The inner and outer learning loops in the Badger architecture [110].

As the current state of the project, there is an evidence that: 1) the fixed shared policy can lead to adaptation during the inner loop; 2) adding experts can help find better solutions (and faster); A few related observations:

- *the goal is the adaptation* - the experts are not taught to deal with a specific problem, but rather they are taught to adapt to any general environment with a variable number of inputs;
- *a stochastic universal policy* - by default, one fixed policy is shared by all experts and the policy is represented by a trained neural network;
- *toy-tasks tested* - by now, the approach needs more effort to be scaled up to a real world setting.

2.8 Frameworks for Neural Networks

There are several popular and widely used neural network frameworks available. Among the most popular ones are:

- *PyTorch* [99] Developed by Facebook's AI Research lab (FAIR), PyTorch is known for its dynamic computational graph and is widely used in academia and industry for research and development of neural network models.
- *TensorFlow* [1] Developed by Google Brain, TensorFlow is a widely used open-source deep learning framework offering a comprehensive ecosystem for building and deploying machine learning models.
- *Keras* [23] Integrated with TensorFlow, Keras provides a user-friendly interface for building neural networks and is preferred by beginners and researchers for its simplicity and ease of use.
- *scikit-learn* [101] Scikit-learn is a popular machine learning library in Python, offering simple and efficient tools for data mining and data analysis, built on NumPy, SciPy, and matplotlib.
- *JAX* [17] Developed by Google Research, JAX is a composable and extensible library for machine learning and numerical computing, gaining popularity for its functional programming style and compatibility with NumPy.

Chapter 3

Related Principles and Technologies

Building upon the comprehensive exploration of neural networks in the previous chapter, we now delve into a detailed examination of additional machine learning principles and technologies. These foundational elements play a pivotal role in the approach developed within this work. Our focus extends to Reinforcement Learning (RL), Multi-Agent Systems (MAS), Genetic Algorithms (GA), and the Human-in-the-Loop approach.

3.1 Reinforcement Learning

Reinforcement learning (RL) stands as one of the three fundamental paradigms in machine learning (see Fig. 3.1). *Unsupervised learning* identifies hidden data patterns from features, particularly excelling in self-supervised learning and producing high-performance pre-trained models. *Supervised learning* relies on a supervisor to furnish a labeled dataset, a process that could be both costly and impractical. Unlike both, reinforcement learning operates without a supervisor or a pre-collected dataset. The system learns from making decisions by interacting with its environment over (usually many) episodes of time.

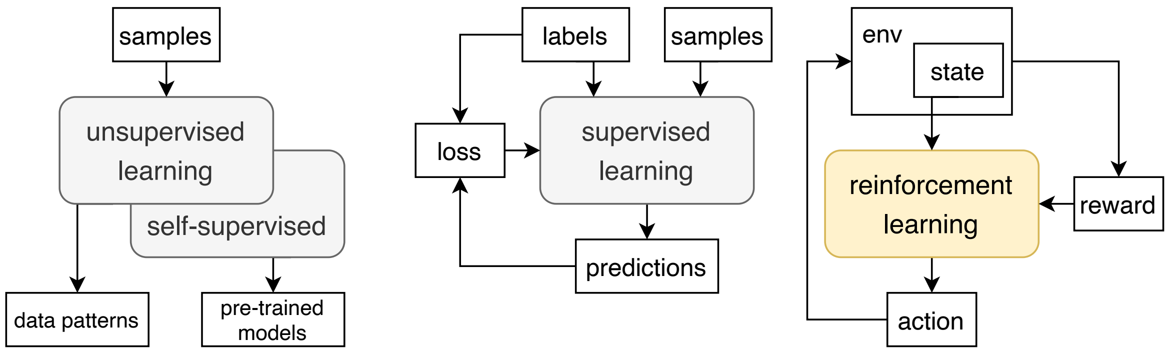


Fig. 3.1 Placement of reinforcement learning within the machine learning family.

The main principle is that we don't explicitly instruct the system on what it should do (provide correct labels), but instead, we evaluate its behaviour with positive or negative feedbacks. The approach is rooted in Pavlov's theory of classical conditioning, where learning occurs through association. The familiar experiment demonstrates a widely acknowledged fact - through the use of positive or negative stimuli, a dog eventually learns to respond appropriately to specific situations.

The inherent nature of RL makes it especially suitable for tasks where collecting labeled data is challenging or tasks aiming to develop effective strategies for achieving positive future outcomes through decision-making over time, e.g. making trading decisions on a stock market. Reinforcement learning is also suitable for tasks aiming to model optimal behaviour in specific environments, such as operating drones, autonomous vehicles, or manufacturing devices. Next, it has demonstrated success in playing games such as Go, Chess, and various video games [135]. Finally, and importantly for this work, it fits very well for learning in multi-agent systems.

3.1.1 Formulation of a RL problem

Many different approaches to RL exist, sharing the same foundational principles while differing in their optimization algorithms. Despite their differences, they all rely on these fundamental terms that have to be defined to formulate a RL problem:

- *Environment.* This is the world of operation that illustrates the problem to be solved. We can encounter either real-world or simulated environments. The environment informs the *agent* about its current *state* and *rewards* it for its taken actions.
- *Agent.* A single entity interacting with the *environment* by making decisions about its next action based on the current *state*. As a feedback, it receives a *reward* for its choice and information about the new *state*.

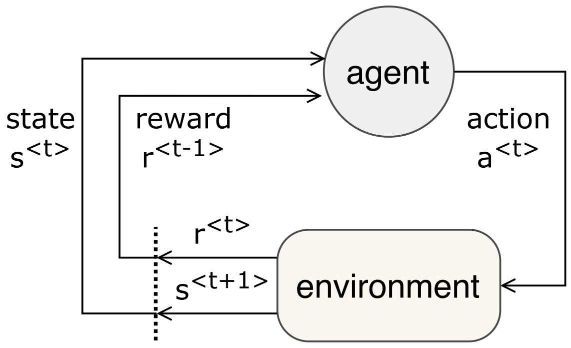


Fig. 3.2 A single step of the reinforcement learning procedure (single agent).

- *State.* The state $s^{<t>}$ characterizes the situation of the *environment* or, depending on the problem settings, the *agent* within it at time t. The state must be self-contained, meaning it should encompass all the information the *agent* requires to decide on its *action*. States can be discrete or continuous.

- *Action.* Each *agent* has a set of (usually discrete) actions representing the moves we want the *agent* to learn. Action $a^{<t>}$ is the selected agent's move at time t based on *state* $s^{<t>}$.

- *Reward.* A positive or negative feedback from the *environment* at time t based on *agent's* move $a^{<t-1>}$ at *state* $s^{<t-1>}$. It reflects the behaviour the *agent* is supposed to learn.

The agent interacts with the environment over a sequence of time-steps $t \in [0, T]$ if the environment has a terminal state $s^{<T>}$, or $t \in [0, \infty)$ otherwise. Tasks with no terminal state are referred to as *continuous*, and these can theoretically last indefinitely, as seen in the example of a robot managing warehouse operations. On the other hand, tasks with pre-defined terminal state(s) are called *episodic*; for instance, a game of Chess concludes when the game is over. Each time-step of an episode can be defined by a state, taken action, and obtained reward. Then we can describe each episode by concatenating these triples into the so-called *trajectory*: $(s^{<0>}, a^{<0>}, r^{<0>}, s^{<1>}, a^{<1>}, r^{<1>}, s^{<2>}, \ldots)$. Trajectories (episodes) are completely independent of each other, and their distinctiveness and variety are closely tied to the effectiveness of the RL algorithm, as we will explore further later on.

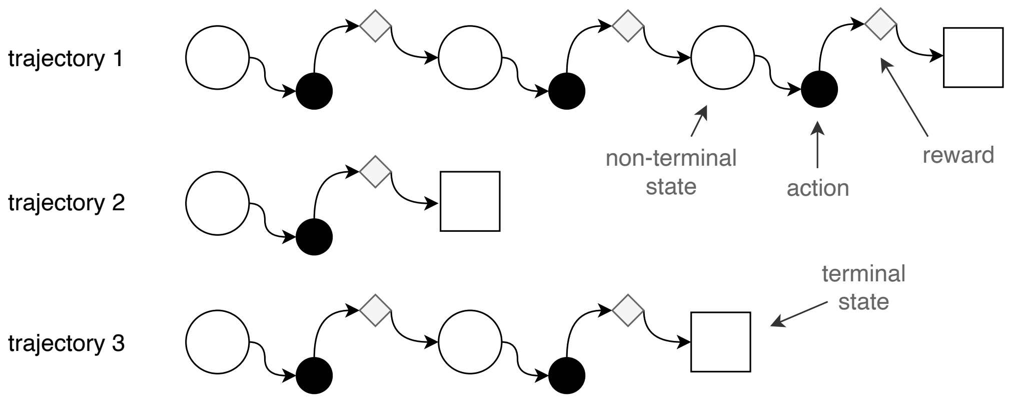


Fig. 3.3 Example of three independent trajectories of episodic RL tasks.

Markov property [84]

Each potential transition from state s_j to state s_i can be characterized by a *transition probability* $p_{ij}^{<t>} = \mathbb{P}[s^{<t+1>} = s_i | s^{<t>} = s_j]$, which may also vary over time. A system has the Markov property if its transition probabilities follow the rule expressed by Eq. 3.1, which can be paraphrased as:

"Future is independent of the past given the present."

$$\mathbb{P}[s^{<t+1>}|s^{<t>}] = \mathbb{P}[s^{<t+1>}|s^{<0>}, s^{<1>}, ..., s^{<t>}] \tag{3.1}$$

This implies that the current state already captures information about past states and is independent of them. This property is crucial for most reinforcement learning algorithms. A sequence of states with the Markov property is termed a Markov process (or Markov chain). The dynamics of such a system (illustrated in the example on Fig. 3.4) are entirely defined by its states (*sleep*, *run*) and corresponding transition probabilities.

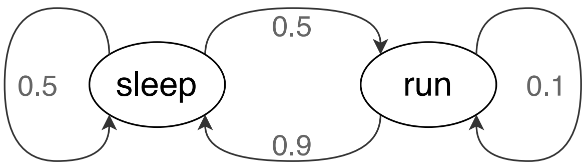


Fig. 3.4 Example of a Markov chain (process) of two states.

Next, we can enhance the chain by incorporating a set of actions available from each state and an immediate reward received for each transition. This leads to a *Markov Decision Process (MDP)*, formally defined as a 4-tuple (S, A, P, ρ), where:

- S is a set of states (a state space);
- A is a set of possible actions, α_s a set of actions available from state s;
- P is a transition probability matrix, e.g. $p^{<t>}_{(i,j,a)} = \mathbb{P}[s^{<t+1>} = s_i | s^{<t>} = s_j, a^{<t>} = a]$ is the transition probability for moving from state s_j to s_i by taking action a.
- ρ is a matrix of immediate rewards for transitioning among states, e.g. $r^{<t>}_{(i,a)} = \mathbb{E}[r^{<t+1>} | s^{<t>} = s_i, a^{<t>} = a]$ is the immediate reward for moving from state s_j to s_i by taking action a.

Reinforcement learning problems are typically structured as Markov Decision Processes. If the agent cannot directly observe the underlying state, it must maintain a probability distribution over the set of possible states, and it is then called a *Partially Observable MDP*. The Markov property is particularly useful for environments with longer episodes, as storing complete past information and using it for decision-making becomes infeasible.

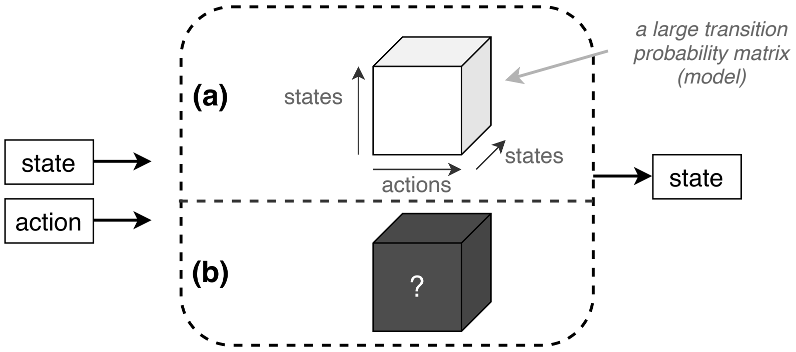


Fig. 3.5 Environment model: (a) observable (rare), (b) hidden (typical case).

As the agent is responsible for generating actions, environments control the transition of states. In environments strictly governed by the laws of physics, one could theoretically define an environment model and a corresponding (very large) transition probability matrix (see Fig. 3.5a). In this case, we can apply so-called *model-based* RL algorithm, which is more discussed later. However, in most cases, environments exhibit complex internal dynamics, filled with unseen factors, making them appear like black boxes to us (see Fig. 3.5a). In RL, we employ an *agent*, as shown in Fig 3.2, to iteratively interact with the environment, studying its behaviour. This process is grounded in three fundamental concepts: *return*, *policy*, and *value*.

Return. Instead of prioritizing only the next highest immediate reward on its trajectory, the agent is focused on accumulating rewards to achieve the maximum sum at the end of each episode. *Return* $R^{<t>}$ is the total reward calculated as a cumulative (and discounted) sum of all rewards obtained from the current state $s^{<t>}$ until the end of the episode, formally defined as given in Eq. 3.2.

$$R^{<t>} = r^{<t>} + \gamma r^{<t+1>} + \gamma^2 r^{<t+2>} + ... + \gamma^{<T-t>} r^{<T>} = \sum_{\tau=t}^{<T>} \gamma^{<\tau-t>} r^{<\tau>} \tag{3.2}$$

Here, $\gamma \in [0,1]$ is a discount factor that prevents the *Return* from growing infinitely for a large number of time-steps. Additionally, it encourages the agent to prioritize immediate rewards, which are more valuable (see Fig. 3.6a). Simultaneously, the formula compels the agent to favor higher total returns over appealing immediate ones (see Fig. 3.6b).

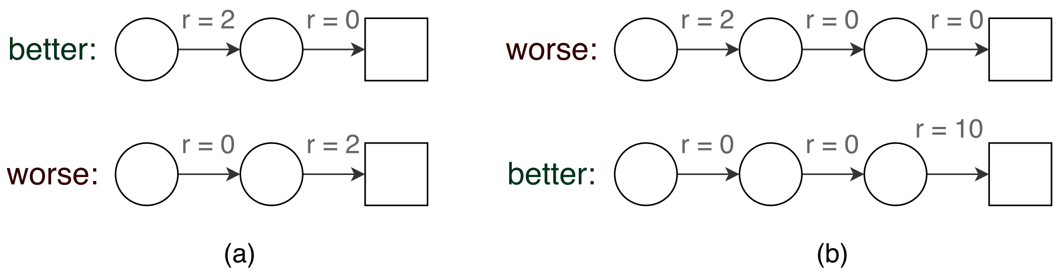


Fig. 3.6 Influence of the discount factor $\gamma \in [0,1]$ in Eq. 3.2.

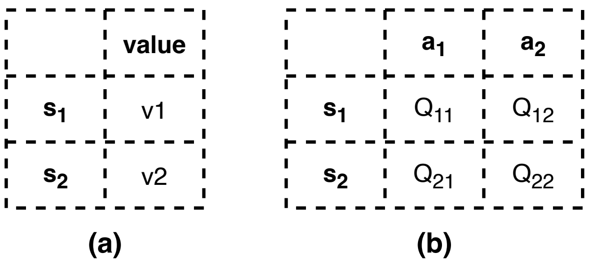
Policy π Policy represents the strategy followed by the agent when deciding which action to take. For example, the strategy could be to always choose a random action or to avoid obtaining low immediate rewards. Theoretically, a policy can be envisioned as a vast look-up table providing best available actions based on the current state. In practice, due to the enormous number of state-action pairs, policies are represented as functions rather than tables. Policies can be *deterministic*, always selecting the same action for a given state, or *stochastic*, more common and also practical in situations where unpredictability is desired, such as in gaming.

A reinforcement learning algorithm makes agents learn their policies. The primary objective of RL methods is to find the optimal *policy* π^* (the best one). In the quest for the best solution, a comparison metric is needed, and in comparing policies, it is the *value*.

Value $v^{(\pi)}(s)$ The value of state s under policy π is computed as the expected *Return* when following the policy π (Eq. 3.3):

$$v^{(\pi)}(s) = \mathbb{E}[R^t | s^t = s] \tag{3.3}$$

In terms of earlier definitions, we can view *reward* as immediate pleasure, while *value* would represent long-lasting happiness [34]. The value can also be interpreted as experience and stored in memory, for example, in the form of a table (see Fig. 3.7a). Alongside *value*, we analogically define the so-called Q-value$^{\pi}(s,a)$, representing the expected *Return* by taking action a in state s and then following policy π (see Fig. 3.7b).



(a)

	value
s_1	v1
s_2	v2

(b)

	a_1	a_2
s_1	Q_{11}	Q_{12}
s_2	Q_{21}	Q_{22}

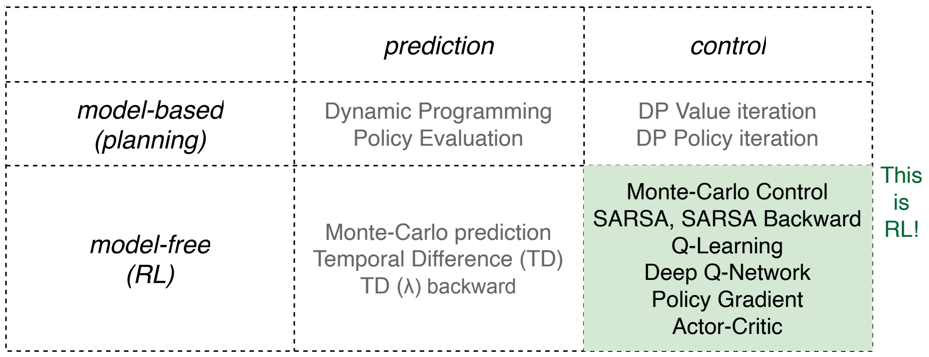
Fig. 3.7 Difference between *value* (a) and *Q-value* (b).

Once we have a metric to evaluate each policy for every state, we can compare policies and assert that the one with a higher *value* is better. The optimal policy is then the one yielding the highest *Return*, and finding the optimal policy solves a RL problem. To summarize, we have formulated the RL problem as follows:

1. Structure the problem as a Markov Decision Process.
2. Define environment, agent, state, action, reward.
3. Apply RL algorithm.
4. Get a trained agent with optimal policy.

3.1.2 Introduction to Optimization Techniques

As previously indicated in Fig. 3.5, environments of certain optimization problems can be explicitly modeled. In such instances, solutions can be derived analytically without the requirement for the agent to interact with the environment. In these cases, rewards and transition probabilities are deterministic and known in advance. These methodologies are classified as *model-based* algorithms.



	prediction	*control*
model-based (planning)	Dynamic Programming Policy Evaluation	DP Value iteration DP Policy iteration
model-free (RL)	Monte-Carlo prediction Temporal Difference (TD) TD (λ) backward	🟩 Monte-Carlo Control SARSA, SARSA Backward Q-Learning Deep Q-Network Policy Gradient Actor-Critic

Fig. 3.8 Categories of optimization algorithms. Green: RL algorithms [34].

However, a significant portion of real-world problems exhibits complexities that hinder the construction of an accurate environment model. Consequently, these problems are addressed using *model-free* techniques. Additionally, as illustrated in Fig 3.8, we can categorize the algorithms into *prediction* and *control* techniques. In *prediction*, the policy serves as the input, and the objective is to output a value function. On the other hand, *control* algorithms do not have input, and their task is to explore the policy space to identify the optimal policy. This category is traditionally referred to when discussing RL algorithms,and as such, they are the primary focus in the subsequent text.

As the environment is generally too complex to be modeled, and its internal operations remain invisible to us, *model-free* algorithms employ an agent to observe the environment's behavior through interaction. This process mirrors human learning and usually starts in a trial-and-error manner. In the context of an episodic task, the agent undergoes numerous episodes with diverse trajectories (see Fig. 3.3), essentially constituting its training data or, more precisely, acquiring experience.

Bellman equation. Having the *value* computed as the expected *Return R* (Eq. 3.2) of a state by following specific policy, the selection of the optimal policy is possible. However, obtaining the *Return* involves tracing all possible trajectories down to the terminal state, which is costly and practically infeasible for large state spaces.

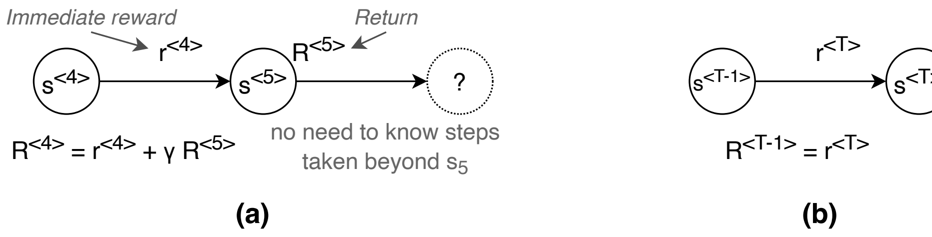


Fig. 3.9 Illustration of the power of Bellman equation (Eq. 3.4).

The Bellman equation (Eq. 3.4) can be expressed in various forms, but what is crucial for RL is the decomposition of the *Return* $R^{<t>}$ of the current state into a sum of two components:

- immediate reward $r^{<t>}$ from an action to reach next state
- discounted value of the next state *Return* $R^{<t+1>}$ by following the same policy onwards, where $\gamma \in [0,1]$ is the discount factor

$$R^{<t>} = r^{<t>} + \gamma R^{<t+1>} \tag{3.4}$$

The same relation can analogically be applied for *value* (Eq. 3.5) and Q-value (Q in Eq. 3.6), forming the foundation for most RL algorithms.

$$v^{(\pi)}(s^{<t>}) = \mathbb{E}[r^{<t>} + \gamma v^{(\pi)}(s^{<t+1>})] \tag{3.5}$$

$$Q^{(\pi)}(s^{<t>}, a^{<t>}) = \mathbb{E}[r^{<t>} + \gamma Q^{(\pi)}(s^{<t+1>}, a^{<t+1>})] \tag{3.6}$$

Crucial observations about the Bellman Equation are:

- The *Return* can be computed recursively, without reaching the end of the episode, by leveraging the *Return* from the next step. When reaching a terminal state, the *Return* equals the immediate *reward* (Fig. 3.9b).
- We can work with estimates, rather than exact values. As it is expensive to measure the actual *Return*, we work with *value* or *Q-value*.
- There are two ways to compute the same thing:

  1. *Return* from the current state
  2. *Reward* from the selected action $+$ *Return* from the next state.

  As we can use estimates here, we can observe the difference between these two estimates and compute the estimation error. By reducing this error, the estimates can be improved.

Temporal Difference Error. Returning to Eq. 3.6, if the estimations were perfectly accurate, the left and right sides of the equation would be equal. However, since this is typically not the case, we can assess the accuracy of the estimations by comparing them. Then, we can calculate the estimation error TDe (Eq. 3.7).

$$TDe = [r^{<t>} + \gamma Q(s^{<t+1>}, a^{<t+1>})] - Q(s^{<t>}, a^{<t>}) \tag{3.7}$$

where TDe is known as the Temporal Difference (TD) error, and the algorithm is formulated to iteratively update the Q-value estimates in a way to minimize the TDe error. Eq. 3.9 outlines the update rule for the fundamental RL algorithm known as *TD(0)*. Here, α serves as a constant step-size parameter, determining the pace of the learning process.

$$Q(s^{<t>}, a^{<t>}) = Q(s^{<t>}, a^{<t>}) + \alpha\, TDe \tag{3.8}$$

$$Q(s^{<t>}, a^{<t>}) = Q(s^{<t>}, a^{<t>}) + \alpha\left[(r^{<t>} + \gamma Q(s^{<t+1>}, a^{<t+1>})) - Q(s^{<t>}, a^{<t>})\right] \quad (3.9)$$

There are multiple parameters of how to improve the estimation process and by combining them we can distinguish among several model-free algorithms as shown in Tab. 3.1. The most common differences are in:

- *frequency* - number of forward steps before value update;
- *depth* - number of backward steps to propagate the update;
- *formula* used to compute the update - multiple variants of the Bellman equation.

algorithm	*output*	*frequency*	*depth*	*formula*
Monte-Carlo prediction	value	episode	episode	Return Error
TD (0)	value	one	one	TD Error
TD (λ)	value	one	N	TD Error (weighted Returns)
Monte-Carlo control	Q-value	episode	episode	Return Error
SARSA (TD control)	Q-value	one	one	TD Error
SARSA (λ)	Q-value	one	N	TD Error (weighted Returns)
Q-Learning	Q-value	one	one	TD-max Error
Deep Q-Network	Q-value	one	one	TD-max Error
Policy Gradient	Policy	episode	episode	Return-based Loss
Actor-Critic	Policy	N	one	Advantage-based Loss

Table 3.1 Parametrization of model-free RL algorithms.

Policy-based vs. Value-based model-free algorithms. We continue to concentrate on model-free algorithms, specifically designed for problems where algebraic solutions are not feasible - the ones that are commonly considered when discussing RL. Among the shared principles across different approaches, they all begin with arbitrary estimates of the searched quantity and incrementally improve these estimates through interactions with the environment, more or less based on Eq. 3.9. The overarching goal for all of them is to find the optimal policy. They can be categorized into two groups: those that find the policy directly (referred to as *Policy-based*) and those that compute the policy indirectly, referred to as *Value-based*. As the optimal policy is closely linked with the optimal value, finding one allows us to derive the other.

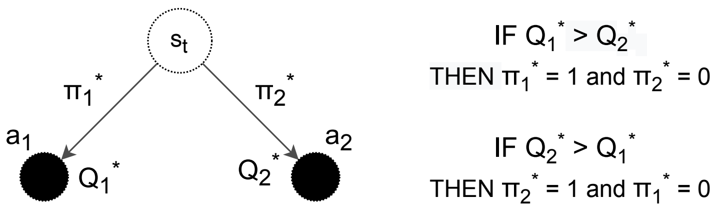
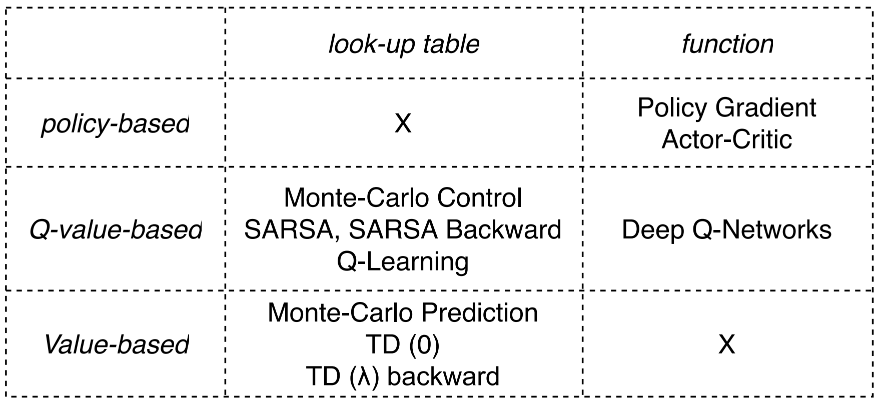


Fig. 3.10 Deriving optimal policy from optimal value with Q_1 and Q_2 being the Q-values of corresponding actions (a_1 and a_2).

As illustrated in Fig. 3.10, the optimal policy can be derived from the optimal value by selecting the action with the highest value. It's worth noting that this renders the resulting policy deterministic and predictable, except in cases of ties. Another key feature distinguishing RL algorithms is whether they utilize a look-up table or a function, leading to the classification of model-free RL algorithms depicted in Fig. 3.11.



	look-up table	*function*
policy-based	X	Policy Gradient Actor-Critic
Q-value-based	Monte-Carlo Control SARSA, SARSA Backward Q-Learning	Deep Q-Networks
Value-based	Monte-Carlo Prediction TD (0) TD (λ) backward	X

Fig. 3.11 Deriving optimal policy from optimal value [34].

Exploration vs. Exploitation. The exploration-exploitation dilemma is a fundamental challenge in reinforcement learning and decision-making problems. It refers to the trade-off between exploring new actions to discover potentially better outcomes and exploiting known actions to maximize immediate rewards. Striking the right balance is crucial for achieving optimal long-term performance. Here are some of the best techniques to address the exploration-exploitation dilemma:

- *Epsilon-Greedy Strategy:* A simple yet effective approach where the agent explores randomly with probability ε and exploits the best-known action with probability $(1-\varepsilon)$. The evolving value of ε can be expressed as stated in Eq. 3.10, ensuring that the agent becomes more inclined to exploit over time (see Fig. 3.12).

$$\varepsilon(\tau) = \varepsilon_F + (\varepsilon_0 - \varepsilon_F) * e^{-\tau/\varepsilon_d} \tag{3.10}$$

Here, τ is the current episode number, $\varepsilon_0 = 0.9$ (starting value at episode 0), $\varepsilon_F = 0.05$ (final value at the last episode F) and value decay $\varepsilon_d = 1000$, being an example of many possible settings [103].

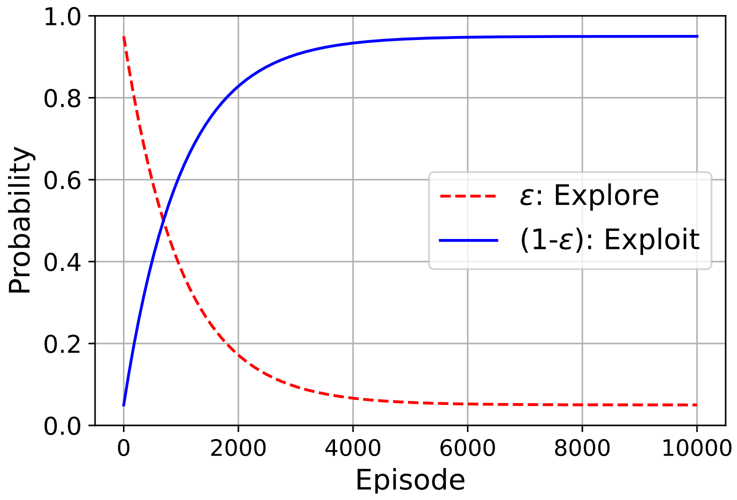


Fig. 3.12 The ε-greedy strategy using a decay based on Eq. 3.10.

- *Upper Confidence Bound (UCB):* Allocates exploration based on uncertainty around estimated action values, giving more emphasis to actions with higher uncertainty.
- *Softmax Exploration:* Assigns probabilities to actions based on their estimated values using a softmax function, allowing for a smooth transition from exploration to exploitation.
- *Thompson Sampling:* A Bayesian approach that samples from a posterior distribution over model parameters, influencing exploration based on uncertainty about the true underlying model.
- *Bootstrapping:* Techniques that involve bootstrapping, such as Temporal Difference learning or Q-learning, balance exploration and exploitation by updating value estimates based on both current rewards and estimated future rewards.

3.1.3 Q-Learning

Q-Learning, firstly introduced in [128], is arguably the most popular RL algorithm relying on a look-up table. This model-free and Value-based approach utilizes a table where each row corresponds to a state, and each column represents an action. The cells in the table store the estimated Q-value for each state-action pair. The update rule follows Eq. 3.11, which slightly differs from the version in Eq. 3.9 by introducing the concept of the *target action*.

$$Q(s^{<t>}, a^{<t>}) = Q(s^{<t>}, a^{<t>}) + \alpha\left[(r^{<t>} + \gamma \max_a Q(s^{<t+1>}, a^{<t+1>})) - Q(s^{<t>}, a^{<t>})\right] \tag{3.11}$$

As illustrated in Fig. 3.13, the target action is the one with the maximal Q-value out of the available actions in the next step. Importantly, this action serves as the target for updating the Q-value of the current state only, but it may not necessarily be the action executed later. In the subsequent step, the ε-greedy procedure may decide to explore a different trajectory. This characteristic makes the algorithm *off-policy*, as the actions executed can differ from those used for learning.

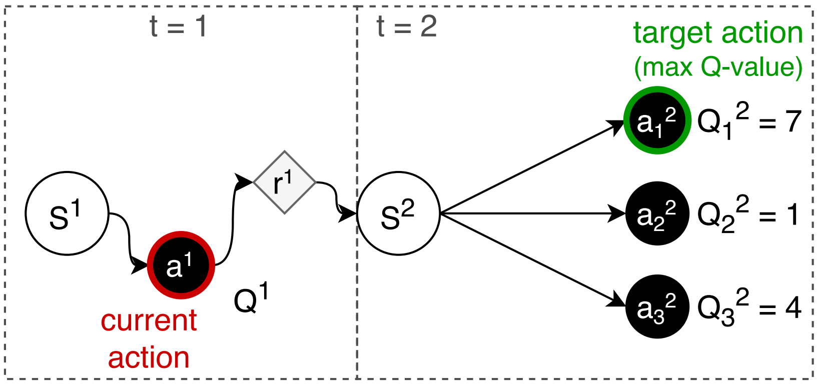


Fig. 3.13 Q-Learning: Example of the target action selection.

The Q-Learning algorithm converges to the optimal policy. From Eq. 3.11, it is evident that updates of the Q-value are based on two estimates and the *Return*. As the *Return* of the last (pre-terminal) state equals the *reward*, which is grounded in actual data, and the updates propagate back to earlier stages, the estimates become more accurate with more episodes. Ultimately, with sufficient iterations and evaluation of all possible options, the optimal Q-values are found.

3.1.4 SARSA

The name *SARSA* [114] is derived as the acronym for the quintuple State-Action-Reward-State-Action. It is an *on-policy* algorithm, implying that it learns the action-value function for the policy it is currently following. This is a key distinction from Q-Learning. The update rule for SARSA is given by Eq 3.12:

$$Q(s^{<t>},a^{<t>}) = Q(s^{<t>},a^{<t>}) + \alpha\left[(r^{<t+1>} + \gamma Q(s^{<t+1>},a^{<t+1>})) - Q(s^{<t>},a^{<t>})\right] \tag{3.12}$$

Compared to the update rule of Q-Learning (Eq. 3.11), in SARSA, the target involves the Q-value of the actual action taken $Q(s^{<t+1>},a^{<t+1>})$. This makes it more conservative and may be preferred in situations where exploration needs to be carefully controlled.

3.1.5 Deep Q-Networks

The Q-Learning algorithm (Sec. 3.1.3) faces limitations due to the restricted capacity of its table. Dealing with an extensive number of states, and potentially actions, makes it impractical to construct a table for such complex problems. Additionally, there is a need to handle continuous state spaces, which might pose challenges in encoding them into discrete states to fit the table format. In such instances, a function can replace the look-up table, and, as we know, neural networks are excellent function approximators.

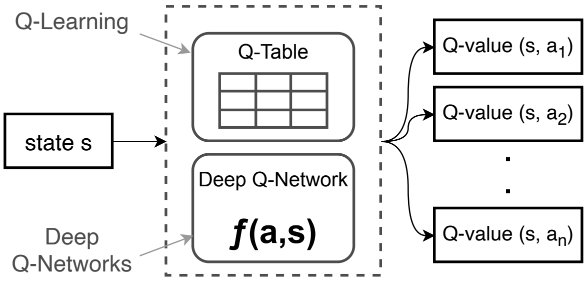


Fig. 3.14 Q-Learning vs. Deep Q-Networks.

In Deep Q-Networks (DQNs, [90]), there is the same dual-action logic as in the case of Q-Learning, so we are also talking about an off-policy algorithm. The data flow in the algorithm is illustrated in Fig. 3.15. There are two neural networks of an identical architecture: 1) *Policy Network* trained to predict the Q-value of the action taken based on the current state, resulting in the predicted Q-value, and 2) *Target Network* responsible for predicting the Q-value of the best possible action from the next state, resulting in the target Q-value.

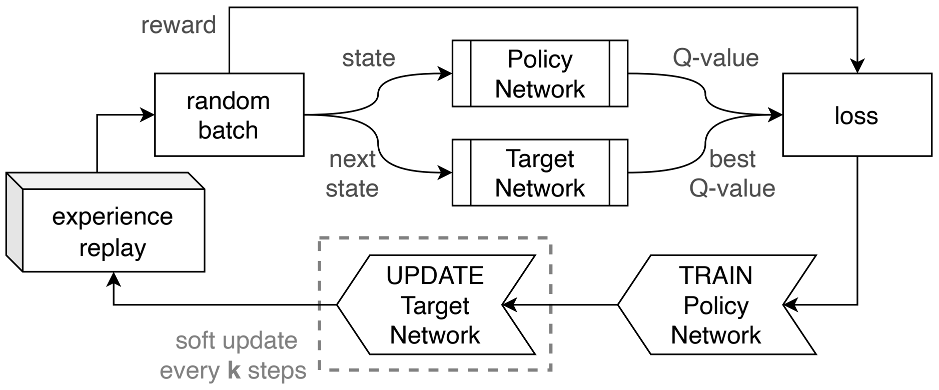


Fig. 3.15 Deep Q-Network reinforcement learning data flow.

The predicted Q-value, target Q-value, and the reward are used to compute the error (Eq. 3.13), and subsequently, Huber Loss L (Eq. 3.14) is typically used, as its nature makes it robust to outliers when the estimates are noisy.

$$\delta = Q(s^{<t>}, a^{<t>}) - (r^{<t>} + \gamma \max_a Q(s^{<t+1>}, a^{<t+1>})) \tag{3.13}$$

$$L(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{3.14}$$

In Deep Q-Networks, two crucial concepts enhance the learning process:

- *Two Identical Networks:* Only the *Policy Network* is trained, not the *Target Network*. Instead it is softly updated, achieved by copying parameters from the Policy Network every k time-steps or using Eq. 3.15 with parameter τ (e.g., $\tau = 0.005$).

$$\text{target\_net\_params} = \tau \cdot \text{policy\_net\_params} + (1 - \tau) \cdot \text{target\_net\_params} \tag{3.15}$$

  If there were no secondary (*Target*) network and both the predicted and target Q-values were computed exclusively from the *Policy Network*, it would be akin to pursuing a moving target. Employing it ensures that target Q-values remain stable, at least for a short period of time, which was showed to result in a more stable training.

- *Experience Replay:* Data accumulation in *Experience Replay* involves shuffling and random batch sampling during policy optimization. Mixing older and recent samples in the batch prevents issues like catastrophic forgetting, ensuring the network learns more robustly. The *Experience Replay* has a limited capacity.

3.1.6 Policy Gradient Algorithms

Model-free RL algorithms, as mentioned earlier, can be categorized into *Value-based* and *Policy-based*. Q-Learning (Sec. 3.1.3) and Deep Q-Networks (Sec. 3.1.5) derive their optimal policy *indirectly* from Q-values and utilize the ε-greedy approach during training. On the other hand, *Policy Gradient* algorithms learn their optimal policies *directly*, as it goes, implying that their outputs include a probability distribution of actions, not just Q-values.

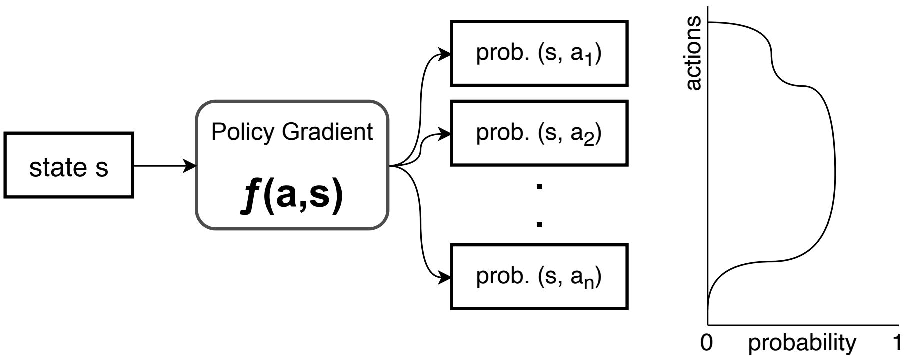


Fig. 3.16 Policy Gradient methods have a probability distribution on their output.

The training data for the network is collected from the interactions of an agent with the environment, and the gained experience (a batch of samples) is acquired once after every episode. A single sample comprises a triple (state $s^{<t>}$, action $a^{<t>}$, reward $r^{<t>}$).

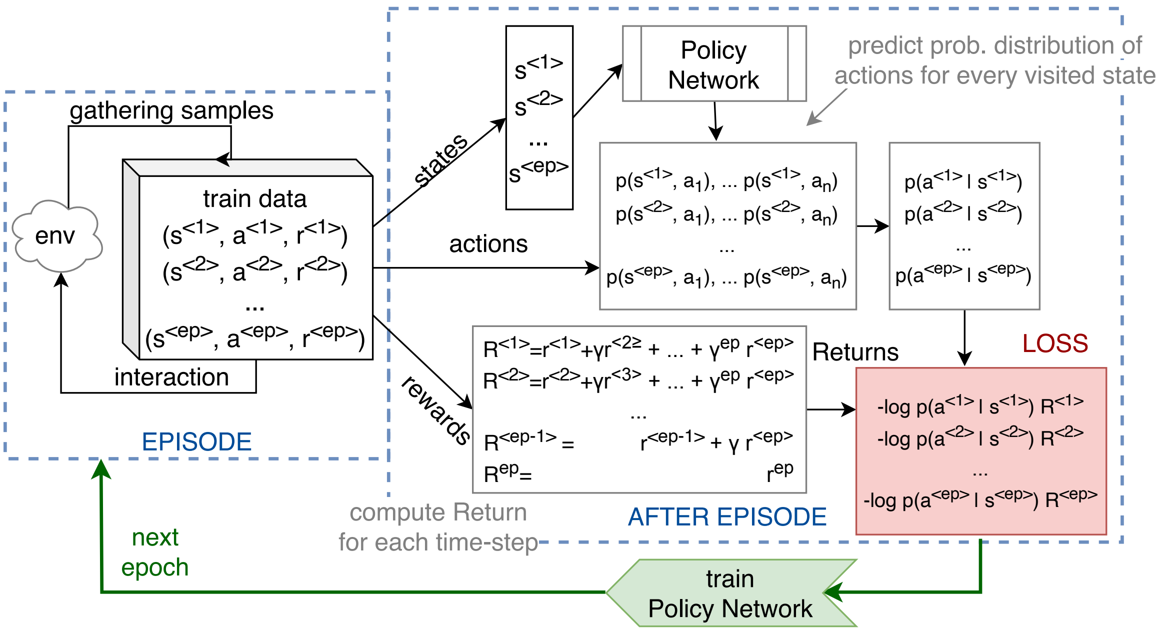


Fig. 3.17 The data flow of the policy gradient method.

A vanilla version of the loss function for network training is computed as given in Eq. 3.16, where $R^{<t>}$ (see Eq. 3.2) is the discounted Return for action $a^{<t>}$ at time-step t. The objective is to assign higher weightage to actions that resulted in higher rewards. Additionally, we aim for the probability to increase only slightly if the action taken was positively rewarded (and vice versa), which is why the log function is used. The negative sign is included because we typically consider Gradient Descent by default.

$$L(s^{<t>}) = -log[p(a^{<t>}|s^{<t>})] \cdot R^{<t>} \tag{3.16}$$

Initially, the action space is uniformly initialized, making each action equally likely, and thus, the algorithm tends to explore. In later stages, it automatically learns to exploit. However, the action is always taken by sampling from the predicted distribution. This allows actions with lower probability to be selected even in later stages, automatically addressing the explore-exploit trade-off. When employing Monte-Carlo sampling to estimate the weights of the optimal policy (through Gradient Ascent this time), the formula for gradient computation can be generalized, as shown in Eq. 3.17. The method is then commonly referred to as the *Reinforce* algorithm or the *Vanilla Policy Gradient* algorithm.

$$\nabla_\theta J(\theta) = \sum_{t=0} \nabla_\theta log\, p_\theta(a^{<t>}|s^{<t>}) \cdot R^{<t>} \tag{3.17}$$

The advantage of this method is its unbiased nature, as we utilize the true Return instead of estimating it. This approach proves particularly beneficial when dealing with continuous or a large number of actions, where DQN often faces challenges. Next, policy-based methods, in general, are usually faster in training as they find their policies directly. On the other hand, the standard Reinforce algorithm may encounter challenges associated with considerable variance in policy gradient estimation. As a result, various methods are employed to enhance its default version.

Proximal-Policy Optimization (PPO) [118]

This method improves upon vanilla policy gradient methods by introducing a clipped surrogate objective function:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}^{<t>}\left[\min\left(\rho^{<t>}(\theta)\hat{A}^{<t>}, \text{clip}\left(\rho^{<t>}(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}^{<t>}\right)\right] \tag{3.18}$$

Here, $L^{CLIP}(\theta)$ represents the clipped surrogate objective function, where $\rho^{<t>}(\theta)$ is the *Advantage* function at time t, $\hat{A}^{<t>}$ is the normalized *Advantage* at time t, clip is the

Clipping function to limit updates and ε is the clipping parameter helping the stability. PPO strikes a balance between exploration and exploitation, using multiple epochs of mini-batch updates for improved convergence. This algorithm has demonstrated effectiveness in handling complex reinforcement learning tasks, offering enhanced stability over vanilla policy gradient methods.

Trust Region Policy Optimization (TRPO) [116]

This method enhances vanilla policy gradient methods, incorporating a trust region constraint to regulate policy updates. The TRPO objective function is given by:

$$L^{TRPO}(\theta) = \hat{\mathbb{E}}^{<t>} \left[\frac{\pi_\theta(a^{<t>}|s^{<t>})}{\pi_{\text{old}}(a^{<t>}|s^{<t>})} \hat{A}^{<t>} - \beta \text{KL}\left(\pi_{\text{old}}(\cdot|s^{<t>}), \pi_\theta(\cdot|s^{<t>})\right) \right] \tag{3.19}$$

Here, $L^{TRPO}(\theta)$ represents the TRPO objective, and β is a Lagrange multiplier. TRPO ensures stable policy updates by constraining policy divergence using the KL divergence term. Despite its effectiveness, TRPO has faced criticism for its computational complexity.

3.1.7 Actor-Critic

In the case of the *Reinforce* approach, employing Monte-Carlo sampling and entire episodes to estimate the Return introduces substantial variance in policy gradient estimation [60]. Given the stochasticity of the environment (random events during an episode) and the stochasticity of the policy, trajectories can result in different Returns. As a result, the Return starting at the same state can vary significantly across episodes. To address this, the solution is to mitigate the variance by using a large number of trajectories, with the hope that the variance introduced in any one trajectory will be reduced in aggregate, providing a more accurate estimation of the return. However, it's important to note that increasing the batch size significantly reduces sample efficiency. Therefore, additional mechanisms are needed to further reduce variance.

The Actor-Critic algorithm combines policy-based (*Actor*) and value-based (*Critic*) methods in RL. The actor decided which action should be taken and critic inform the actor how good was the action and how it should adjust. A similar type of architecture can also been seen in Generative Adversarial Network (GAN, Sec. 2.5), where both discriminator and generator participate in a game. Similarly, *Actor* and *Critic* are participating in a game, but unlike GAN, both of them are improving over time. In this approach, we learn two function approximations, both represented by neural networks:

- A policy that dictates the actions of our agent (*Actor*): $\pi_0(s)$

- A value function, $\hat{q_w}(s,a)$, evaluates the quality of the taken action, providing assistance in updating the policy. This corresponds to the *Q-value*.

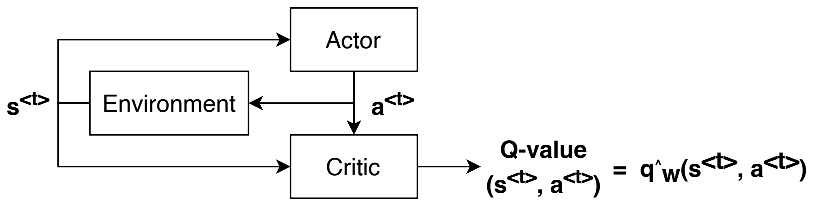


Fig. 3.18 Actor-Critic principle.

The *Actor* updates its policy parameters using the Q-value. In the default version, this is done with Eq. 3.20, where ∇_θ represents the change in policy parameters (weights) and $\hat{q}_w(s,a)$ stands for the action value estimate, i.e., the Q-value.

$$\nabla_\theta = \alpha\nabla_\theta(\log\pi_\theta(s,a))\cdot\hat{q}_w(s,a) \tag{3.20}$$

In the next time-step $(t+1)$, the *Actor*, based on its updated parameters, produces the next action to take ($a^{<t+1>}$) given the new state $s^{<t+1>}$. Simultaneously, the *Critic* updates its value parameters using Eq. 3.21. Different learning rates for policy and value are considered, using β in this case. On closer inspection, the first part of the equation corresponds to the *TDe* error (Eq. 3.7).

$$\nabla_w = \beta[r^{<t>} + \gamma\hat{q}_w(s^{<t+1>},a^{<t+1>}) - \hat{q}_w(s^{<t>},a^{<t>})]\cdot\nabla_w\hat{q}_w(s^{<t>},a^{<t>}) \tag{3.21}$$

Advantage Actor-Critic (A2C) [89]

To further stabilize learning, we can use the *Advantage* function $A(s,a)$ for the *Critic* instead of the action value function. The *Advantage* function (3.22) calculates the relative advantage of an action compared to the others possible at a state. It determines how taking a specific action at a state is better compared to the average value of the state ($V(s)$), where $Q(s,a)$ is the Q-value.

$$A(s,a) = Q(s,a) - V(s) \tag{3.22}$$

The extra reward is what goes beyond the expected value of that state. If $A(s,a) > 0$, our gradient is pushed in that direction. If $A(s,a) < 0$ (our action performs worse than the average value of that state), our gradient is pushed in the opposite direction. The challenge in implementing this advantage function lies in the need for two value functions — $Q(s,a)$

and $V(s)$. Fortunately, as long as $Q(s^{<t>}, a^{<t>}) = r^{<t>} + \gamma V(s^{<t+1>})$, we can utilize the TD error as a reliable estimator of the *Advantage* function (Eq. 3.23).

$$A(s,a) = r^{<t>} + \gamma V(s^{<t+1>}) - V(s^{<t>}) \tag{3.23}$$

The A2C method streamlines the learning process by using the *Advantage* function directly for policy updates, eliminating the need for a separate value function. This simplification not only speeds up learning but also enables efficient parallelization, making better use of computational resources. In essence, A2C is a synchronous, deterministic variant of Asynchronous Advantage Actor-Critic (A3C).

Asynchronous Advantage Actor-Critic (A3C) This upgrade extends the A2C algorithm by introducing an asynchronous training scheme for improved efficiency. In A3C, multiple agents operate concurrently, each maintaining its own actor and critic networks. These agents explore the environment independently and periodically update a global network with their experiences, facilitating shared parameter updates. The advantage function ($A(s,a)$) remains crucial for policy guidance. A3C's key contribution lies in its parallelized training approach, enabling multiple agents to explore the state space simultaneously, leading to faster convergence and enhanced sample efficiency compared to A2C.

3.1.8 Reinforcement Learning Frameworks

Some of the best libraries for working with reinforcement learning include:

- *OpenAI Gym* [18] is an open source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API.
- *Stable Baselines* [48] is a set of improved implementations of Reinforcement Learning (RL) algorithms based on OpenAI Baselines.
- *Torch-RL* [103] is an open-source RL library for PyTorch.
- *Dopamine* [21] is a research framework for fast prototyping of reinforcement learning algorithms, developed by Google Research.
- *Ray RLlib* [77] is an open-source library for reinforcement learning (RL), offering support for production-level, highly distributed RL workloads, while maintaining unified and simple APIs for a large variety of industry applications.

3.2 Multi-Agent Systems

Each independent unit capable of decision making is generally rated or judged by the way it chooses its moves (actions) over time. That unit can be a human, a single cell or even an artificial system of any complexity, where the complexity can be given by the set of available actions. Every time the selected action is taken, the state of the unit is updated and moreover, other nearby units as well as the entire environment the units operate in can be affected. This is related to the known decision-making theorem popularized in [94] that highlights the importance of taking strategies of your rivals into account, when building own strategy. The overall score (with respect to the common goal of a group of units) is maximized in case of the *Nash equilibrium*, which happens when none of the units wishes to adjust its strategy even if each knows strategies of all other units.

The term *Multi-Agent System* (MAS) refers to any environment containing multiple independent units (agents) that interact with each other and with the environment (Fig. 3.19). The theory is general and applicable to many domains, for example human teams (companies), distributed software systems or communication networks.

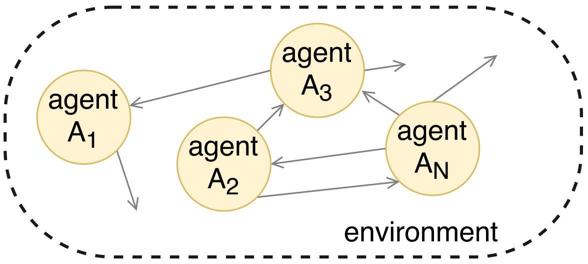


Fig. 3.19 Multi-agent system - *N* agents interacting with each other and with the environment.

The key feature of such systems is the *emergence principle*, a phenomenon that occurs when a system is observed to have properties its parts (agents) do not have on their own and this unexpected behaviour emerges only when the parts interact in a wider whole. This way the agents together are capable of solving problems of complexity beyond the knowledge and abilities of their own separately. The common behaviour of the system can even show signs of *intelligence* despite the primitive nature of single units. The primary characteristics of a multi-agent system include:

- *Local view*: Agents have a local view only, the whole system and the addressed problem are too complex for them.
- *Emergent behaviour:* Interactions among agents lead to emergent behavior—patterns arising collectively but not explicitly programmed.

- *Autonomy*: Agents are independent, self-aware and (to some extend) autonomous entities with their own decision-making processes and actions.
- *Distributed Nature:* Multi-agent systems are inherently distributed, with decentralized decision-making. None of the agents is designated as the one in charge.
- *Interaction:* Agents interact with each other and their environment, involving communication, coordination, cooperation, or competition.
- *Perception:* Agents have the ability to perceive and gather information about their environment.
- *Adaptivity:* Multi-agent systems are designed to be adaptable and flexible, allowing agents to respond to changes.
- *Concurrency:* Agents operate concurrently, enabling parallelism and efficient resource utilization. By default, operations are executed asynchronously.
- *Heterogeneity:* Agents may (or may not) exhibit heterogeneity in capabilities, knowledge, or roles.
- *Decomposition* - complex tasks are decomposed to elementary tasks addressed by elementary components.

A solid overview of multi-agent technologies is presented in [33]. In general, the interest in multi-agent systems has been increasing lately. Typically, they are useful for projects, where more entities have to cooperate, projects based on distributed systems or projects, where conventional methods become inconvenient (for instance, caused by limits of a single unit). They are predicted to be widely used in the future.

3.2.1 Multi-Agent Reinforcement Learning

The standard reinforcement learning principle (see Fig. 3.2) is defaultly derived for training a single agent in the environment. Analogically, it can be applied to multiple agents in the same environment, which turns the process into a *Multi-Agent Reinforcement Learning (MARL)* problem. A solid review of MARL systems is provided in [134]. Regarding this work, MARL has several design choices to be considered.

Shared vs. individual policy. In the context of MAS, there is a choice between shared and individual policies for the agents. As illustrated in Fig. 3.20a, all the agents can share a single decision making system (policy π), which is implemented for example in [110] - see Sec. 2.7.3, or each agent can follow its own individual strategy (depicted in Fig. 3.20b).

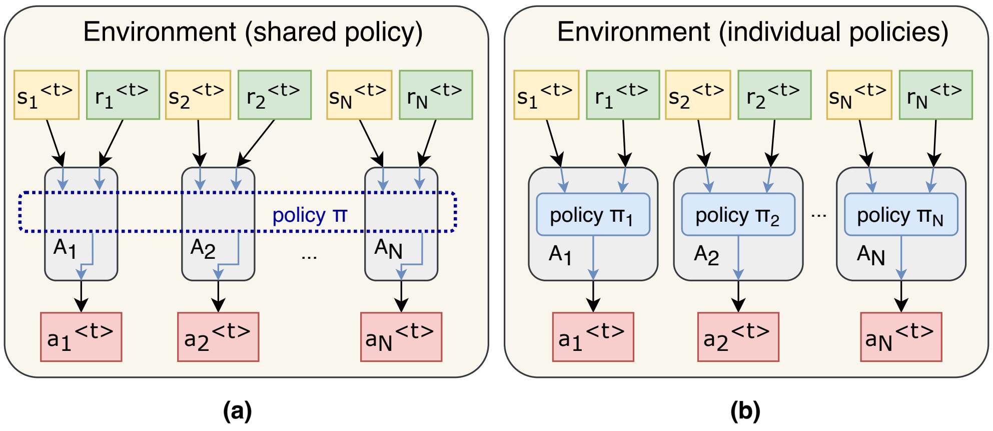

Environment (shared policy)
$s_1^{<t>}$ $r_1^{<t>}$ $s_2^{<t>}$ $r_2^{<t>}$ $s_N^{<t>}$ $r_N^{<t>}$
policy π
A_1 A_2 ... A_N
$a_1^{<t>}$ $a_2^{<t>}$ $a_N^{<t>}$
(a)
Environment (individual policies)
$s_1^{<t>}$ $r_1^{<t>}$ $s_2^{<t>}$ $r_2^{<t>}$ $s_N^{<t>}$ $r_N^{<t>}$
policy π_1 policy π_2 ... policy π_N
A_1 A_2 A_N
$a_1^{<t>}$ $a_2^{<t>}$ $a_N^{<t>}$
(b)

Fig. 3.20 MARL: (a) shared policy vs. (b) individual policies.

Joint vs. individual actions, states and rewards. Figure 3.21 provides an overview of possible design choices related to the definition of actions, states, and rewards in Multi-Agent Systems (MAS). In both illustrations ((a) and (b) in Fig. 3.21), actions a_i can be considered and applied either individually (depicted in red) or compiled into a single joint action a (depicted in purple). Additionally, a global state s of the environment is commonly defined (depicted in picture (a) in the figure), but the task can also be designed with multiple independent states s_i for each agent individually (depicted in picture (b)). Similarly, there are various possibilities for defining the reward.

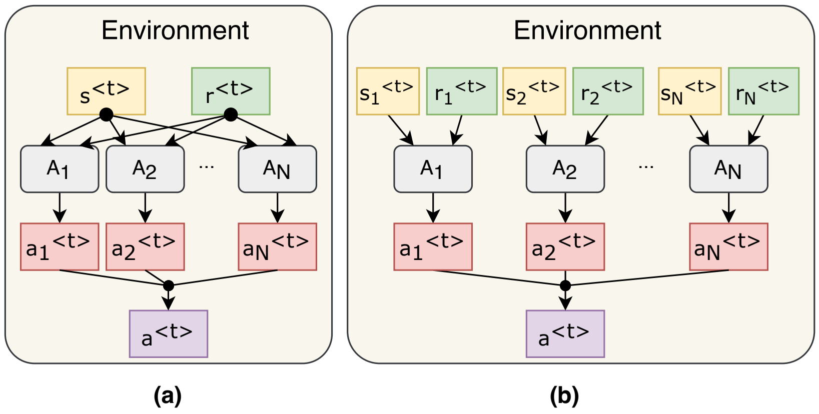

Environment
$s^{<t>}$ $r^{<t>}$
A_1 A_2 ... A_N
$a_1^{<t>}$ $a_2^{<t>}$ $a_N^{<t>}$
$a^{<t>}$
(a)
Environment
$s_1^{<t>}$ $r_1^{<t>}$ $s_2^{<t>}$ $r_2^{<t>}$ $s_N^{<t>}$ $r_N^{<t>}$
A_1 A_2 ... A_N
$a_1^{<t>}$ $a_2^{<t>}$ $a_N^{<t>}$
$a^{<t>}$
(b)

Fig. 3.21 MARL: Joint vs. individual actions, states and rewards.

Synchronization. Finally, when agents cooperate or compete within the same environment, their interactions with each other become crucial. The synchronization and order in which they take potentially individual actions and update states significantly impact global performance. Three possible synchronization methods are illustrated in Fig. 3.22:

(a) Actions are executed at the same time (t-1) as well as the consecutive states are generated together at the next time step t. In this case, the agents are synchronized and the process is independent of their order.

(b) The agents are processed one by one. Each takes its action and receives a new state immediately before the following agent makes its action. This mode is dependent of agents order, but the new state of each agent depends on his last action only.

(c) On the contrary, the last approach generates new states after the action cycle of all agents and therefore, the states are influenced by actions of the others as well.

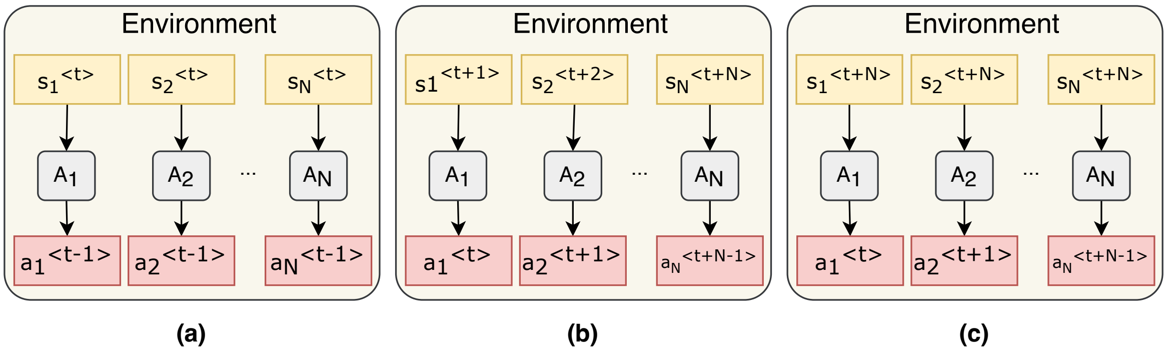

Environment
$s_1^{<t>}$ $s_2^{<t>}$ $s_N^{<t>}$
A_1 A_2 ... A_N
$a_1^{<t-1>}$ $a_2^{<t-1>}$ $a_N^{<t-1>}$
Environment
$s_1^{<t+1>}$ $s_2^{<t+2>}$ $s_N^{<t+N>}$
A_1 A_2 ... A_N
$a_1^{<t>}$ $a_2^{<t+1>}$ $a_N^{<t+N-1>}$
Environment
$s_1^{<t+N>}$ $s_2^{<t+N>}$ $s_N^{<t+N>}$
A_1 A_2 ... A_N
$a_1^{<t>}$ $a_2^{<t+1>}$ $a_N^{<t+N-1>}$
(a)
(b)
(c)

Fig. 3.22 MARL: Multiple variants of agents synchronization.

3.2.2 MARL Frameworks

- *NetLogo* [131] is a multi-agent programmable modeling environment. It is used by many hundreds of thousands of students, teachers, and researchers worldwide.
- *Vectorized Multi-Agent Simulator (VMAS)* [13] is a vectorized framework designed for efficient MARL benchmarking.
- *BenchMARL* [14] is a MARL training library created to enable reproducibility and benchmarking across different MARL algorithms and environments.
- *ML-Agents Toolkit* [66] is an open-source project that enables games and simulations to serve as environments for training intelligent agents, developed by Unity.

3.3 Genetic Algorithms

Genetic algorithm (GA) is a powerful optimization and search method inspired by the process of natural selection. Developed by John Holland in the 1960s, GAs are widely used in solving complex problems across various domains, from engineering and biology to finance and also AI. The fundamental idea behind GA is to simulate the process of evolution to find optimal solutions to a problem. The terminology includes:

- *Chromosome:* In the context of GA, a chromosome represents a potential solution to the problem at hand. It is often (but not necesarilly) encoded as a string of binary digits, with each bit representing a specific aspect or parameter of the solution.
- *Population:* A population consists of a group of potential solutions – chromosomes. The diversity within the population allows genetic algorithms to explore a broader solution space, increasing the likelihood of finding an optimal solution.
- *Genes:* Genes are the components of a chromosome that encode specific features or parameters. These can represent different characteristics of a potential solution, such as numerical values, binary strings, or discrete options.
- *Fitness Function:* The fitness function evaluates how well a particular chromosome solves the problem at hand. It assigns a numerical value to each chromosome, indicating its performance. The goal is to maximize or minimize this value based on the nature of the optimization problem.
- *Crossover (Recombination):* Crossover is the genetic operation that combines the genetic material of two parent chromosomes to produce one or more offspring. This process mimics the crossover of genetic material during reproduction in nature.
- *Mutation:* Mutation involves making small random changes to a chromosome. This operation introduces diversity into the population and helps prevent premature convergence to suboptimal solutions.
- *Selection:* Selection determines which chromosomes will be chosen as parents for the next generation. Fit chromosomes, as determined by the fitness function, are more likely to be selected, mirroring the principle of natural selection.

The algorithm is illustrated in Fig. 3.23. In practice, there are several design choices, such as appropriately setting the *Fitness* function or determining the structure of the chromosome (the numerical representation of an entity) meant to represent the final solution of the problem.

Additionally, there is a dilemma concerning which entities should be selected for *crossover* as well as for *mutation*. Relying solely on the *Fitness* function throughout may result in overlooking crucial parts of the search space. Finally, numerous options exist for designing the *crossover* and *mutation* operations themselves.

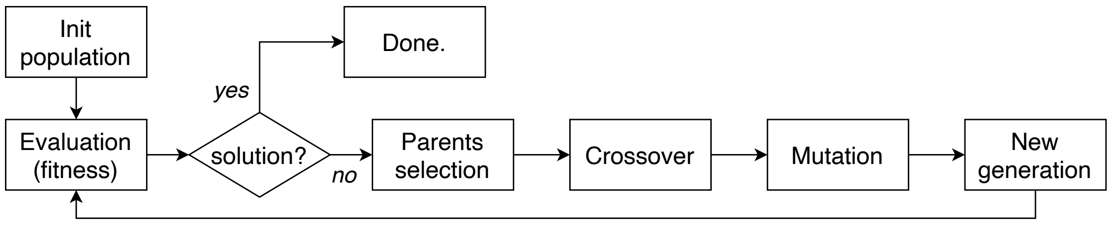


Fig. 3.23 Diagram of the Genetic Algorithm workflow.

In summary, the capability of genetic algorithms to explore complex solution spaces and discover near-optimal solutions becomes valuable in scenarios where traditional optimization techniques may fail. With their ability to adapt to multiple diverse tasks, genetic algorithms remain a valuable tool for addressing complex challenges across various domains.

3.4 Human-in-the-Loop

The human-in-the-loop (HITL) methodology involves integrating human feedback or oversight into a machine learning system, creating a symbiotic relationship between human intelligence and machine capabilities. Humans actively guide, supervise, and offer crucial feedback and corrections to an otherwise automated system. It is typically deployed in tasks involving human-machine interaction. The principle is illustrated in Fig. 3.24.

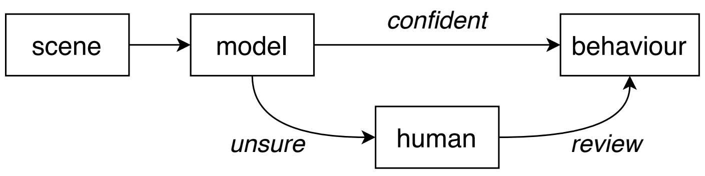


Fig. 3.24 The human-in-the-loop (HITL) principle.

Broadly, we can define two categories based on the party leading the interaction [91]:

- *Active learning*: The system takes charge of the interaction, viewing the user as a means to annotate unlabeled data for its machine learning algorithms.
- *Machine teaching*: The user leads the interaction, determining the type of data and knowledge they wish to impart to the machine learning algorithms of the system.

Chapter 4

Multi-Agent based Neural Networks

Having extensively covered neural network architectures and explored various learning optimization techniques in preceding chapters – including an analysis of optimization algorithms rooted in RL and multi-agent systems – this chapter now shifts the focus. Here, we delve into the second segment of this study: the introduction of the proposed methodology for constructing neural networks from scratch. At first, the problem formulation of this study is put. Then, the key ideas and concepts are presented, leading us to the description of the developed algorithm and opening the research questions.

4.1 Problem Formulation

Within the broad spectrum of machine learning challenges and despite its potential applicability across other categories, the proposed method has been developed and demonstrated to address classification problems of non-sequential data (dressed in yellow in Fig. 4.1).

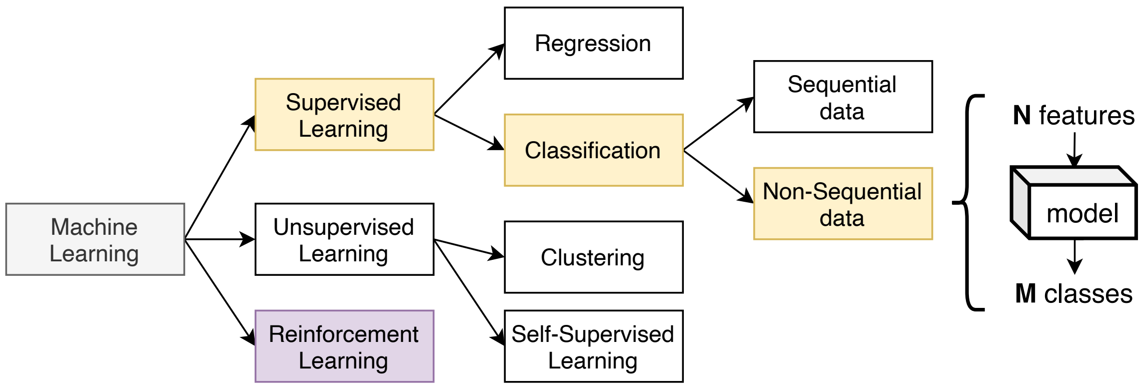


Fig. 4.1 Depiction of the proposed method's focus within machine learning (highlighted in yellow). Reinforcement learning principles (purple) are leveraged as a supportive tool.

This problem is characterized by the number of input features N, the number of output classes M, and a dataset comprising S samples. The goal of the method is to produce a feedforward neural network architecture, called *target network* (or *Net*), capable of effectively classifying the data. That means maximizing accuracy on the dataset while simultaneously minimizing the number of parameters p in the network architecture. To provide a formal definition, we can state that given a dataset D consisting of S samples $(x_1, x_2, ..., x_S) \in \mathbb{R}^N$ and their corresponding targets $(y_1, y_2, ..., y_S) \in \mathbb{R}^M$, the objective is to discover a model comprising p parameters capable of classifying the testing portion of the dataset D_{test} with an accuracy $\text{acc} = \frac{n_C}{n_C + n_F}$, where n_C represents the number of correctly classified samples from D_{test} and n_F denotes the number of failures. Hence, we have two criteria for a successful outcome, prioritized as follows: 1) target network accuracy; 2) number of parameters p. This study investigates primarily the following hypotheses.

Hypothesis H1 *Fully-connected network architectures include redundant synapses*. This assumption is supported in [20] demonstrating that pruned neural networks, even with a high percentage of parameters removed, perform equally well as fully-connected baselines.

Furthermore, a similar behaviour can be observed when employing the *dropout* mechanism (refer to Sec. 2.6.3 and Fig. 2.39). This technique demonstrates that randomly deactivating individual components of a network during training enhances its overall generalization capabilities.

Hypothesis H2 *A sparse (not fully-connected) network, modularized into distinct components, may offer improved explainability compared to fully-connected versions.*

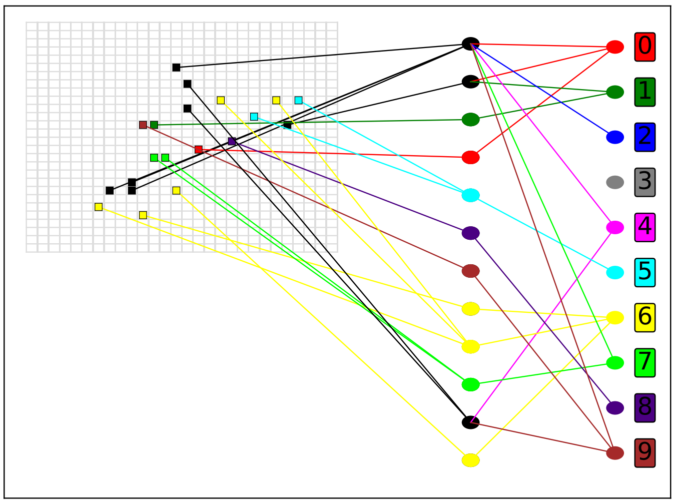


Fig. 4.2 An illustration of a pruned (and partially explainable) network, overtaken from [20].

In Fig. 4.2, a simplified example of a pruned network is illustrated, utilizing only 20 input features (out of the original 728) and 38 synapses to achieve a classification accuracy of 50% on the MNIST dataset [29]. This visualization allows tracking the flow of data from features to individual classes, indicated by colors. Consequently, we can elaborate on which parts of the network handle specific information and how different features influence various classes.

Research basis At this stage, we reference the *Universal Approximation Theorem* detailed in [26] to underscore the motivation for tackling the challenge of neural architecture search. Let ϕ be any non-constant, bounded, and monotonically-increasing continuous function (activation function), and let $\sigma(x)$ represent the output function of a neural network with a single hidden layer. Then, for any continuous function f defined on a compact subset K of $\mathbb{R}^n$ and any $\varepsilon > 0$, there exists a neural network with weights and biases such that:

$$\left|\sigma(x) - f(x)\right| < \varepsilon \tag{4.1}$$

for all x in K. This statement asserts that a feedforward neural network with a single hidden layer, comprising a finite number of neurons, can approximate any continuous function on a compact subset of Euclidean space to any desired degree of accuracy, provided a sufficiently large number of neurons. Additionally, in 1991, Hornik demonstrated [57] that it is not the specific choice of the activation function, but rather the multilayer feed-forward *architecture* itself, that gives neural networks the potential of being universal approximators. And this is a strong motivation for further research.

Inapplicability of brute-force techniques Assuming a feedforward neural network (illustrated in Fig. 2.3) with $N = 2$ input neurons, one hidden layer of r neurons, and one output neuron ($M = 1$), the potential number of parameters is given by $p_M = N \cdot r + r + r \cdot M + M$.

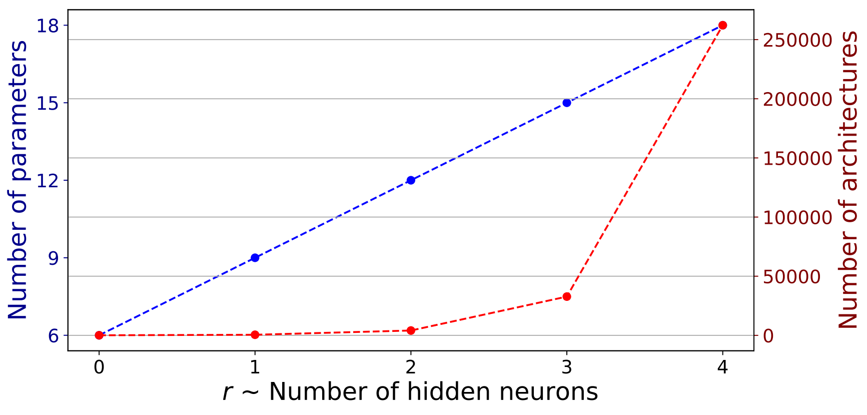


Fig. 4.3 Number of architecture variants (right y-axis / red) vs. number of hidden neurons.

Given that each parameter (synapse) can either be present (ON) or absent (OFF) in sparse networks, leading to two states for each, the total number of potential network architecture variations (i.e., the size of the state space) is $n_S = 2^{p_M}$. This poses computational infeasibility for brute-force algorithms, even with a small number of hidden neurons ($r < 10$), as depicted in Fig. 4.3, and particularly for deep, commonly utilized structures. Hence, optimization techniques are needed to address this challenge.

4.2 Key Concepts

In this section, we introduce the key ideas that form the foundation and contribute to the uniqueness of the proposed algorithm for neural architecture search (Sec. 4.3). The algorithm is composed as a combination of the following concepts.

4.2.1 No Layers

As previously specified in the preceding section, the target network architecture is constrained to be feedforward, given by the condition in Eq. 2.1. However, in addition to the standard layer-wise approach (Fig. 4.4a), this work introduces a more flexible data flow by eliminating the organization to layers. Neurons are granted additional freedom by allowing connections to any other neuron that activates later in time, as indicated by their vertical positions in Fig. 4.4b. The implementation ensures there are no loops in the data flow.

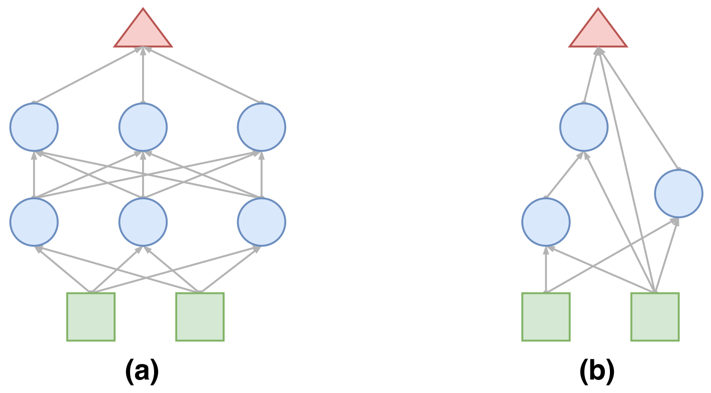


Fig. 4.4 Elimination of layers: a) standard layer-wise approach; b) proposed approach.

This approach aligns with the concept of *skipping connections* employed in *Residual networks*, which has demonstrated efficiency and is frequently utilized (e.g., in the Transformer block - see Fig. 2.29). At the same time, it adheres to the biological model, as there are also no layers in the human brain.

4.2.2 Network Architecture Representation

To implement the layer-free architecture shown in the previous section (4.2.1), we introduce the concept of a *Grid*. While it may remind us of the conventional layers-wise approach, it serves primarily for implementation purposes. This way, we can still utilize fast matrix computations for both forward and backward passes through the network, while simultaneously accommodating their sparsity and connectivity potential introduced in Fig. 4.4b.

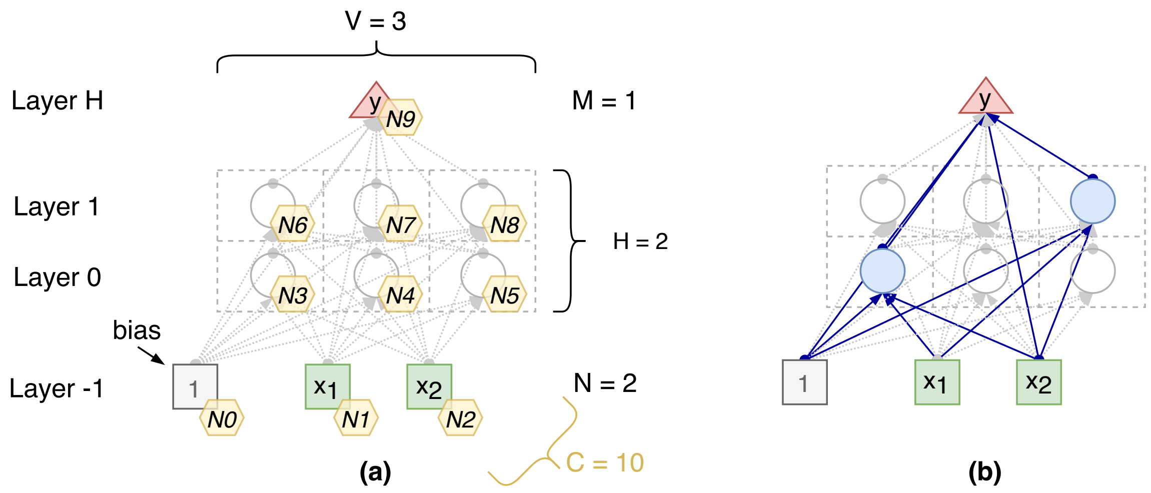


Fig. 4.5 The concept of a *Grid* defining the target network capacity.

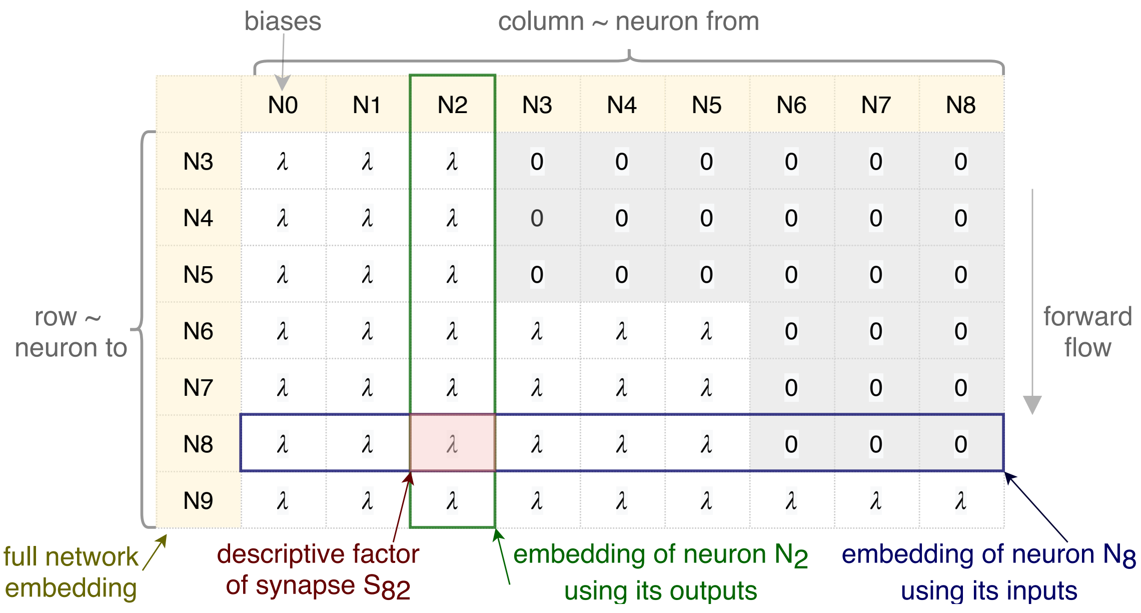
The *Grid* concept is illustrated in Fig. 4.5a (left), while on the right (Fig. 4.5b), we see a randomly selected example of a single architecture variant. Importantly, we prepend one more feature that is always 1 to the feature vector on the first position. By connecting it to all the neurons in the grid, we implement the bias like this, while we keep the intuition of synapses corresponding to individual parameters. Each cell in the *Grid*, including the one for biases and the input neurons, has a number (indicated by the little yellow boxes in Fig. 4.5a). The total number of cells in a *Grid* is denoted as C (in the example, including bias, $C = 10$).

Beyond facilitating matrix computations, the *Grid* also establishes important parameters for the optimization task. The number of samples N, as well as the number of classes M, is determined by the specific classification problem defined by its dataset D. Next, we define two parameters that characterize the capacity of the target network and the maximum number of its parameters. The height of the *Grid* is denoted by H ($H = 2$ in the selected example in Fig. 4.5) and in standard approaches it would correspond to the maximum number of hidden layers. The width of the *Grid* is represented by V ($V = 3$ in Fig. 4.5), aligning with the number of hidden neurons in a single layer. The first of these "layers," as we refer to them for the sake of matrix computations and as depicted in Fig. 4.5, is labeled with -1

and corresponds to the vector of input neurons carrying sample features, including the zero position for biases. With this configuration, H (corresponding to the height of the *Grid*) corresponds to the "layer" of output neurons (activated as the latest). With this, we can define the maximum potential number of parameters p_M (capacity) of the configured *Grid* as:

$$p_M = p_i + po + ph = (N+1) \cdot (H \cdot V + M) + M \cdot H \cdot V + \frac{H \cdot (H-1)}{2} \cdot V^2 \tag{4.2}$$

Here, p_i is the number of synapses coming from input neurons (including the biases), p_o signifies the total number of synapses connected from hidden neurons to the output cells, and p_h denotes the number of synapses within the *Grid* body (not linked to either input or output neurons). For the example in Fig. 4.5, the network capacity is computed as $p_M = 21 + 6 + 9 = 36$.


biases
column ~ neuron from
row ~ neuron to
forward flow
full network embedding
descriptive factor of synapse S82
embedding of neuron N2 using its outputs
embedding of neuron N8 using its inputs

	N0	N1	N2	N3	N4	N5	N6	N7	N8
N3	λ	λ	λ	0	0	0	0	0	0
N4	λ	λ	λ	0	0	0	0	0	0
N5	λ	λ	λ	0	0	0	0	0	0
N6	λ	λ	λ	λ	λ	λ	0	0	0
N7	λ	λ	λ	λ	λ	λ	0	0	0
N8	λ	λ	λ	λ	λ	λ	0	0	0
N9	λ	λ	λ	λ	λ	λ	λ	λ	λ

Fig. 4.6 The overall network architecture representation using a matrix embedding.

Using the defined notation and still considering the *Grid* example illustrated in Fig. 4.5a, the network architecture can be fully described by a matrix introduced in Fig. 4.6. Here, λ serves as a descriptive factor of arbitrary nature for parameter (synapse) representation. By default, when determining whether the corresponding synapse is present (ON) or absent (OFF), we use $\lambda \in \{0,1\}$. Alternatively, synapse weight or other descriptive factors can be employed, as will be discussed later in this work. The forward flow is manually ensured in the top-right part of the matrix by strictly setting the values to zero, as these connections cannot exist.

4.2.3 Multi-Agent System Analogy

The development of neural network architectures has been inspired by the biological template (see Sec. 2.1). Interestingly, in contrast of the amazing capabilities of a biological brain as a whole, the functionality of a single cell (Fig. 2.1) is pretty elementary. Its responsibility is virtually just to sum up the incoming signals and to decide whether to *fire* (activate itself) or not, while the cell itself is definitely not capable of understanding or even observing the complex behaviour of the whole brain. Accordingly, what makes the brain such a powerful machine is the enormous number of the elementary units and the way they interact.

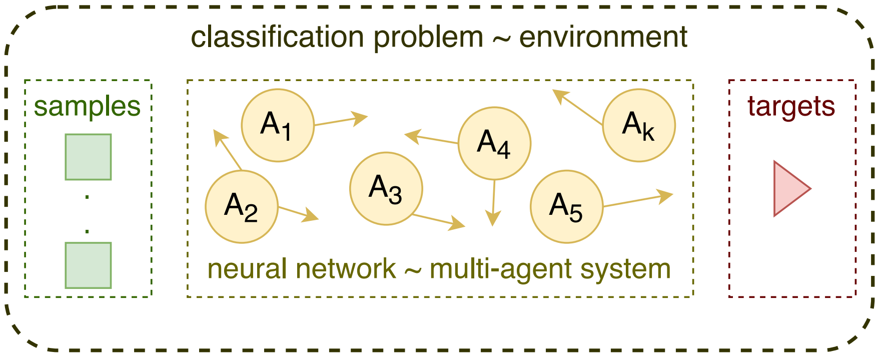

classification problem ~ environment
samples
targets
A_1
A_2
A_3
A_4
A_5
A_k
neural network ~ multi-agent system

Fig. 4.7 Neural network viewed as a multi-agent system.

Following the principles of *multi-agent systems* (see Sec. 3.2), we find a perfect analogy here. Independent agents capable of primitive actions interact with each other and the environment. These agents possess only a local view, and the problem addressed by the entire system is too complex for any single agent to comprehend. However, when many agents collaborate, the system can exhibit intelligent behavior, effectively tackling a globally complex task. Moreover, the *adaptive* nature of multi-agent systems aligns with the ability of neural networks to learn from new data samples. A crucial aspect explored and leveraged in this work is the *decomposition* of complex tasks and the *modularity* of network architecture.

4.2.4 Self-Attention of Multi-Agent Systems

Expanding upon the theory presented in the previous section of viewing a neural network architecture as a multi-agent system, a significant decision arises regarding the definition of a single agent within the network. During the algorithm's development, various configurations were considered. For instance, initially, a single neural cell was deemed the single agent. However, it was ultimately decided to define an agent as a single synapse (i.e., a single parameter of the network). Hence, the concept presented in this section is illustrated using this configuration, although the principle is general and potentially adaptable to multiple different configurations.

In a multi-agent system like this, the agents are expected to interact, collaborate, and possibly communicate with each other to collectively tackle the shared complex task of constructing the network architecture. To guide each agent in decision-making within the RL algorithm at a particular time-step, designing a well-descriptive state representation is crucial. This representation should capture the current state of the environment (i.e., how effectively the problem at hand is currently classified). Additionally, what can aid each agent is the relationship to the current states of other agents. As the number of agents in the system can be very large, we seek an elegant and computationally efficient approach to relate individual agents. This is where we attempt to apply the *Attention mechanism* (detailed in Sec. 2.4.1), originally designed primarily for NLP tasks to establish relationships between individual words in text, even in very long passages.

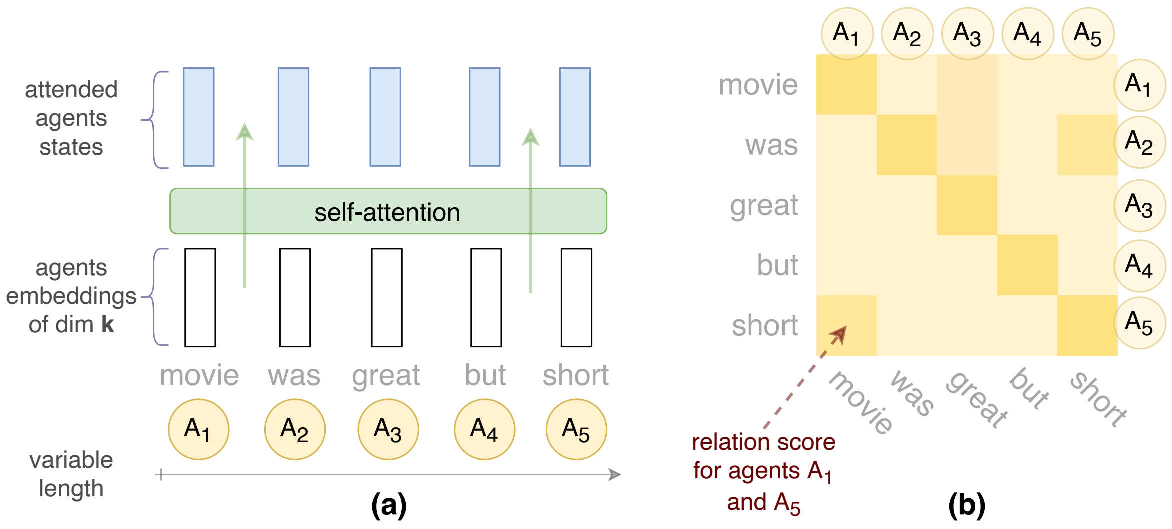


Fig. 4.8 Idea of employing the *Attention mechanism* to map agents relations: (a) Attending embeddings of agents analogically to a NLP task; (b) Attention scores $\sim$ agents relations.

In Fig. 4.8, the analogy with the NLP task is depicted. Just as we can create an embedding (along with positional encoding) of a word within a sentence, we can similarly arrange the agents of a network in a row and embed them into descriptive vector representations. There are several advantageous aspects here with significant potential:

- The sequence length (i.e., the number of agents in a network) can be very long.
- Computing relations among agents during the forward pass is very fast, involving just a few matrix multiplications.

- The representations and relations can be trained using *query*, *key*, and *value* matrices.
- With this configuration, we can compute the *self-attention* of a single network. Additionally, we can attend to agents of one network with the agents of another using the *cross-attention* mechanism with the query substitution trick described in Fig. 2.30.
- To enhance the approach's capabilities, we can transition to the *multi-head attention*, aiming to identify and represent multiple patterns among the agents. These patterns could potentially relate to solving individual classes within the classification task.

This enables the agents in the network to have awareness of each other's situations. The pivotal design choice here is the formation of a single agent's embedding, represented by a vector of dimensionality *k*. This aspect is one of the research questions further discussed in subsequent sections.

4.2.5 Employing Genetic Principles

Referring back to Sec. 3.3, which introduced another optimization technique commonly known as *Genetic algorithms*, we now demonstrate that several principles from this domain can also be applied. Once again, we identify several analogies. In a genetic algorithm, the iterative loop seeks a *chromosome* that represents the solution to the given problem. This chromosome consists of individual *genes*, formed into a representation, typically illustrated as shown in Fig. 4.9. Relating this terminology to the task addressed in this work, we can map the chromosome as the network architecture we aim to design, where genes correspond to individual agents in our network.

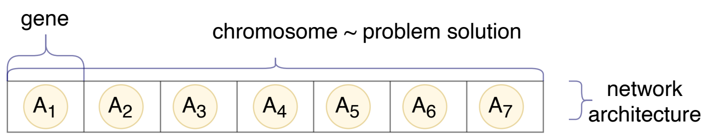


Fig. 4.9 Idea of applying genetic algorithms principles for neural architecture search.

With this setup, we can apply principles like the *Fitness* function to evaluate the qualities of individual solutions (network architectures). Additionally, the *mutation* operation, involving a (typically random) alteration in the *genes* over generations, aligns well with changes in the network architecture. However, instead of being random, these changes could be more sophisticatedly induced using agents' decisions. Finally, there is potential for the *crossover* operation of *chromosomes* (architectures), where the *cross-attention* mechanism mentioned in the previous section (4.2.4) can be a valuable tool. Of course, there are numerous research questions in this area that will be further elaborated upon later.

4.3 The Algorithm

This section presents the core of the work: an iterative algorithm integrating the concepts introduced in Sec. 4.2. An overall view of the developed framework is shown in Fig. 4.10.

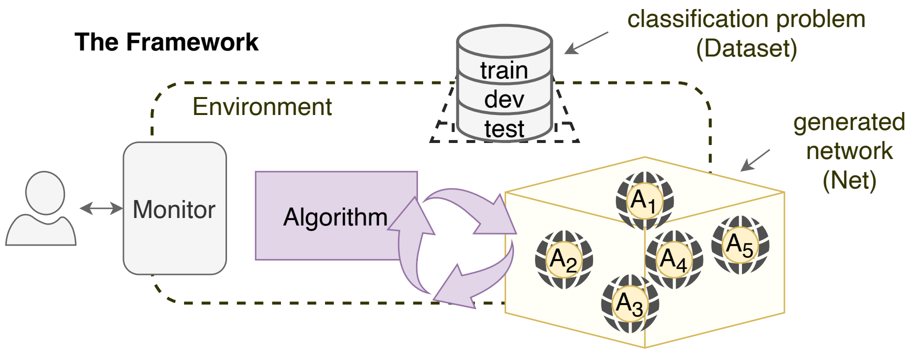


Fig. 4.10 An overall view of the developed framework.

At this point, it is fitting to recapitulate the analogies and fundamental principles previously discussed. Returning to the problem formulation, we have a classification task represented by a dataset (with dimensionality N and M classes), and we aim to design a network architecture (referred to as *Net*) that maximizes classification accuracy (*acc*) while minimizing the number of parameters p. Drawing from the ideas introduced in Sec. 4.2, we consider the following:

- A predefined *Grid* structure (see Fig. 4.5) is characterized by its width V and height H. It ensures a forward data flow and grants neurons freedom in connectivity by eliminating traditional layers. The capacity of the network p_M is defined by Eq. 4.2.
- The classification problem, seamlessly integrated into the framework like a floppy disc, defines the *Environment* of a multi-agent system.
- The generated network architecture (*Net*) corresponds to a *solution* to the problem at hand, analogous to an individual *entity (chromosome)* within a genetic algorithm population. This entity's characteristics, such as the number of parameters p, classification accuracy *acc*, and loss L, can be formed into a *fitness* function (more in Sec. 4.3.2).
- At the same time, *Net* embodies a *multi-agent system.* A single synapse (parameter) of the network represents an individual agent.
- Additionally, an integrated *Monitor*, represented by a graphical interface, facilitates asynchronous communication while the algorithm is in progress.

Now, further specifications of the generated network design are needed. The implementation is carried out in PyTorch [99] by designing a custom module tailored to the needs of the proposed framework. The ReLU activation function, depicted in Figure 2.4, is selected for all cells in the generated network – recent research [98] published in 2021, focusing on the Universal Approximation Theorem, confirms its suitability. The *AdamW* optimizer is employed, implementing the *decoupled weight decay regularization* technique [82], with an initial learning rate set to $lr = 0.07$. Finally, we choose out of two loss functions:

- We use the *categorical cross-entropy loss* (Eq. 2.43) for problems with $M > 2$.
- Alternatively, to ensure the algorithm can generate the minimal possible network architecture, a special trick is employed for classification problems of two classes. Here, a single output neuron with thresholding is utilized, allowing for binary classification using the *binary cross-entropy loss* (Eq. 2.42). In this scenario, the parameter M is manually set to 1 to align with the configuration described in Eq. 4.2, which computes the number of parameters based on M. Thus, in case of two classes, we have $M = 1$.

The main loop of the algorithm follows the principle of Genetic Algorithms (GA, Sec.3.3), albeit with slight adjustments from the typical version shown in Fig. 3.23. The primary distinction lies in skipping the *crossover* operation in this version of the algorithm, although its potential inclusion is discussed in Sec. 4.4. Illustrated in Fig.4.11, the algorithm initiates at time-step $t = 0$ (yellow box). Subsequently, the iterative optimization algorithm performs its search for the target network architecture (*Net*). Each iteration corresponds to a single *generation* in GA terminology and also represents a single episode of a RL algorithm, which is hidden within the purple *mutation* box and elaborated upon in detail in Sec. 4.3.5. The loop terminates upon meeting the specified termination condition, indicating the selection of a suitable candidate of target network architecture (green box).

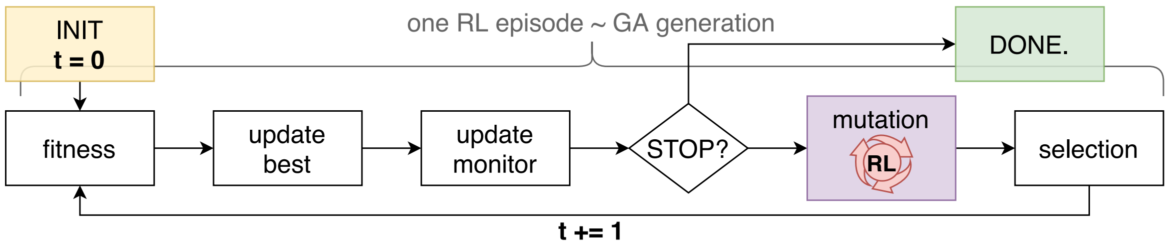


Fig. 4.11 The main loop of the algorithm based on the GA principle.

In App. B.3, there is the default configuration, aligning with the blocks depicted in Fig. 4.11 and their parametrization. Each of these blocks is now detailed in the following sections.

4.3.1 Initialization

The algorithm begins by setting the initial population with predefined number of entities, denoted as G (defaultly set to $G = 4$) and differing in the randomly set values of their parameters. For the purpose of constructing the network architecture from scratch, two methods (Fig. 4.12) were implemented: a) *zero*: no synapses are present (newborns have no parameters, default); b) *cherries*: initialize synapses connecting all input neurons (including the bias) to all output neurons (newborn entities contain $p = (N+1) \cdot M$ parameters).

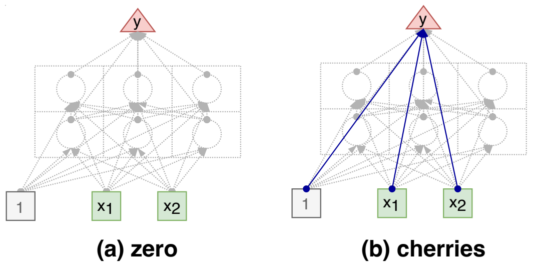
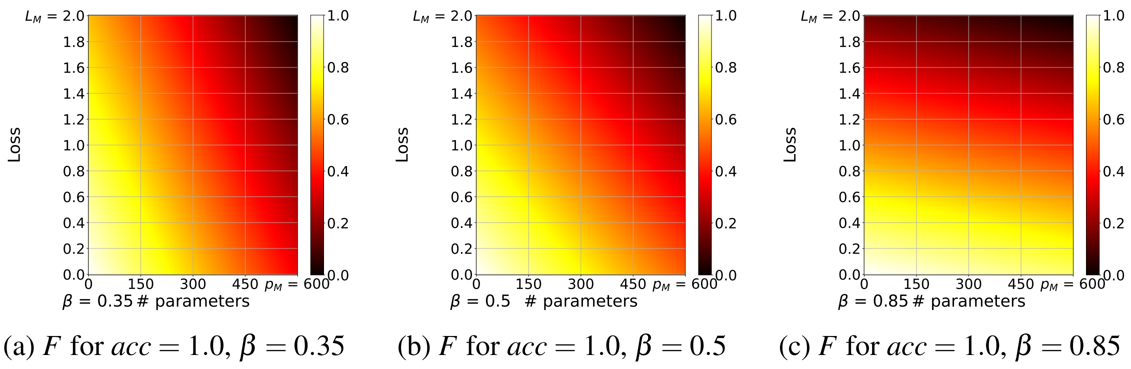


Fig. 4.12 Algorithm initialization methods - (b) example: $N = 2$ and $M = 1$ leads to $p = 3$.

4.3.2 Fitness Function

The fitness function evaluates individual entities (network architectures) based on their ability to maximize classification accuracy while minimizing the number of parameters. Besides the accuracy *acc*, we also employ a relative version of current loss L with the default maximal loss $L_M = 2$ if not specified otherwise by the problem in hand. With p representing the number of parameters and p_M denoting the capacity of the *Grid*, the fitness function is given by Eq. 4.3. Meta-parameter $\beta \in [0,1]$ (defaultly $\beta = 0.6$) determines the trade-off between the two optimized qualities and its impact is demonstrated in Fig. 4.13.

$$F = acc \cdot (1 - (\beta \cdot \frac{L}{L_M} + (1-\beta) \cdot \frac{p}{p_M})) \tag{4.3}$$



(a) F for $acc = 1.0$, $\beta = 0.35$ (b) F for $acc = 1.0$, $\beta = 0.5$ (c) F for $acc = 1.0$, $\beta = 0.85$

Fig. 4.13 Fitness function value across different β configurations.

4.3.3 Monitor - Graphical Interface

The backend algorithm is complemented by a web-based graphical interface. Its primary function is to assist in making crucial design decisions during the development of the overarching algorithm and to validate proof-of-concept solutions for low-dimensional tasks. This visualization tool is invaluable for gaining insights into the functioning of the running program. It is developed using the latest web technologies, primarily built in React.js, and utilizing the *3D Force-Directed Graph* library [6] for dynamic and interactive network illustration. Data is transmitted to the web server using the MQTT protocol, leveraging a public broker to enable asynchronous communication even from a public domain where the webpage is hosted.

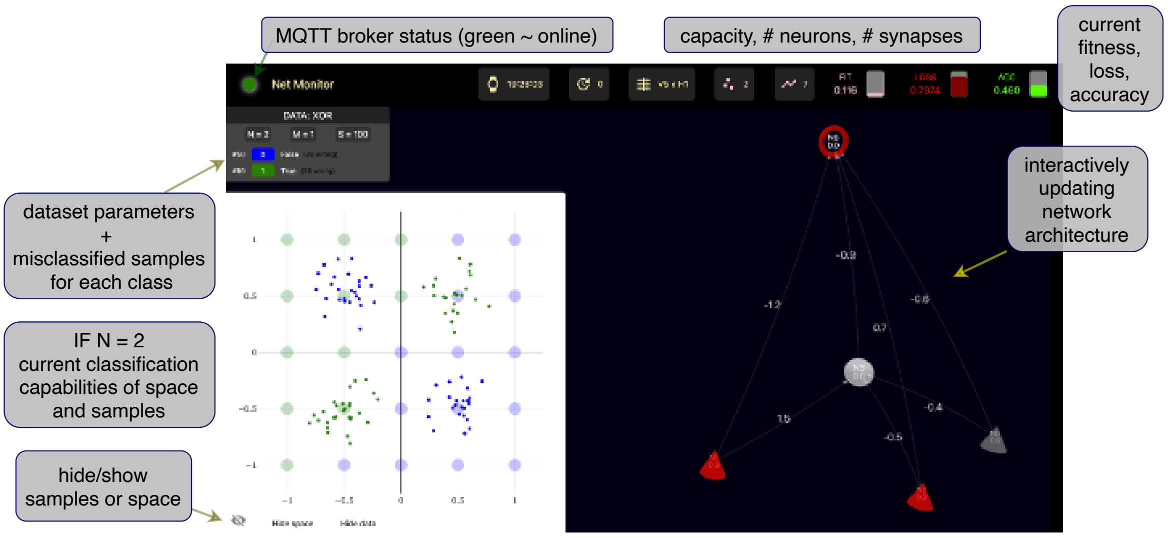

MQTT broker status (green ~ online)
capacity, # neurons, # synapses
current fitness, loss, accuracy
interactively updating network architecture
dataset parameters + misclassified samples for each class
IF N = 2 current classification capabilities of space and samples
hide/show samples or space
Net Monitor
DATA: XOR
N = 2
M = 1
S = 100
Hide space
Hide data

Fig. 4.14 Screenshot of the graphical interface.

The visualization of the current network architecture is presented as a draggable and adjustable 3D object, automatically laid out on the page for optimal clarity. It can display arbitrary scores or weights of the synapses and visually distinguish between input, body, and output neurons. Additionally, it maintains the order of neurons from bottom to top, aligning with the forward data flow. In the case of a two-dimensional problem, a 2D space is provided on the left side of the page. Here, we can observe how the current network classifies the entire 2D space (transparent circles), as well as the actual samples from the dataset. Additionally, the dataset description is displayed, allowing users to extend a pop-up window to view all misclassified samples for each class. The top bar provides information such as the timestamp of network compilation, the current time-step of the algorithm, and statistics such as the number of neurons and synapses in the architecture. Finally, in the top right corner, we can access crucial network capabilities such as fitness, loss, and classification accuracy.

4.3.4 Termination Condition

Several variants of termination conditions were explored during the algorithm's development. Potential strategies include achieving a designated accuracy threshold monitoring the convergence of the fitness function across multiple episodes, which may also involve minimizing p and thus is more suitable. In this version, the algorithm is manually stopped using the *graphical interface* (Sec. 4.3.3), typically when an architecture that meets our requirements is observed. This approach aligns with the *human-in-the-loop principle* (described in Sec. 3.4), where interaction with the algorithm occurs during the process. In this scenario, the human intervention is represented by terminating the algorithm at the right time. At the same time, the *best* entity observed is iteratively updated (see Sec. 4.3.7).

4.3.5 Multi-Agent based Mutation

In this section, we introduce the key component of the main algorithm that allows the entities transform (mutate). Everything described here takes place within the purple *Mutation* box in Fig. 4.11. The mutation process integrates a reinforcement learning algorithm, specifically the *Asynchronous Advantage Actor Critic* (A3C) method, as detailed in Sec. 3.1.7. Importantly, the main outer loop of the *genetic algorithm* corresponds to the episodes of the RL algorithm. Therefore, the GA acts as an intelligent selector of the initial state for the RL algorithm at the beginning of each episode (which is usually chosen randomly in standard RL). The entity on the input to the mutation process is called *mutatee*. Drawing from the approach introduced in Sec. 4.2.3, each entity is considered to be a *multi-agent system*. In this analogy, each agent within the system, representing a single synapse in the network, has two possible actions: 1) *ON* or 2) *OFF*, indicating its current presence within the network architecture. The transformed entity on the output of *mutation* is called *mutant*.

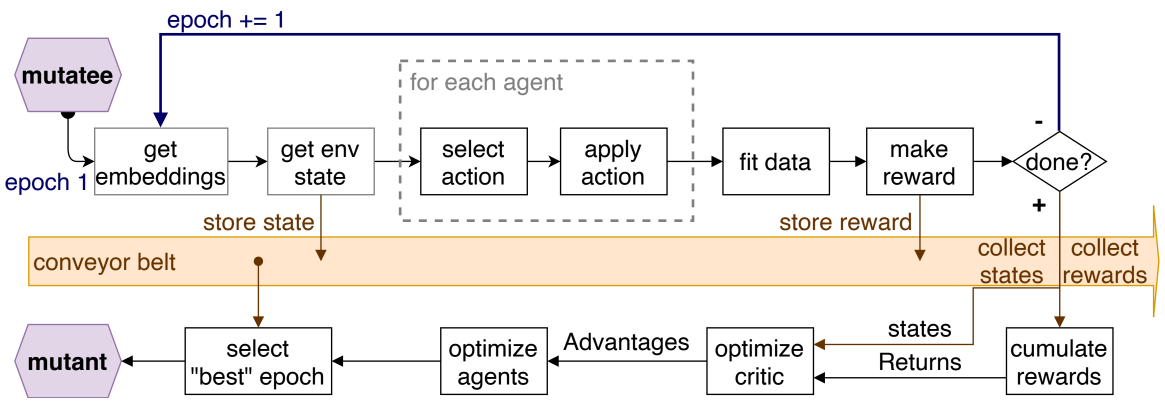


Fig. 4.15 The *mutation* process within a single episode of the GA outer loop.

Figure 4.15 illustrates the overall iterative process, equivalent to a single *trajectory* within the state space, using RL terminology. Each block in the diagram is described in detail in the subsequent paragraphs. Notably, with a *policy-based* RL algorithm such as A3C, the experiences gained from navigating the search space are stored across epochs, as depicted by the *conveyor belt* metaphor in Fig. 4.15. Policy optimization steps of all *actors* (agents) as well as the *critic* are performed at the end of each episode, following the quitting from the inner loop of epochs.

This aspect is not evident in Fig. 4.16; however, this illustration provides a detailed perspective of a single epoch of the RL algorithm. Importantly, it shows how the global state of the environment is computed and utilized by individual agents. Here, we leverage the critical concept introduced in Sec. 4.2.4 of utilizing the *Self-Attention* mechanism. Hence, it is appropriate to update the RL method name. As we incorporate *Attention*, it results in **A4C**.

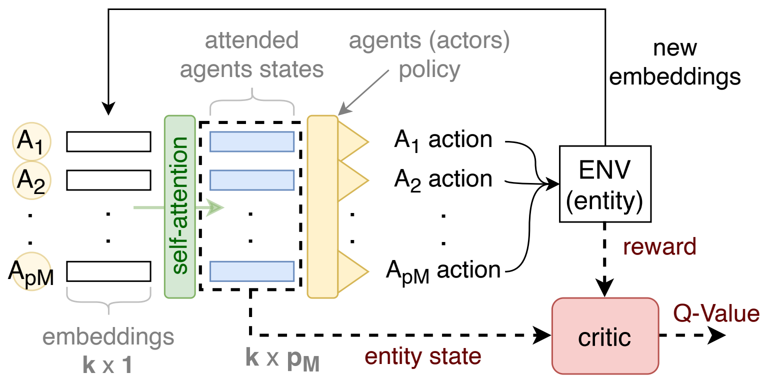


Fig. 4.16 A single epoch of the A4C learning (no policy updates included).

The formation of agent embeddings is discussed below. Each embedding has a shape of $k \times 1$. Consequently, a matrix of these embeddings for all agents has a shape of $k \times p_M$. Let χ'_{A_i} denote the embedding vector of agent A_i. Following the mechanism of *trainable attention* (Sec. 2.4.1), we compute $q_i = W_q \cdot \chi'_{A_i}$, $\kappa_i = W_\kappa \cdot \chi'_{A_i}$, and $v_i = W_v \cdot \chi'_{A_i}$, where W_q, W_κ and W_v are matrices of trainable parameters, each of dimension $k \times k$. Then, the *attended state* χ_{A_i} for agent A_i is computed as given by Eq. 4.4.

$$\chi_{A_i} = \sum_j softmax(\frac{q_i^T \cdot \kappa_j}{\sqrt{k}}) \cdot v_j \tag{4.4}$$

As the first dimension is fixed (dimensionality k) and the second one can vary, it allows for a processing of both global entity state and states of individual agents using the same attention mechanism. Moreover, with just one forward pass, we can obtain attended states for all agents simultaneously, meeting our objective of effectively informing each agent about the situations of others. As the mechanism preserves the input dimensionality in the output, we get a matrix of dimensionality $k \times p_M$ as the input for the *critic* model and a vector of dimensionality $k \times 1$ for the *actor* (agent) model.

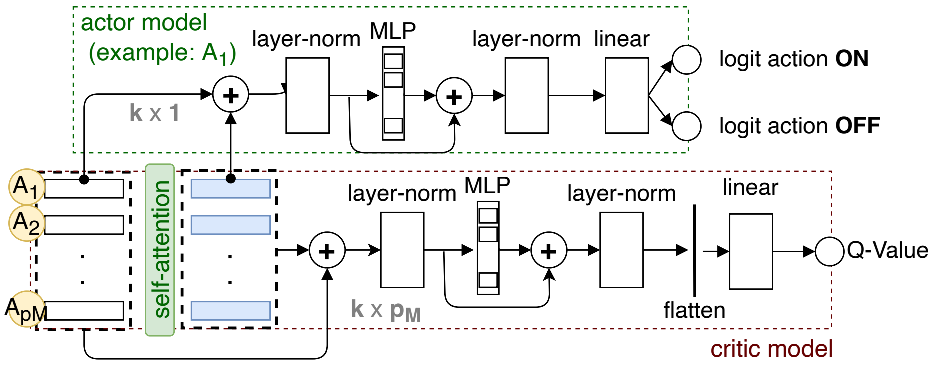


Fig. 4.17 Actor (agent) and critic models structures sharing the attended input.

Illustrated in Fig. 4.17, both models utilize the original agents embeddings using *skip connections*. The subsequent architecture for both models closely resembles the *Transformer* block. However, there is a difference in the output: the *critic* model requires a flattening operation as it deals with a matrix input, producing a single logit value representing the *Q-value*. On the other hand, the *actor* model outputs as many logits as there are defined actions, which in this case are two for actions *ON* and *OFF*. These logits are subsequently passed through the *Softmax* function to produce probabilities for each action. However, it's worth noting that softmax is not employed during model optimization and is therefore omitted from the figure. The *Q-value* on the output of the *critic* model serves as the evaluation metric for the current state of the *entity* (the current design of the network architecture).

Agents and Critic Optimization The models are optimized at the end of each episode, which is triggered by reaching the maximum number of epochs, by default set to $T = 100$. Alternatively, the inner loop can be quitted by meeting a termination condition, depicted by the *done* block in Fig. 4.15. However, this is only feasible for tasks where the optimal (minimal) network architecture is known. Typically, this information is unavailable, leading to the execution of all epochs in every episode.

Given rewards $r^{<t>}$ and state values $V^{<t>}$ for each epoch $t \in \{1, 2, ..., T\}$, we employ the *Generalized Advantage Estimation* (GAE, [117]) to compute the *Advantages*, as defined by Eq. 4.8. Here, done$^{<t>} \in \{0, 1\}$ denotes the termination state flag at epoch t, and $\hat{A}^{<T>} = 0$. The values for each t are calculated in reverse order. The *Advantages* are then normalized. By employing (*GAE*), we can effectively manage reward delays and noisy rewards, thereby improving the estimation of the true value of actions. The default parameter settings are $\gamma = 0.95$ and $\lambda = 0.98$, where γ represents the learning rate and λ determines the balance between short-term and long-term advantages.

$$\delta^{<T>} = r^{<T>} - V^{<T>} \tag{4.5}$$

$$\delta^{<t>} = r^{<t>} + \gamma \cdot V^{<t+1>} \cdot (1 - \text{done}^{<t+1>}) - V^{<t>} \tag{4.6}$$

$$\hat{A}^{<t>} = \delta^{<t>} + \gamma \cdot \lambda \cdot (1 - \text{done}^{<t>}) \cdot \hat{A}^{<t+1>} \tag{4.7}$$

$$A^{<t>} = \frac{\hat{A}^{<t>} - \text{mean}(\hat{A}^{<t>})}{\text{std}(\hat{A}^{<t>})} \tag{4.8}$$

At this point, a very important research question arises regarding the design of the actor (agent) model. Specifically, it pertains to whether to employ:

- An individual policy model (with separate parameters) for each single agent.
- A shared policy model for agents of the same entity.
- A shared policy model for all agents across the algorithm, even across different entities.

The *Actor* model – either a single one in the case of the *shared-policy* mode of learning or separately for each agent – is optimized using Proximal Policy Optimization (PPO, Eq. 3.18 [118]) using a clipping value parameterized by $\varepsilon = 0.2$, the computed *Advantages* $A^{<t>}$ and taken actions $a^{<t>}$ over the epochs. Then we use $\hat{R}^{<t>} = A^{<t>} + V^{<t>}$ as the desired targets for the *Critic* model, which uses the *Mean Squared Error* (MSE) loss function (Eq. 2.39). For both models, we employ the *AdamW* optimizer [82] with an initial learning rate of 0.001 and utilize the *Gradient Clipping* method [133] with *max_norm* $= 0.5$.

The choice of sharing the policy among agents is closely linked to the design of the agent's embedding, including considerations about whether to include the agent's position into the embedding or not. For the *critic*, the same model is utilized for all entities throughout the algorithm episodes to accumulate a maximum amount of experience.

Agent Embedding Here we have another research question, as forming the embedding of a single agent is crucial for the algorithm. Two variants were designed (shown in Fig. 4.18).

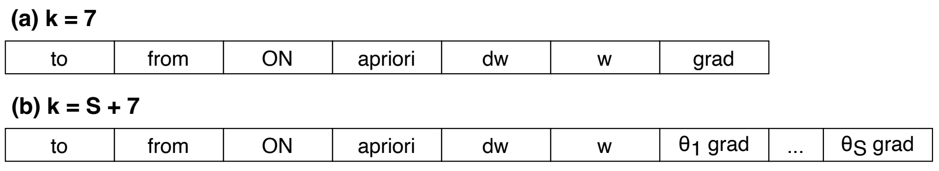


Fig. 4.18 Two proposed variants of the agent embedding (θ_i being training samples).

Both variants are suitable for the shared policy of agents as both include an encoded position of the synapse in the network. The *Grid* assigns a unique number to each neuron in the network, thus, we can encode the source and the target neuron of the synapse. We use a normalized value, obtained by dividing the neuron number by the total number of cells C in the *Grid*. This makes the first two features static. Next, we incorporate information about the current presence of the synapse in the network ($ON \sim 1$, $OFF \sim 0$). Then, we introduce a feature called *apriori*, which provides information about the current situation of the surrounding neurons (dead or active).

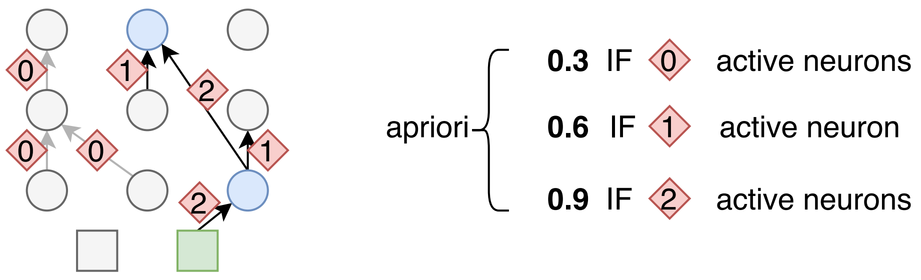


Fig. 4.19 The prior probability of activating an agents is based on its current suroundings.

The next two features are the current weight $w^{<t>}$ of the synapse and the change in weight since the beginning of training, defined as $dw^{<t>} = w^{<t>} - w^0$. For the first variant (a), the last seventh feature is the mean gradient value obtained after fitting all training samples by computing a backward pass using the chain rule of backpropagation. In the case of the second variant (b), this gradient value is computed for every single sample separately, resulting in an embedding vector of length $k = S + 7$, where S is the number of training samples.

Learning of Self-Attention Returning to Fig. 4.17, another research question arises regarding the training of the *attention* mechanism employed at the input to both the agent and critic models. There are three potential variants for training the *attention* parameters:

- Utilizing only the actor model, which learns to predict the *Q-Value* of the $k \times p_M$ state of the entity.
- Employing only the model (or models if their policies are individual) of the agents, learning the actions to take based on a $k \times 1$ agent's state.
- Using a combination of both - agents' updates as well as critic's updates.

Reward The design of the reward function is another critical aspect of the algorithm, as it determines the learning objectives for the multi-agent system. After an extensive experimental exploration of numerous variants, while maintaining the relation to the *Fitness* function of the GA (Eq. 4.3), the formulation presented in Eq. 4.9 was derived. Its negative nature encourages faster algorithm convergence and the high value of 10 favors the discovery of an optimal architecture, if it is known for the task at hand.

$$r = \begin{cases} 10 & \text{if optimal found} \\ (2 - acc) \cdot [\beta \cdot \frac{L}{L_M} + (1-\beta) \cdot \frac{p}{p_M}] & \text{otherwise} \end{cases} \tag{4.9}$$

Selection of the mutant Finally, at the end of the mutation process, once we've exited the inner loop and the models are already optimized, the last step is to generate the transformed network architecture, referred to as the *mutant*. This entity is then considered as a candidate for the population of the next generation of the GA (outer loop). It was determined to select the architecture with the highest *Fitness* value (Eq. 4.3) across all epochs.

4.3.6 New Generation Selection

Finally, in reference to Fig. 4.11, the last decision to make is the selection of G entities (defaulted to $G = 4$) to represent the next generation. As previously mentioned, this selection essentially determines the initial state for the RL algorithm within the *mutation* block in the subsequent episode (typically done randomly in standard RL). In this context, it was determined to leverage structures obtained from previous generations (*mutants*) by selecting $\frac{G}{2}$ best-performing entities based on the fitness function. To determine the best architectures from the available candidates, for each of them, we apply the following procedure:

1. Reset parameters (weights).
2. Retrain a predefined number of epochs on the training set (by default 100 epochs).
3. Evaluate the performance on validation data with the *Fitness* function (Eq. 4.3).

The generation is then completed by adding $\frac{G}{2}$ initial structures, as defined in Sec. 4.3.1, aiming to avoid diverging from unexplored architectures by following suboptimal trajectories.

4.3.7 Collecting Results

The best entity encountered during the loop is being updated. We differentiate between the best entity of the current run and the best ever found for the problem. As the *Fitness* function

(Eq. 4.3) depends on network capacity p_M, which can vary, it's only relevant for the current run. To retain the best-ever network architecture, we compare a new candidate entity (A) with the current best entity (B) as depicted in Fig. 4.20. This follows the objective to prioritize the accuracy over minimizing the number of parameters. Additionally, as a byproduct, we obtain trained *Actor* and *Critic* models, which can be utilized as demonstrated in Chap. 5.

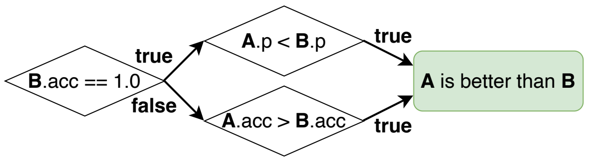


Fig. 4.20 Determining if entity A is better than entity B (different runs).

4.4 Potential Extensions

The proposed method is complex, incorporating various machine learning principles and raising several research questions for evaluation. Additionally, here are a few ideas not implemented in this work, which deserve consideration and further exploration.

- *Parallelization:* The methodology is currently implemented in Python with minimal code optimization. However, there is potential for enhancement which could expand the capacity of the *Grid*. Parallelization can be applied at two levels: 1) at the entity level (outer loop - genetic algorithm); and 2) at the agent level (inner loop - RL).
- *Crossover operation:* Although not currently implemented, the algorithm framework is designed to accommodate crossover, potentially leveraging the *cross-attention* mechanism as discussed in Sec. 4.2.4.
- *Generation selection:* This idea suggests integrating the ε-greedy strategy (presented in Fig. 3.12), for selecting populations in the GA (Sec. 4.3.6). Initially favoring random initial structures for exploration, the strategy would gradually shift towards exploiting learned experience by prioritizing generated *mutants*.
- *Kalman filter:* Applying this well-known state estimation tool to enhance the state of the agents presents a promising research avenue. Using an analytical solution could yield significant improvements in the algorithm's *explainability*.
- *Human-in-the-loop.* This concept is introduced in Sec. 3.4 and the idea is to integrate it into the *graphical interface* (Sec. 4.3.3) for asynchronous interaction with the algorithm during processing. For example, we could interactively prioritize selected samples or classes and make the network more *adaptive* to new conditions.

Chapter 5

Experimental Evaluation

This section evaluates the presented methodology. We begin by describing the experimental setup and introducing selected evaluation metrics. Subsequently, we introduce multiple example datasets and employ them to evaluate the method. Finally, the obtained results are summarized and the observations are listed.

5.1 Setup and Metrics

To prevent overfitting and assess the generalization capabilities of generated networks, we employ dataset splitting into four subssets, in contrast to the standard approach of three sets typically used in machine learning tasks. The type of itemization bullet corresponds to those used in the figures below.

- ● *Training set*: The `train` subset is exclusively used for training the neural networks. It serves to fit the specified number of epochs during the RL algorithm and for retraining the network before evaluation in the *Selection* process.
- ■ *Development set*: The `dev` subset is utilized for evaluating entities during the *mutation* process (RL algorithm) when computing the *Fitness* function and reward.
- ⧫ *Validation set*: The `val` subset is used to evaluate entities in the outer loop (GA algorithm) when computing the *Fitness* function for selecting the best entity of the current generation.
- ★ *Testing set*: The `test` subset is never used in the training process of the algorithm. These data samples are reserved for overall result evaluation.

As previously discussed in Sec. 4.3.7, there are two main outcomes of the algorithm:

- *best-ever entity*: This is the main outcome of the algorithm. For a fixed classification problem, the algorithm can be run several times with various configurations (see App. B.3). The *best-ever* entity represents the neural network architecture, including trained parameters, that performed the best on the testing set over all runs. The quality of the entity is described by the following metrics:
  - Classification accuracy on the testing subset.
  - Loss on the testing subset. The loss functions used are:
    1. *Binary Crossentropy* (Eq. 2.42) for tasks with $M = 1$.
    2. *Categorical Crossentropy* (Eq. 2.43) for tasks with $M > 1$.
  - Number of parameters corresponding to the number of active synapses in a structure defined by the *Grid* (introduced in Fig. 4.5). Additionally, we can observe the number of dead neurons in the architecture. The presence of dead neurons indicates that some synapses may be redundant.
- *trained agents*: As a side result of the training process, we obtain two trained models:
  1. *Critic* model: Capable of evaluating the current state of the entity and thus can be used as a metric provider.
  2. *Actor* model (or a set of models in the case of individual policies of agents): Capable of making decisions for each single synapse about being or not being in the network given the current entity state. Hence, by only applying these actions in a loop, we can potentially transform any state of the entity (any initialization of it) into the efficient final form.

During the episodes of the running algorithm, several measures can be observed, providing insights into various aspects of the algorithm's performance and behaviour during training.

- *Critic Loss*: The critic model is also trained in parallel during the RL algorithm.
- *Sum of Returns*: As the mutation models are trained, the cumulative rewards should increase, and thus the sum of *Returns* over epochs should also increase with episodes. We can observe them individually for each entity or take the mean over them.
- *Fitness Function*: Taking the value of the best entity in each episode, a plot over episodes can be formed.
- *Number of Epochs till Done*: This is only applicable for tasks where the minimal (target) architecture is known. This number should decrease with time.

5.1.1 Default Configuration

The most critical configuration settings of the algorithm are outlined in Tab. 5.1, while the complete configuration file is available in App. B.3. This configuration, derived as the optimal choice, is utilized in all experiments unless specified otherwise.

Outer loop (GA)		Inner loop (RL) - A4C Mutation	
Population size (G)	4	Agents policy	shared
Init method	(a) zero (Fig. 4.12)	Agent state	(a) k=7 (Fig. 4.18)
Fitness β	0.6	RL timesteps (T)	50
Fitness retraining	100 epochs	Retrain epochs (Z)	5
Stop method	HITL (GUI)	Attention updates	agents only
Mutation method	A4C	Agents shuffling	true
Crossover method	not used	Agents synchronization	false

Table 5.1 Default configuration settings used in the experiments.

Importantly, the shared agents policy proved to be effective, resulting in a significant improvement in processing time as only one model is trained for all agents—this was crucial. Next, the shorter version of the agents' state ($k = 7$) was utilized, while the alternative option (Fig. 4.18b) is left for future research. Parameter Z represents the number of epochs that are fitted to the network being generated at each timestep of the RL algorithm after taking agents' actions. It helps the agents see the influence of their actions and allows the entity to learn while traversing its trajectory. The research question addressed in Fig. 4.16 pertained to how the parameters of the shared *Attention* are updated. It proved beneficial to utilize only agents' updates, which makes sense as agents are the ones who creates the state. Finally, the order of agents taking their actions is randomized each epoch of the RL algorithm before actions are taken, and their states, on which these actions are based, are not synchronized. This means that each agent makes its decision based on the output of the *Attention* mechanism updated by the action of the previous agent. Consequently, the agents are aware of each other's behaviour without the need for network retraining after taking each single action.

5.2 Examples

The datasets used for evaluation range from low-dimensional classification problems, valuable for method development and design choices, to multi-dimensional tasks. These tasks, each showcasing different features of the algorithm, demonstrate the scalability and applicability of the approach across various domains, for example in NLP, as demonstrated in Sec. 5.2.4.

5.2.1 Fundamental 2D Problems

We begin with two elementary 2D tasks, both involving classification into two classes. The *Basic-2D* task, as defined in Tab. 5.2, is linearly separable by a single line in the 2D space. In contrast, the second 2D task, the XOR problem defined by Tab. 5.3, cannot be separated by a single line in the 2D space and hence requires a more sophisticated network architecture.

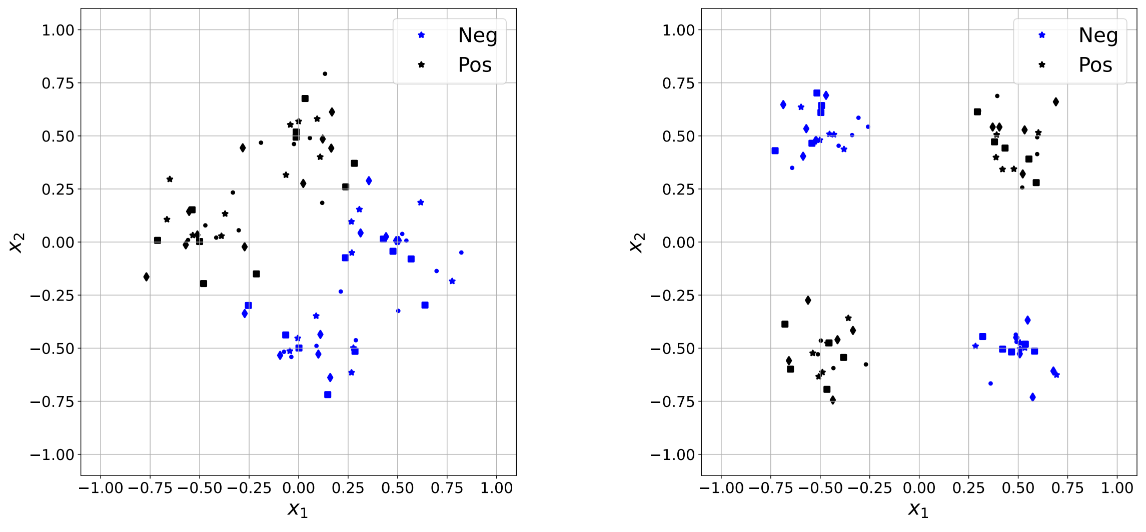
x_1	x_2	y
0	1	1
-0.5	0	1
0.5	0	0
0	-1	0

Table 5.2 Basic 2D problem.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.3 XOR problem.

Working with these tasks offers the advantage of being easily visualized in the 2D space. Both problems are depicted in Fig. 5.1, where the four subsets described in Sec. 5.1 are indicated by the shapes of their markers. The data is generated using the problem definition in Tab. 5.2, complemented by additional mechanisms to supplement variability in the dataset.



(a) Basic-2D dataset.

(b) XOR dataset.

Fig. 5.1 Fundamental 2D tasks of two classes - illustration in the 2D space.

Importantly, the minimal required network architecture (and the minimal number of parameters p) is known for both of these problems, as illustrated in Fig. 5.2. For the XOR problem, two network architectures are known (Fig. 5.2 - $b1$ and $b2$), both considered minimal. However, $b1$ is preferred due to its lower number of synapses ($p = 7$ with biases).

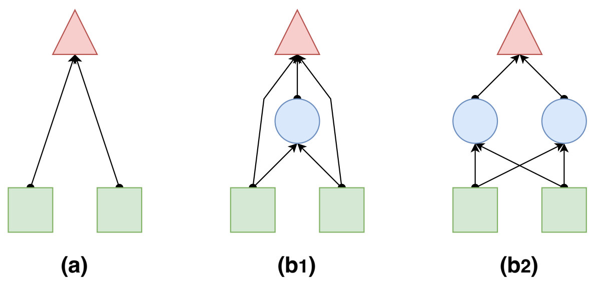
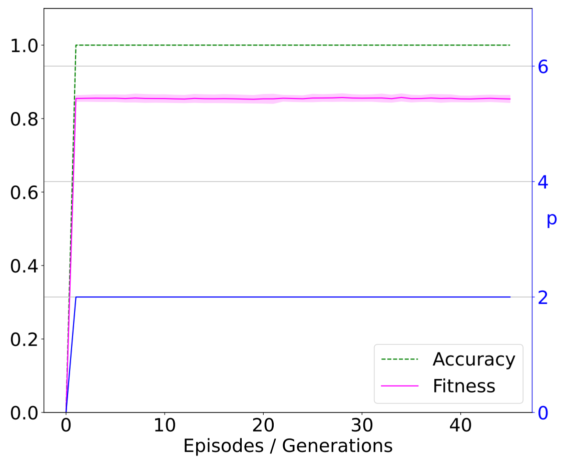
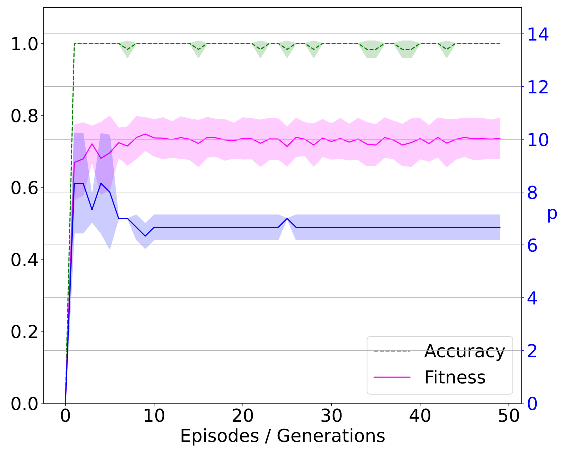


Fig. 5.2 Minimal structures, 2D problems: a) Basic-2D; b1) XOR ($p = 7$); b2) XOR ($p = 9$).

The performance of the algorithm on these fundamental 2D tasks is depicted in Fig. 5.3. Since the algorithm is fully stochastic and dependent on several randomized initializations, the experiment was run 10 times, and the figures provide the mean and standard deviation values across these realizations. Each of these figures shows the statistics of the best entity available at the given episode. The left y-axis represents the accuracy and the value of the entity's fitness (see Eq. 4.3). The right y-axis indicates the number of parameters p.



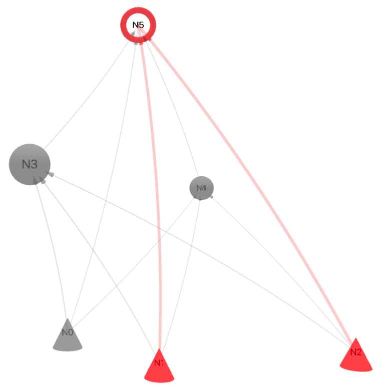
(a) Performance on the Basic-2D task.



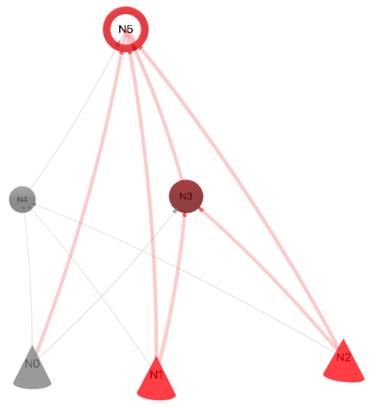
(b) Performance on the XOR task

Fig. 5.3 Algorithm performance on 2D tasks - mean and standard dev. of 10 realizations.

As the optimal architecture is known for these fundamental tasks, we can state that it was evidently found in all realizations for both cases. For the *Basic-2D* problem, the algorithm quickly determined that only $p = 2$ parameters are needed, even though it worked with a *Grid* of capacity $p_M = 11$ - two hidden neurons were successfully left out. In the case of the XOR problem, the resulting number of parameters across realizations oscillated between $p = 6$ and $p = 7$, indicating that with a good initialization, one of the biases is actually also not needed. The *Grid* capacity for the XOR problem was tested with two ($p_M = 11$) and three ($p_M = 19$) hidden neurons. The resulting network architectures are illustrated using the developed web-based *Monitor* (see Sec. 4.3.3) in Fig. 5.4.



(a) Resulitng Basic-2D network ($p = 2$).

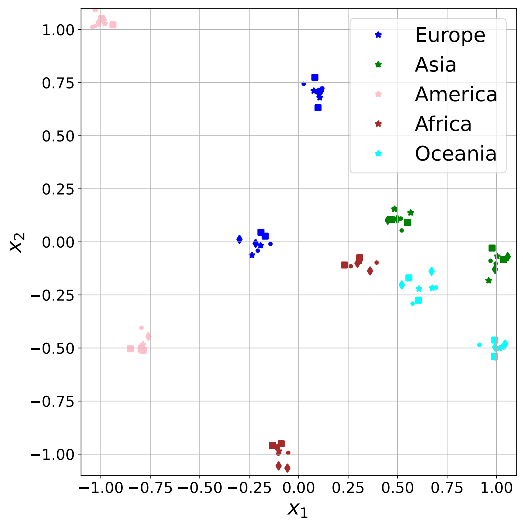


(b) Resulting XOR network ($p = 6$).

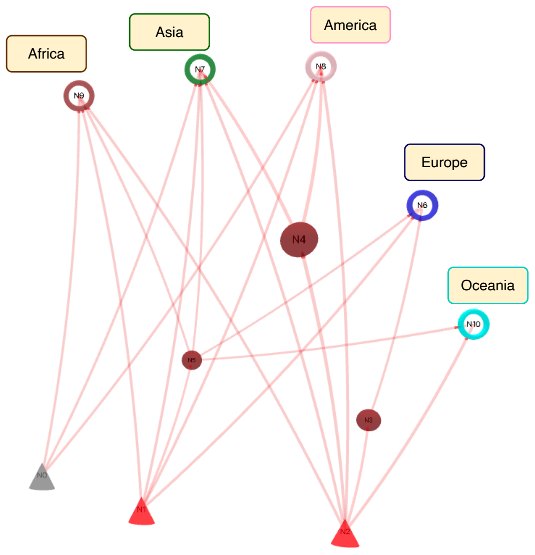
Fig. 5.4 Generated networks - fundamental 2D tasks (corresponding with targets - Fig. 5.2).

5.2.2 A Multi-class 2D Problem - Continents

In the second example, we remain in the 2D space, but the classification task becomes more challenging as we distinguish between five classes. Therefore, $M = 5$, and the *Categorical Crossentropy* loss function is used (Eq. 2.43). The data are inspired by the layout of the continents (Fig. 5.5a). The generated network of $p = 21$, achieving 100% accuracy, is interesting in terms of *explainability*. For example, we can observe that neuron N4 works for classes *Asia* and *America* based solely on the x_2 feature.



(a) Multi-class 2D dataset - *Continents*.



(b) Best architecture for the multi-class 2D task.

Fig. 5.5 The multi-class 2D task and a partially explainable network architecture solving it.

5.2.3 A Multidimensional Problem - Michalski's Trains

Next, we transition back to a classification task with two classes ($M = 1$), but this time the problem is multidimensional ($N = 7$). The *Michalski's trains* problem was originally introduced in [73], further elaborated in [92], and recently used in [20] to evaluate a network pruning algorithm. The task involves determining concise decision rules to distinguish between two sets of trains (Eastbound and Westbound). A simplified version from [92] is illustrated in Fig. 5.6. Each train is described by 7 binary features listed in Tab. 5.4.

	feature	*encoded as **0***	*encoded as **1***
1	car length	long	short
2	car type	open	closed
3	cabin pattern	vertical lines	horizontal lines
4	load shape	triangle	circle
5	color of trailer wheels	white	black
6	color of first car wheel	white	black
7	color of second car wheel	white	black

Table 5.4 Features describing the trains (table overtaken from [20]).

With the information in Tab. 5.4, we can encode the trains shown in Fig. 5.6 into feature vectors as shown in Tab. 5.5. The task is then to determine the minimal number of input features needed for the east-west classification based on the six possible types of trains. In this task, we obviously cannot split the dataset into subsets as usual.

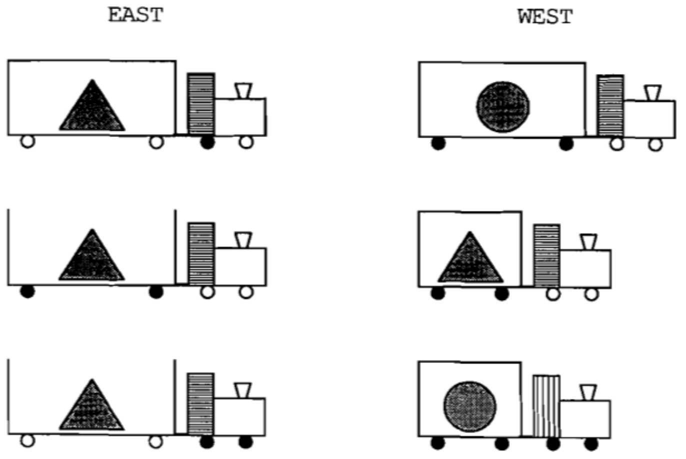


Fig. 5.6 The Michalski's Trains task [92].

class EAST	
east 1	$[0,1,1,0,0,0,1]$
east 2	$[0,0,1,0,1,0,0]$
east 3	$[0,0,1,0,0,1,1]$
class WEST	
west 1	$[0,1,1,1,1,0,0]$
west 2	$[1,1,1,0,1,0,0]$
west 3	$[1,1,0,1,1,1,1]$

Table 5.5 Trains: encoded features.

Looking at Fig. 5.6, the optimal solution for this problem is to retain the *car length* and *load shape* features (1 and 4), as they are sufficient to distinguish *west* trains from *east* trains. This provides us with information about the optimal target network architecture - it should have only one output neuron ($M = 1$) and three parameters (two synapses and the bias). This is also supported by [92], where one hidden neuron is used to learn this problem.

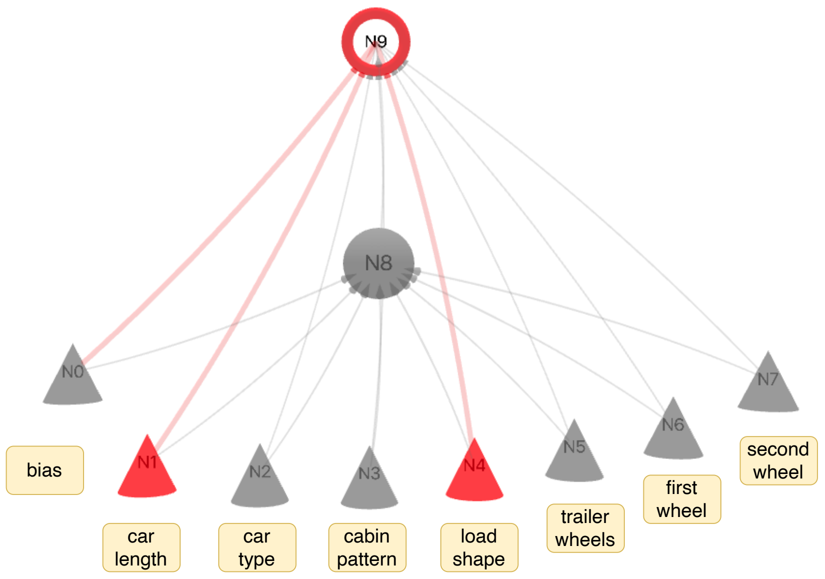
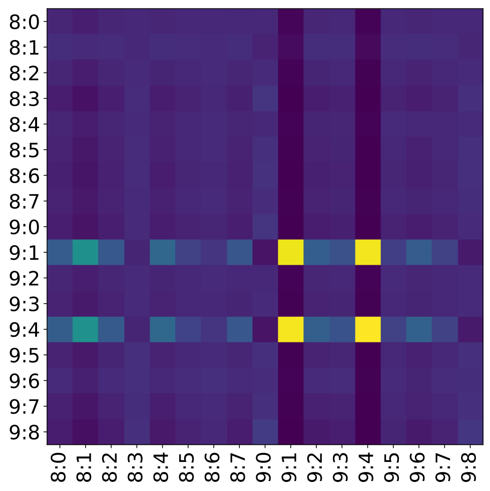
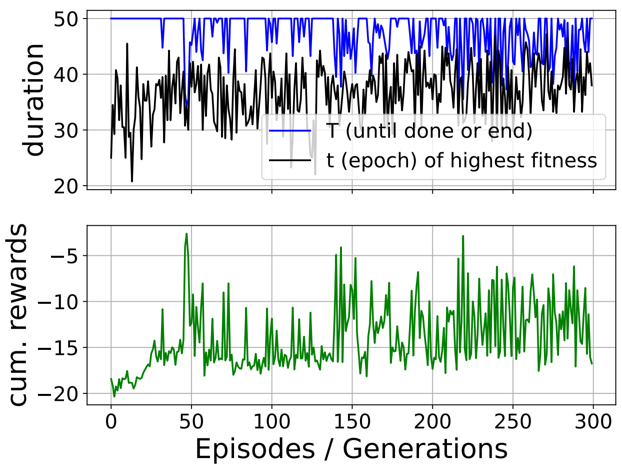


Fig. 5.7 Resulting generated network architecture for the Michalski's trains problem.

Therefore, we conducted experiments with a tiny *Grid* consisting of a single hidden neuron only, resulting in a capacity of $p_M = 19$ (for $N = 7$). Fig. 5.7 shows that the algorithm successfully identified the two needed features and resulted in the expected architecture.



(a) Trains: attention scores.



(b) Trains: episode durations and cum. rewards.

Fig. 5.8 The Michalski's trains problem: trained attention scores and best entity statistics.

Furthermore, as depicted in Fig. 5.8a, the significance of the two selected features is supported by the trained attention scores of the best entity. Here, the axis tick labels are encoded as `neuron-to:neuron-from`, and therefore, with 9 being the number of the output neuron, the positions `9:1` and `9:4` fit.

In Fig. 5.8b, we can see statistics of a single selected run of the algorithm over GA generations (RL episodes). The bottom figure shows the cumulative rewards gained. We can notice that the peaks perfectly align with the epochs where the optimal solution was found and hence the agents were awarded by $r = 10$ (see Eq. 4.9). The top figure shows, for each episode, the timestep (epoch of the RL algorithm) when the highest fitness was found (black) and the length of the episode (blue). Here we have a maximum of $T = 50$ epochs, unless the optimal solution is found and the inner loop is exited earlier.

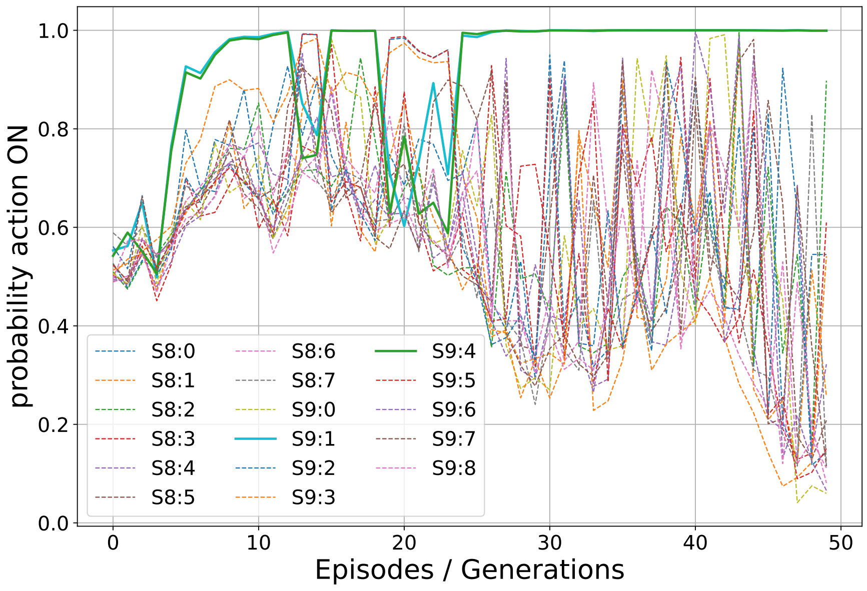


Fig. 5.9 Probability of agents taking action *ON* (activating themselves).

Finally, Fig. 5.9 illustrates the learning process of the agents' shared policy represented by a model described in Fig. 4.17. Although the model is shared by all agents, each agent has its own state derived using the *Attention* mechanism, resulting in different model outputs for different agents. In the figure, we can observe the probability of taking the *ON* action (activating self in the network) for every single agent over the episodes. The input uses the state of the currently best-performing entity (network) for each episode.

Interestingly, we observe that the two important synapses (`9:1` and `9:4`, highlighted in Fig. 5.9) are closely related to each other. They switch to a probability of being active equal to one around the 25th episode and then never leave this status. In contrast, the other agents tend to be deactivated as the *actor* and *critic* models learn over the episodes.

5.2.4 Intent Classification

In this section, we explore an example that is multidimensional ($N = 16$) and multi-class ($M = 4$) for evaluating the algorithm. The task involves intent classification in the domain of *Natural Language Processing* (NLP). The dataset, manually created for demonstration purposes, is very small (eight samples per class, two in each subset). The samples are listed in Tab. 5.6, sorted into their classes, and also subsets indicated by the shapes on the left, that correspond to the settings in Sec. 5.1.

	NEGATIVE	**POSITIVE**	**HELLO**	**QUESTION**
•	'I hate you.'	'You are beautiful.'	'Hello.'	'What are you doing?'
•	'Your dog is so ugly.'	'The weather is nice today.'	'Hi.'	'Who are you?'
■	'I am depressed.'	'I like you.'	'Good afternoon.'	'What do you want?'
■	'I have so bad day today.'	'You are the best.'	'Good evening'	'What is the time?'
⧫	'It is a rubbish!'	'It is gonna be OK.'	'Hey.'	'Where do you live?'
⧫	'You are stupid!'	'Sounds good'	'Good morning.'	'What is your name?'
★	'I do not like it.'	'You are perfect to me.'	'Aloha!'	'What music do you like?'
★	'This color is the worst.'	'This is my favourite one.'	'Hi there!'	'Are you OK?'

Table 5.6 Samples of the intent classification dataset sorted into classes and subsets.

To transform the textual inputs into encoded samples, two external tools are employed.

1. *Sentence Transformer library* [109]: As depicted in Fig. 5.10, a pre-trained BERT model transforms a textual input of arbitrary length into an embedding vector of a fixed dimension. In this case, the model `all-MiniLM-L6-v2` was used, resulting in an embedding dimension of 348. The embedding encodes also the semantics of the input.
2. *Principal Component Analysis (PCA)* [40]: Since the algorithm is not optimized for high-dimensional data yet, the dimension is reduced using the well-known method of PCA. Empirically derived, the dimension of the samples is reduced to $N = 16$, resulting in an explained variance ratio of 0.79 (see Fig. 5.11a). This has proven to be sufficient for our purposes, but it leaves room for potential future experiments.

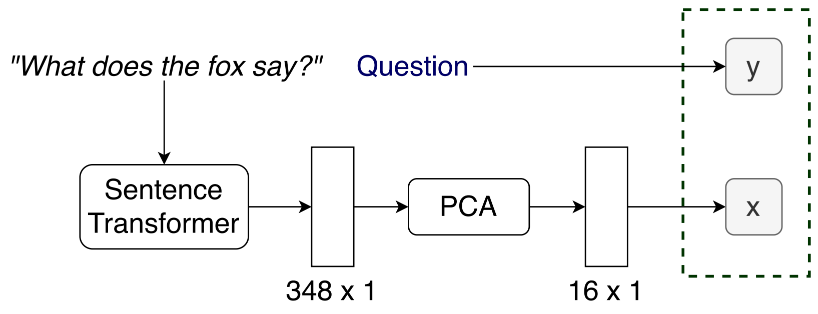
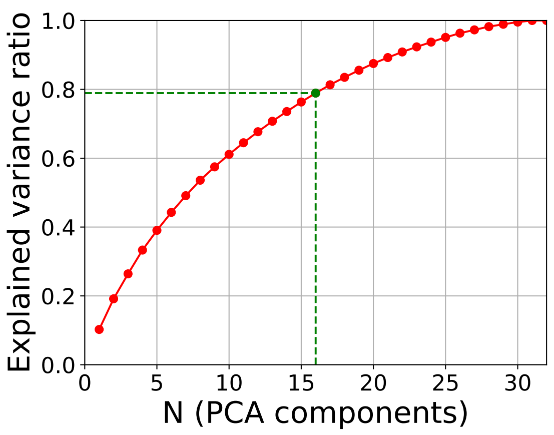
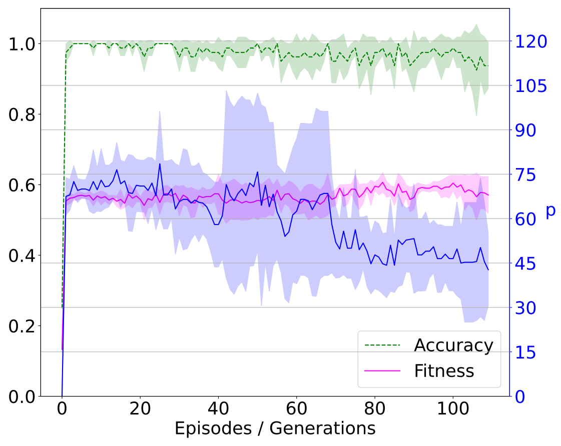


Fig. 5.10 Creation of the intent classification dataset.



(a) Intents: PCA explained variance vs. N.



(b) Performance on the Intents task

Fig. 5.11 Intent classification: Performance of the algorithm (mean of 10 realizations).

In contrast to Tab. 5.1, in this task, experiments were performed with the key parameter $\beta \in \{0.6, 0.7\}$, proving to yield better performance. Figure 5.11b presents the algorithm evaluation, following the same setup as used for the 2D tasks. The left y-axis corresponds to the *accuracy* and *fitness* values, while the right y-axis displays the number of parameters. The figure depicts the mean and standard deviation values of the best available entity for each episode (generation) of the algorithm following the criterion presented in Sec. 4.3.7.

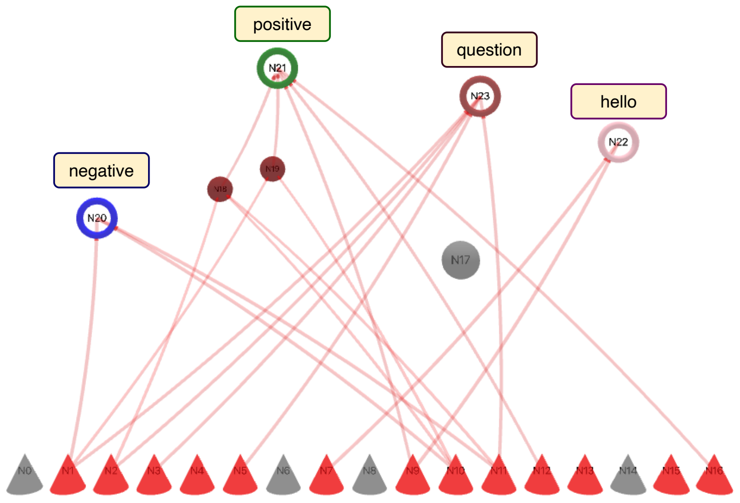


Fig. 5.12 Resulting generated network architecture for the intent classification task.

In this scenario, the *Grid* comprises three hidden neurons, resulting in an available capacity of $p_M = 131$. The overall best architecture found (Fig. 5.12) consists of $p = 25$ synapses while maintaining the validation accuracy of 100%. Additionally, it leads to insights about the architecture; for example, we can see that one of the hidden neurons was completely left out or that the *Hello* class only requires two features.

5.3 Summary of Results

The algorithm underwent evaluation using five classification problems. Initially, it was demonstrated on two fundamental 2D tasks: a linearly separable *Basic-2D* dataset requiring a single neuron for resolution, and the linearly inseparable XOR problem, necessitating an additional neuron. In both cases, the algorithm produced optimal network architectures. Subsequently, its scalability was demonstrated on a multi-class problem (Sec. 5.2.2), followed by application to a multidimensional (but binary) classification problem of Michalski's trains (Sec. 5.2.3). Finally, the algorithm was applied to a multidimensional and also multi-class task of intent classification in Sec. 5.2.4. The results are summarized in Tab. 5.7.

			Basic-2D	XOR	Multi-class	Trains	Intents
V x H			1 x 1	3 x 1	3 x 1	1 x 1	3 x 1
pM			11	19	29	19	131
potential variants			2048	524K	537M	524K	$2.7 \cdot 10^{39}$
best generated entity	p		2	6	21	3	25
	dev	acc	1.00	1.00	0.90	1.00	0.88
		loss	0.09	0.05	1.13	0.02	1.11
		fitness	0.86	0.83	0.40	0.92	0.49
	val	acc	1.00	1.00	1.00	1.00	1.00
		loss	0.05	0.06	1.11	0.02	0.94
		fitness	0.87	0.82	0.45	0.92	0.62
	test	acc	1.00	1.00	1.00	1.00	0.75
		loss	0.07	0.04	1.12	0.02	1.29
		fitness	0.86	0.83	0.44	0.92	0.37

Table 5.7 Results summary for all presented examples.

The most significant observations gathered during the algorithm evaluation are as follows.

- The algorithm demonstrates scalability to multidimensional and multi-class problems. Notably, even for extending the *Grid* capacity beyond a hundred parameters, as demonstrated in Section 5.2.4, the algorithm succeeded and produced an architecture.
- The integration of the *Attention* mechanism as a communication tool for agents in the multi-agent system appears promising (see Fig. 5.8a).
- Generated networks show potential of partial explainability (Figs. 5.5b and 5.12). As all components are considered essential, signals from features to classes can be tracked.

Chapter 6

Discussion

This work is devoted to the challenging task of crafting neural networks architectures from scratch and tailoring them for arbitrary classification problems. Unlike traditional neural architecture search methods, which typically involve tuning meta-parameters through Bayesian optimization, reinforcement learning, or evolutionary algorithms, here the state space is directly formed by different network architecture variants, particularly their components. Solving this task with brute force methods, such as searching the state space, yields as many as 2^{p_M} potential variants, where p_M represents the total number of available parameters to be decided for inclusion in the network. (e.g. for $p_M = 131 \quad \sim \quad 2.7 \cdot 10^{39}$ variants).

The proposed method was tested across five diverse classification problems, consistently achieving 100% validation accuracy while using only a fraction of the available parameters. In the *Intent classification* scenario (Sec. 5.2.4), for instance, the algorithm identified a minimal network configuration with just $p = 25$ synapses out of $p_M = 131$ potential parameters. Despite room for improvement (fails on test set, Tab. 5.7), the algorithm's capability to design compact yet effective networks on given data is rare in the field of machine learning.

Revisiting Hypotheses

- *Hypothesis H1*, originally confirmed in [20] through network pruning, finds further support from a different perspective in this study. Discovering a functional network architecture while utilizing only a fraction of available parameters suggests the presence of redundant components within fully-connected structures.

- *Hypothesis H2*, regarding the partial explainability of sparse neural network architectures, warrants further investigation. Initial indications supporting this concept are observed in Figs. 5.5b, 5.7, and 5.12, where the roles of individual neurons in relation to class and input features can be vizualized.

Regarding the objectives of the thesis outlined in Sec. 1.1, all can be considered achieved. The methodology was proposed, and the algorithm was practically implemented (as detailed in App. B). However, further enhancements to the implementation would be beneficial, as the proof-of-concept version developed in Python hampers processing speed, limits the exploration of numerous arising research questions, and disables the use of *Grid*s with high capacities of parameters.

6.1 Future Research Directions

As highlighted repeatedly in this thesis, this work presents a multitude of research avenues that can be further explored. The potential extensions outlined in Section 4.4 has already offered some of the ideas. Let's complement these extensions for a comprehensive overview.

- *Agent Definition*: The overall methodology of the proposed algorithm is based on several analogies, where one of them is considering a single synapse (parameter) to be the *agent*. However, it could be expanded by defining agents as neurons or entire networks, offering alternative perspectives.
- *Crossover of Entities*: Introducing crossover operations in the outer loop of the genetic algorithm, possibly leveraging the cross-attention mechanism (see Sec. 2.4.1).
- *Training the Attention*: Further experiments and refinement of training strategies are needed in this area. The mechanism can be updated by the *Critic*, the *Actor* or both.
- *Agent's Embedding*: The second strategy outlined in Fig. 4.18b remains unexplored.
- *Agent's Synchronization*: A detailed investigation into the order of agents' actions is warranted. The agents can be activated in *parallel* or *sequentially* (shuffled or not).
- *Transfer Learning*: Utilizing pre-trained *Critic* and *Actor* models from one task/domain as a starting point for learning in is another idea to be tested.
- *Configuration Tweaking*: The configuration presented in Tab. 5.1 is the best that was derived in this work, however, the number of potential combinations for the configuration of the framework is enormous and thus there is always room for further exploration.
- *Signal Tracking*: Having the generated network architecture with only synapses that are needed for solving the problem in hand (are not redundant), allows us to track the signals from features to classes and back. This could enhance the adaptivity and targeted modifications in trained networks.

Chapter 7

Conclusion

This study offers an alternative perspective on neural networks and their principles, diverging from the prevailing trend dominated by large language models in AI research. In contrast to traditional approaches that prioritize scaling up model sizes, this work focuses on designing very compact network architectures and tries to find the meaning of their individual components.

The proposed methodology is novel and never seen or published before. It involves several intriguing concepts, such as identifying analogies among various machine learning strategies or employing the attention mechanism as a cohesive element for individual agents of the multi-agent system. The state of the work is definitely not final. Presented proof-of-concept experiments have shown promising results, indicating the potential of the proposed methodology, however, many more ideas remain unexplored. An advanced version of such an algorithm on a larger scale holds the potential to generate tailored network architectures for any given machine learning problem, while providing deep insights into the inner workings of the network. This may enable targeted modifications to trained network architectures.

As this study reaches its conclusion, we find resonance in the introductory quote. Our research curiosity has driven us to uncover answers to specific questions, yet simultaneously, it has prompted the emergence of new questions for further exploration.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., and et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[2] Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147–169.

[3] AI, M. (2023). Seamless: Multilingual expressive and streaming speech translation.

[4] Amidi, A. and Amidi, S. (2018). Cs 230 - deep learning cheatsheets. [Online; accessed 19-February-2021].

[5] Ao, J., Wang, R., Zhou, L., Wang, C., Ren, S., Wu, Y., Liu, S., Ko, T., Li, Q., Zhang, Y., Wei, Z., Qian, Y., Li, J., and Wei, F. (2022). Speecht5: Unified-modal encoder-decoder pre-training for spoken language processing.

[6] Asturiano, V. (2019). 3d force-directed graph — vasturiano.github.io. https://vasturiano.github.io/3d-force-graph/. [Accessed 15-02-2024].

[7] Baevski, A., Zhou, H., Mohamed, A., and Auli, M. (2020). wav2vec 2.0: A framework for self-supervised learning of speech representations. *CoRR*, abs/2006.11477.

[8] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. cite arxiv:1409.0473Comment: Accepted at ICLR 2015 as oral presentation.

[9] Baker, B., Gupta, O., Naik, N., and Raskar, R. (2017). Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations*.

[10] Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2006). Greedy layer-wise training of deep networks. In Schölkopf, B., Platt, J. C., and Hofmann, T., editors, *NIPS*, pages 153–160. MIT Press.

[11] Benistant, L., Huberman, B., Bonner, A., Gunn, E., and Kindig, C. (2016). Towards data science - a medium publication sharing concepts, ideas, and codes. [Online; accessed 10-February-2021].

[12] Bergstra, J., Yamins, D., and Cox, D. D. (2012). Making a science of model search. *CoRR*, abs/1209.5111.

[13] Bettini, M., Kortvelesy, R., Blumenkamp, J., and Prorok, A. (2022). Vmas: A vectorized multi-agent simulator for collective robot learning. *The 16th International Symposium on Distributed Autonomous Robotic Systems*.

[14] Bettini, M., Prorok, A., and Moens, V. (2023). Benchmarl: Benchmarking multi-agent reinforcement learning. *arXiv preprint arXiv:2312.01472*.

[15] Bloem, P. (2019). Transformers from scratch — peterbloem.nl. https://peterbloem.nl/blog/transformers. [Accessed 01-02-2024].

[16] Bourlard, H. and Kamp, Y. (1987). Auto-association by multilayer perceptrons and singular value decomposition. Manuscript M217, Philips Research Laboratory, Brussels, Belgium.

[17] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

[18] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[19] Bromley, J., Bentz, J. W., Bottou, L., Guyon, I., LeCun, Y., Moore, C., Säckinger, E., and Shah, R. (1993). Signature verification using a "siamese" time delay neural network. *IJPRAI*, 7(4):669–688.

[20] Bulín, M. (2017). Optimization of neural network. Master's thesis, University of West Bohemia, Univerzitní 2732/8, 301 00 Pilsen, Czech Republic.

[21] Castro, P. S., Moitra, S., Gelada, C., Kumar, S., and Bellemare, M. G. (2018). Dopamine: A Research Framework for Deep Reinforcement Learning.

[22] Chaudhari, S., Polatkan, G., Ramanath, R., and Mithal, V. (2019). An attentive survey of attention models. *CoRR*, abs/1904.02874.

[23] Chollet, F. et al. (2015). Keras. https://keras.io.

[24] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555.

[25] Cortes, C. and Vapnik, V. (1995). Support vector networks. *Machine Learning*, 20:273–297.

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

[27] Dayan, P., Hinton, G. E., Neal, R. N., and Zemel, R. S. (1995). The Helmholtz machine. *Neural Computation*, 7:889–904.

[28] DeepAI (2019). Softmax function definition. https://deepai.org/machine-learning-glossary-and-terms/softmax-layer. (Accessed on 06/13/2023).

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

[30] Dent Neurologic Institute (2021). 22 facts about the brain | world brain day. [Online; accessed 19-February-2021].

[31] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. cite arxiv:1810.04805Comment: 13 pages.

[32] Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyper-parameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 3460–3468. AAAI Press.

[33] Dorri, A., Kanhere, S. S., and Jurdak, R. (2018). Multi-agent systems: A survey. *IEEE Access*, 6:28573–28593.

[34] Doshi, K. (2020a). Reinforcement learning made simple. https://towardsdatascience.com/reinforcement-learning-made-simple-part-1-intro-to-basic-concepts-and-terminology-1d2a87aa060. Accessed: 2024-01-28.

[35] Doshi, K. (2020b). Transformers explained visually — medium.com. https://medium.com/towards-data-science/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452. [Accessed 01-02-2024].

[36] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14:179–211.

[37] Elsken, T., Metzen, J. H., and Hutter, F. (2019). Neural architecture search: A survey.

[38] Falkner, S., Klein, A., and Hutter, F. (2018). BOHB: robust and efficient hyperparameter optimization at scale. *CoRR*, abs/1807.01774.

[39] Fan, Y., Alon, D., Shen, J., Peng, D., Kumar, K., Long, Y., Wang, X., Iliopoulos, F., Juan, D.-C., and Vee, E. (2023). Layernas: Neural architecture search in polynomial complexity.

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

[41] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202.

[42] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks.

[43] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2015). Lstm: A search space odyssey. *CoRR*, abs/1503.04069.

[44] Guliyev, N. J. and Ismailov, V. E. (2017). On the approximation by single hidden layer feedforward neural networks with fixed weights. *CoRR*, abs/1708.06219.

[45] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.

[46] He, X., Zhao, K., and Chu, X. (2019). Automl: A survey of the state-of-the-art. *CoRR*, abs/1908.00709.

[47] Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory*. Wiley, New York.

[48] Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable baselines. https://github.com/hill-a/stable-baselines.

[49] Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800.

[50] Hinton, G. E. (2007). Boltzmann machine. *Scholarpedia*, 2(5):1668.

[51] Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. M. (1995). The wake-sleep algorithm for unsupervised neural networks. *Science*, 268:1158–1161.

[52] Hinton, G. E. and Sejnowski, T. (1986). Learning and relearning in boltzmann machines. In *Parallel distributed processing: Explorations in the microstructure of cognition*, pages 282–317–. MIT Press, Cambridge, MA.

[53] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580.

[54] Ho, T. K. (1995). Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, page 278, USA. IEEE Computer Society.

[55] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[56] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America*, 79(8):2554–2558.

[57] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257.

[58] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

[59] Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 148:574–591.

[60] Hugging Face (2022). Hugging face deep rl course. https://huggingface.co/learn/deep-rl-course/unit6/introduction. [Accessed 30-01-2024].

[61] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

[62] Ismailov, V. E. (2023). A three layer neural network can represent any multivariate function. *Journal of Mathematical Analysis and Applications*, 523(1):127096.

[63] Jaafra, Y., Luc Laurent, J., Deruyver, A., and Saber Naceur, M. (2019). Reinforcement learning for neural architecture search: A review. *Image and Vision Computing*, 89:57–66.

[64] Jones, T. (2017). Recurrent neural networks deep dive. [Online; accessed 25-March-2021].

[65] Jordan, I. M. (1997). Serial order: A parallel distributed processing approach. *Neural-Network Models of Cognition - Biobehavioral Foundations. Advances in Psychology*, pages 471–495.

[66] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2020). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

[67] Kandasamy, K., Neiswanger, W., Schneider, J., Póczos, B., and Xing, E. P. (2018). Neural architecture search with bayesian optimisation and optimal transport. *CoRR*, abs/1802.07191.

[68] Karnin, E. D. (1990). A simple procedure for pruning back-propagation trained neural networks. *IEEE Transactions on Neural Networks*, 1(2):239–242.

[69] Karpathy, A. (2015). Andrej karpathy blog: The unreasonable effectiveness of recurrent neural networks. [Online; accessed 19-February-2021].

[70] Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69.

[71] Kramer, M. (1991). Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37:233–243.

[72] Kurenkov, A. (2020). A brief history of neural nets and deep learning. *Skynet Today*.

[73] Larson, J. and Michalski, R. S. (1977). Inductive inference of vl decision rules. *ACM SIGART Bulletin*, (63):38–44.

[74] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551.

[75] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324.

[76] LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. pages 598–605.

[77] Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I., and Stoica, I. (2018). RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*.

[78] Lin, L. J. (1993). *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh.

[79] Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's thesis, University of Helsinki.

[80] Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A. L., Huang, J., and Murphy, K. (2017). Progressive neural architecture search. *CoRR*, abs/1712.00559.

[81] Liu, Y., Sun, Y., Xue, B., Zhang, M., and Yen, G. G. (2020). A survey on evolutionary neural architecture search. *CoRR*, abs/2008.10937.

[82] Loshchilov, I. and Hutter, F. (2017). Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101.

[83] Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y. D., Deb, K., Goodman, E. D., and Banzhaf, W. (2018). NSGA-NET: A multi-objective genetic algorithm for neural architecture search. *CoRR*, abs/1810.03522.

[84] Markov, A. A. (1953). The theory of algorithms. *Journal of Symbolic Logic*, 18(4):340–341.

[85] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[86] Miikkulainen, R., Liang, J. Z., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and Hodjat, B. (2017). Evolving deep neural networks. *CoRR*, abs/1703.00548.

[87] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546.

[88] Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

[89] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning.

[90] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[91] Mosqueira-Rey, E., Hernández-Pereira, E., Alonso-Ríos, D., Bobes-Bascarán, J., and Fernández-Leal, Á. (2023). Human-in-the-loop machine learning: A state of the art. *Artificial Intelligence Review*, 56(4):3005–3054.

[92] Mozer, M. and Smolensky, P. (1988). Skeletonization: A technique for trimming the fat from a network via relevance assessment. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988]*, pages 107–115. Morgan Kaufmann.

[93] Narendra, K. S. and Parthasarathy, K. (1990). Identification and control of dynamical systems using neural networks. *IEEE Transactions on Neural Networks*, 1(1):4–27.

[94] Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(48-49).

[95] OpenAI (2020). Chatgpt: A conversational language model by openai. Unpublished.

[96] OpenAI (2021). Dall-e — openai.com. https://openai.com/dall-e-3. [Accessed 31-01-2024].

[97] OpenAI (2023). Gpt-4 — openai.com. https://openai.com/research/gpt-4. [Accessed 03-02-2024].

[98] Park, S., Yun, C., Lee, J., and Shin, J. (2020). Minimum width for universal approximation. *CoRR*, abs/2006.08859.

[99] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[100] Peddinti, V., Povey, D., and Khudanpur, S. (2015). A time delay neural network architecture for efficient modeling of long temporal contexts. In *INTERSPEECH 2015, 16th Annual Conference of the International Speech Communication Association, Dresden, Germany, September 6-10, 2015*, pages 3214–3218. ISCA.

[101] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[102] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268.

[103] PyTorch (2022). Introduction to torchrl. https://pytorch.org/rl/tutorials/torchrl_demo.html. [Accessed 30-01-2024].

[104] Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., and Sutskever, I. (2022). Robust speech recognition via large-scale weak supervision.

[105] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training.

[106] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683.

[107] Ramachandran, P., Zoph, B., and Le, Q. V. (2017). Searching for activation functions. *CoRR*, abs/1710.05941.

[108] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548.

[109] Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

[110] Rosa, M., Afanasjeva, O., Andersson, S., Davidson, J., Guttenberg, N., Hlubucek, P., Poliak, M., Vitku, J., and Feyereisl, J. (2019). BADGER: learning to (learn [learning algorithms] through multi-agent communication). *CoRR*, abs/1912.01513.

[111] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

[112] Ruder, S. (2016). An overview of gradient descent optimization algorithms. [Online; accessed 25-March-2021].

[113] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536.

[114] Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report TR 166, Cambridge University Engineering Department, Cambridge, England.

[115] Sabour, S., Frosst, N., and Hinton, G. E. (2017). Dynamic routing between capsules. *CoRR*, abs/1710.09829.

[116] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015a). Trust region policy optimization. *CoRR*, abs/1502.05477.

[117] Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

[118] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

[119] Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45(11):2673–2681.

[120] Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In *Parallel distributed processing: Explorations in the microstructure of cognition*, pages 194–281–. MIT Press, Cambridge, MA.

[121] Stanford University (1960). *Adaptive "adaline" Neuron Using Chemical "memistors."*. Stanford Electronics Laboratories. Solid State Electronics Laboratory and Widrow, B. and United States. Office of Naval Research and United States. Army. Signal Corps and United States. Air Force and United States. Navy.

[122] Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

[123] Tan, M., Chen, B., Pang, R., Vasudevan, V., and Le, Q. V. (2018). Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626.

[124] Team, K. (2022). Keras documentation: Timedistributed layer — keras.io. https://keras.io/api/layers/recurrent_layers/time_distributed/. [Accessed 01-02-2024].

[125] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023). Llama: Open and efficient foundation language models.

[126] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. cite arxiv:1706.03762 Comment: 15 pages, 5 figures.

[127] Waibel, A., Hanazawa, T., Hinton, G. E., Shikano, K., and Lang, K. (1987). Phoneme recognition using time-delay neural networks. Technical Report TR-I-0006, Advanced Telecommunication Research Institute, International Interpreting Telephony Research Laboratories, Kyoto, Japan.

[128] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford.

[129] Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

[130] Wikimedia Commons (2007). Wikimedia commons - a collection of freely usable media files. [Online; accessed 10-February-2021].

[131] Wilensky, U. (1999). Netlogo. http://ccl.northwestern.edu/netlogo/, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

[132] Yamaguchi, K., Sakamoto, K., Akabane, T., and Fujimoto, Y. (1990). A neural network for speaker-independent isolated word recognition. In *The First International Conference on Spoken Language Processing, ICSLP 1990, Kobe, Japan, November 18-22, 1990*. ISCA.

[133] Zhang, J., He, T., Sra, S., and Jadbabaie, A. (2020). Why gradient clipping accelerates training: A theoretical justification for adaptivity.

[134] Zhang, K., Yang, Z., and Basar, T. (2019). Multi-agent reinforcement learning: A selective overview of theories and algorithms. *CoRR*, abs/1911.10635.

[135] Zheng, Y. (2019). Reinforcement learning and video games.

[136] Zoph, B. and Le, Q. (2017). Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*.

[137] Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578.

[138] Švec, J., Lehečka, J., and Šmídl, L. (2022). Deep LSTM Spoken Term Detection using Wav2Vec 2.0 Recognizer. In *Proceedings of Interspeech 2022*, pages 1886–1890.

Appendix A

Notation Conventions

Notation conventions used in this study are overtaken from [20].

First of all, we define a dataset consisting of samples X and labels Y'.

$$\underset{n\times p}{X} = \begin{bmatrix} X_1 & X_2 & \cdots & X_p \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}$$

$$\underset{1\times p}{Y'} = \begin{bmatrix} Y'_1 & Y'_2 & \cdots & Y'_p \end{bmatrix}$$

where X_1 is the first sample, p is the number of samples and n is the problem dimension. Y' is the vector of labels. A label can be represented as a number or a string. For example, we can set $Y'_1 = "a"$ be a label of sample X_1, which is a sample of phoneme `"a"`. To make it work together with our neural network implementation, each label has a transcript, which is unique for every class. The transcript is so called one-hot vector, a zero vector of length m (number of classes), which has the only one `"1"` at the position corresponding to its class. For example, if we classify 5 phonemes and the class `"a"` was assigned to position 2, its transcript Y_1 would be:

$$\underset{5\times 1}{Y_1} = \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \\ y_{41} \\ y_{51} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

A general matrix of these transcripts Y is then:

$$\underset{m\times p}{Y} = \begin{bmatrix} Y_1 & Y_2 & \cdots & Y_p \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1p} \\ y_{21} & y_{22} & \cdots & y_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mp} \end{bmatrix}$$

We consider Y to be a predicted output of our neural network. Analogically, we get a general matrix of a desired output of a network and those two can be item-wise compared.

$$\underset{m\times p}{U} = \begin{bmatrix} U_1 & U_2 & \cdots & U_p \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1p} \\ u_{21} & u_{22} & \cdots & u_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m1} & u_{m2} & \cdots & u_{mp} \end{bmatrix}$$

Moreover, we decipher the matrices of weights and biases. We have a vector W of weight matrices $W^{(i)}$, which is always of length $(q+1)$, where q is the number of hidden layers.

$$\underset{1\times(q+1)}{W} = \begin{bmatrix} W^{(1)} & W^{(2)} & \cdots & W^{(q+1)} \end{bmatrix}$$

Shapes of matrices $W^{(i)}$ then reveals the network structure. For example we itemize $W^{(1)}$, which carries the information about problem dimension n. Let's assume we have j neurons in the first hidden layer.

$$\underset{j\times n}{W^{(1)}} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \cdots & w_{1n}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & \cdots & w_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1}^{(1)} & w_{j2}^{(1)} & \cdots & w_{jn}^{(1)} \end{bmatrix}$$

Clearly, the first (row) index indicates the neuron we are going to and the second (column) index indicates the neuron we are coming from. A corresponding bias vector would look as follows.

$$\underset{j\times 1}{B^{(1)}} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_j^{(1)} \end{bmatrix}$$

Finally, the error matrix in the output layer of m neurons for p samples is given as follows:

$$\underset{m\times p}{\Delta^{(q+1)}} = \begin{bmatrix} \delta_{11}^{(q+1)} & \delta_{12}^{(q+1)} & \cdots & \delta_{1p}^{(q+1)} \\ \delta_{21}^{(q+1)} & \delta_{22}^{(q+1)} & \cdots & \delta_{2p}^{(q+1)} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{m1}^{(q+1)} & \delta_{m2}^{(q+1)} & \cdots & \delta_{mp}^{(q+1)} \end{bmatrix}$$

Then for $\xi = 1$, the errors corresponding to the first sample X_1 are:

$$\Delta_{(1)}^{(q+1)} = \begin{bmatrix} \delta_{11}^{(q+1)} \\ \delta_{21}^{(q+1)} \\ \vdots \\ \delta_{m1}^{(q+1)} \end{bmatrix}$$

Appendix B

Project API

The project is hosted in a private GitHub repository[1], where both the versioned thesis (serving as documentation for the framework) and the framework's code can be found. Next, the repository contains file `requirments.txt` listing all the required packages and their versions necessary to run the code.

B.1 Framework Implementation

The majority of the code is implemented in Python, primarily leveraging the PyTorch library [99]. The backend engine is complemented by a web-based graphical interface implemented in React/TypeScript. The connection between these two components is established via the MQTT protocol and a public broker. The framework is modularized as depicted in Fig. B.1.

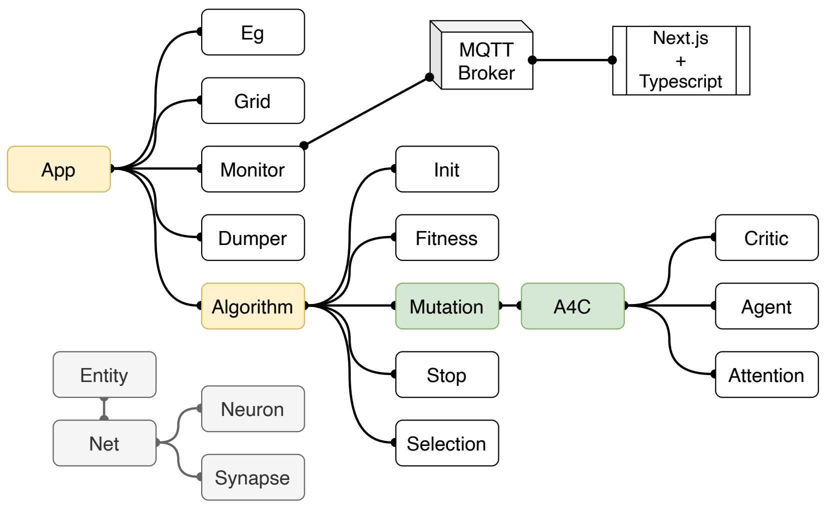


Fig. B.1 Implementation - structure of the framework.

[1]https://github.com/kitt10/phd

The list of all Python modules is detailed in Sec. B.2, with each module representing a methodology outlined in the thesis. The framework is designed to be highly versatile, capable of handling any specified classification problem placed within the `/code/egs/` directory and inheriting from the *Dataset* class declared in `dataset.py`.

Subsequently, each algorithm run is parametrized using a configuration file located in the `/code/cfg/` folder, passed as an argument to the main process. The configuration can be highly granular, covering every aspect of the framework, as demonstrated by an example configuration file in Sec. B.3. The main process is initiated with the following command.

```
$ python app.py -eq EG -cfg CFG_FILENAME -d DEVICE -g H V
```

`-eg -example`: Example (task) name in the `/code/egs/` directory. Required.

`-cfg -config`: Name of the configuration file in the `/code/cfg/` folder. Default: `default`

`-g -grid`: Capacity of the *Grid*, two integers: `V` $\sim$ width and `H` $\sim$ height. Default: `3 1`

`-d -device`: Force run on specific device (*cpu* / *cuda*). Detected automatically otherwise.

`-s -seed`: Fixing the random seed for reproducability. Not used by default.

`-e -eval`: If present, the evaluation mode is active. By default it is inactive.

An example of the command to initiate the main genetic algorithm is provided below. It is important to specify the output folder where the trained models, responsible for the mutation operation, will be saved (`dumper/save_models_dir` in the configuration file—see Sec. B.3). Optionally, starting from a pre-trained checkpoint (`mutation/kwargs/load_models_dir`) is also supported.

```
$ python app.py -eq XOR -d cpu -g 4 1
```

The best-found network architecture during the training process is saved to the directory of the example, along with the statistics, in the `/egs/EG/best/` directory, where `EG` is the name of the example. Finally, if we want to run the mutation operation using a pre-trained model on a network from its current state (by default generated randomly based on the `init` parameters in the configuration file), we can execute the following command and observe the mutation process on-the-fly. The pre-trained model is specified in the configuration file.

```
$ python app.py -eq EG --eval
```

The real-time evolution of the network architecture can be observed using the web-based interface (see Sec. 4.3.3) on a public domain. However, the MQTT broker configuration is required for this purpose, which is not provided in the thesis publicly for obvious reasons. Finally, the `watcher.py` script can independently monitor training statistics, also in real-time.

B.2 Structure of the Workspace

```
GitHub repo
├─ thesis
└─ code
   ├─ cfg
   │  └─ config.yml
   ├─ egs
   │  ├─ BASIC2D
   │  ├─ XOR
   │  ├─ CHAL2D
   │  ├─ MNIST
   │  ├─ INTENTS
   │  └─ dataset.py
   └─ lib
      ├─ a4c.py
      ├─ actor.py
      ├─ agent.py
      ├─ algorithm.py
      ├─ attention.py
      ├─ critic.py
      ├─ crossover.py
      ├─ dumper.py
      ├─ entity.py
      ├─ evaluation.py
      ├─ fcn.py
      ├─ fitness.py
      └─ generation.py

...
└─ ...
   ├─ ...
   │  ├─ grid.py
   │  ├─ init.py
   │  ├─ monitor.py
   │  ├─ mutatees.py
   │  ├─ mutation.py
   │  ├─ net.py
   │  ├─ neuron.py
   │  ├─ parents.py
   │  ├─ plotter.py
   │  ├─ selection.py
   │  ├─ stop.py
   │  └─ synapse.py
   ├─ models
   ├─ monitor
   │  ├─ deploy
   │  └─ devel
   ├─ runs
   ├─ app.py
   └─ watcher.py
```

B.3 Configuration File Example

```
algorithm:
  G: 4
  net:
    lr: 0.07
    loss_lim: 0.01
init:
  method: InitDefault
  kwargs:
    #method: cherries
    method: zero
mutation:
  method: MutationA4C
  kwargs:
    load_models_dir: ./models/xor
    T: 50
    Z: 5
    k: 7
    reward_done: 10.
    gamma: 0.95
    lam: 0.98
    clip: 0.2
    centralized: False
    sync: False
    shuffle: True
    att_updates:
      #- critic
      - agents
    lr_critic: 0.001
    lr_actor: 0.001
    actor_hidden_dim: 64
    critic_hidden_dim: 64
fitness:
  method: FitnessDefault
  kwargs:
    Fm: 1.
    beta: 0.35
    retrain_epochs: 0
stop:
  method: StopHITL
  kwargs:
    prob_thr: 0.75
selection:
  method: SelectionFiftyFifty
  kwargs: {}
dumper:
  path: ./runs
  period: 1
  save_models_dir: ./models/xor-2
mqtt:
  ip: '0.0.0.0'
  port: 1883
  uname: 'XXX'
  passwd: 'XXX'
  topic: 'XXX'
monitor:
  on_visual: True
  on_textual: False
  width: 200
  height: 250
eval:
  T: 100
  retrain: 10
```

PUBLICATIONS

2024 ŠVEC, J.; BULÍN, M.; FRÉMUND, A.; POLÁK, F. Asking Questions Framework for Oral History Archives. In: Advances in Information Retrieval 45th European Conference on Information Retrieval, ECIR 2024, Glasgow, Scotland, March 24–28, 2024, Proceedings. Heidelberg: Springer, 2024.

2023 BULÍN, M.; ADAMEC, M.; NEDUCHAL, P.; HRÚZ, M.; ŠVEC, J. Voice-Interactive Learning Dialogue on a Low-Cost Device. In: Pattern Recognition 7th Asian Conference, ACPR 2023, Kitakyushu, Japan, November 5–8, 2023, Proceedings, Part III. Heidelberg: Springer, 2023, s. 369-382. ISBN 978-3-031-47664-8, ISSN 0302-9743.

- BULÍN, M.; ŠVEC, J.; IRCING, P. The System for Efficient Indexing and Search in the Large Archives of Scanned Historical Documents. In: Advances in Information Retrieval 45th European Conference on Information Retrieval, ECIR 2023, Dublin, Ireland, April 2–6, 2023, Proceedings, Part III. Heidelberg: Springer, 2023, s. 206-210. ISBN 978-3-031-28240-9, ISSN 0302-9743.
- KUNEŠOVÁ, M.; MATOUŠEK, J.; LEHEČKA, J.; ŠVEC, J.; MICHÁLEK, J.; TIHELKA, D.; BULÍN, M.; HANZLÍČEK, Z.; ŘEZÁČKOVÁ, M. Ensemble of Deep Neural Network Models for MOS Prediction. In: ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). New York: IEEE, 2023. ISBN 978-1-72816-328-4, ISSN 1520-6149.
- ŠVEC, J.; BULÍN, M.; FRÉMUND, A.; POLÁK, F. Asking Questions: an Innovative Way to Interact with Oral History Archives. In: Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH. New York: ISCA, 2023, s. 3679-3680. ISSN 2308-457X.
- BULÍN, M. Multimodal Low-Cost Robotic Entity based on Raspberry Pi. In: Studentská vědecká konference 2023 - magisterské a doktorské studijní programy, sborník rozšířených abstraktů. Plzeň: Západočeská univerzita v Plzni, 2023, s. 38-39. ISBN 978-80-261-1161-0.

2022 ŠVEC, J.; FRÉMUND, A.; BULÍN, M.; LEHEČKA, J. Transfer Learning of Transformers for Spoken Language Understanding. In: Text, Speech, and Dialogue 25th International Conference, TSD 2022, Brno, Czech Republic, September 6–9, 2022, Proceedings. Cham: Springer International Publishing, 2022, s. 489-500. ISBN 978-3-031-16269-5, ISSN 0302-9743.

- BULÍN, M. Real-time Robot Teaching Dialog. In: Studentská vědecká konference 2022 - magisterské a doktorské studijní programy, sborník rozšířených abstraktů. Plzeň: Západočeská univerzita v Plzni, 2022, s. 32-33. ISBN 978-80-261-1095-8.

2021 BULÍN, M. On Using Multi-Agent Technologies to Build Neural Networks. Rigorous thesis, University of West Bohemia, Univerzitní 2732/8, 301 00 Plzeň 3, 2021.

- BULÍN, M. On Growing Neural Networks with Multi-Agent Principles. In: SVK 2021 - magisterské a doktorské studijní programy, sborník rozšířených abstraktů. Plzeň: Západočeská univerzita v Plzni, 2021, s. 28-29. ISBN 978-80-261-1022-4.
- GRUBER, I.; HRÚZ, M.; IRCING, P.; NEDUCHAL, P.; ZÍTKA, T.; HLAVÁČ, M.; ZAJÍC, Z.; ŠVEC, J.; BULÍN, M. OCR Improvements for Images of Multipage Historical Documents. In: 23rd International Conference, SPECOM 2021, St. Petersburg, Russia, September 27–30, 2021, Proceedings. Cham: Springer, 2021, s. 226-237. ISBN 978-3-030-87801-6, ISSN 0302-9743.

2020 BULÍN, M; ŠVEC, J.; IRCING P. Full-text search through MALACH archive using speech recognition. The 10th Anniversary Conference of Malach Centre for Visual History, 2020.

- GRUBER, I.; IRCING, P.; NEDUCHAL, P.; HRÚZ, M.; HLAVÁČ, M.; ZAJÍC, Z.; ŠVEC, J.; BULÍN, M. An Automated Pipeline for Robust Image Processing and Optical Character Recognition of Historical Documents. In: Speech and Computer, 22nd International Conference, SPECOM 2019, St. Petersburg, Russia, October 7-9,2020, Proceedings. Cham: Springer, 2020, s. 166-175. ISBN 978-3-030-60275-8, ISSN 0302-9743.

2019 BULÍN, M.; ŠMÍDL, L.; ŠVEC, J. On Using Stateful LSTM Networks for Key-Phrase Detection. In: Text, Speech, and Dialogue 22nd International Conference, TSD 2019, Ljubljana,Slovenia, September 11-13, 2019, Proceedings. Cham: Springer, 2019, s. 287-298. ISBN 978-3-030-27946-2, ISSN 0302-9743.

- BULÍN, M. Reading text in images with EAST and Kaldi. In: Studentská vědecká konference 2019 - magisterské a doktorské studijní programy, sborník rozšířených abstraktů. Plzeň: Západočeská univerzita v Plzni, 2019, s. 28-29. ISBN 978-80-261-0867-2, ISSN neuvedeno.

2018 BULÍN, M.; ŠMÍDL, L.; ŠVEC, J. Towards Network Simplification for Low-Cost Devices by Removing Synapses. In: Speech and Computer 20th International Conference, SPECOM 2018 Leipzig, Germany, September 18–22, 2018, Proceedings. Cham: Springer Nature Switzerland AG, 2018, s. 58-67. ISBN 978-3-319-99578-6, ISSN 0302-9743.

- ŠVEC, J.; BULÍN, M.; PRAŽÁK, A.; IRCING, P. UWebASR – Web-based ASR engine for Czech and Slovak. In: CLARIN Annual Conference 2018 Proceedings. 2018, s. 190-193.
- BULÍN, M. Real-time keyword spotting on RaspberryPi with pruned LSTM. In: Studentská vědecká konference 2018 - magisterské a doktorské studijní programy, sborník rozšířených abstraktů. Plzeň: Západočeská univerzita v Plzni, 2018, s. 33-34. ISBN 978-80-261-0867-2, ISSN neuvedeno.

2017 BULÍN, M. Optimization of Neural Network. Diplomová práce, Západočeská univerzita v Plzni, Univerzitní 2732/8, 301 00 Plzeň 3, 2017.

- BULÍN, M.; ŠMÍDL, L. Insight of Neural Network by Removing Synapses. In: Studentská vědecká konference 2017 - magisterské a doktorské studijní programy, sborník rozšířených abstraktů. Plzeň: Západočeská univerzita v Plzni, 2017, s. 39-40. ISBN 978-80-261-0706-4.

2016 BULÍN, M. Classification of Terrain based on Proprioception and Tactile Sensing for Multi-legged Walking Robot. Master Thesis, University of Southern Denmark, Campusvej 55, 5230 Odense, Denmark, 2016.

2014 BULÍN, M. Detekce sémantických entit. Bakalářská práce, Západočeská univerzita v Plzni, Univerzitní 2732/8, 301 00 Plzeň 3, 2014.