UNIVERSITY
OF WEST BOHEMIA

Rigorous Thesis

# On Using Multi-Agent Technologies to Build Neural Networks

**Ing. Martin Bulín, MSc.**

*A thesis 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:
Department of Cybernetics

April 16, 2021

# Declaration of Authorship

I, Ing. Martin Bulín, MSc., declare that this thesis titled, “On Using Multi-Agent Technologies to Build Neural Networks” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

Date:

UNIVERSITY OF WEST BOHEMIA

# *Abstract*

Faculty of Applied Sciences

Department of Cybernetics

Doctor of Philosophy (Ph.D.)

**On Using Multi-Agent Technologies to Build Neural Networks**

by Ing. Martin Bulín, MSc.

The unreliable behaviour of deep neural networks for specific use cases slows down their deployment in real world applications. The mainstream way to catch up the remaining percents in performance relies on iterative tuning of hyper-parameters, collecting new data and increasing computational power, while the behaviour inside the trained model is usually kept shrouded in mystery. As the commonly used network architectures have been found unnecessarily complicated as well as limited, e.g. by the strict arrangment of neurons into layers, this work proposes an alternative method to work with neural principles and to design network architectures. The neural principles are combined with the theory of multi-agent systems and reinforcement learning with the goal of generating tailored networks for given classification tasks. The tailored networks are believed to shake off redundant parts and to allow less restricted way of connecting neurons in comparison to standard structures. This could possibly improve the performance as well as enable targeted fixes in the network even after the training. The main loop of the algorithm has been already implemented and tested on two basic 2D experiments with expected results. The work is currently at the state of prooving the scalability to multidimensional classification tasks.

# *Acknowledgements*

I would like to express my gratitude to my supervisor Prof. Ing. Josef PSUTKA, CSc. for his support of my work.

Next, I would like to thank my collegues Ing. Jan ŠVEC, Ph.D. and Ing. Luboš ŠMÍDL, Ph.D. for their valuable comments regarding this work.

Finally, I would like to thank all members of my family for their continuous support of my studies.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **A2C** | **A**dvantage **A**ctor **C**ritic |
| **A3C** | **A**synchronous **A**dvantage **A**ctor **C**ritic |
| **AE** | **A**uto**e**ncoder |
| **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 |
| **BM** | **B**oltzmann **M**achine |
| **BP** | **B**ack**P**ropagation |
| **BPTT** | **B**ack**P**ropagation **T**hrough **T**ime |
| **BRNN** | **B**i-directional **R**ecurrent **N**eural **N**etwork |
| **CIFAR** | **C**anadian **I**nstitute **F**or **A**dvanced **R**esearch |
| **CNN** | **C**onvolutional **N**eural **N**etwork |
| **DBN** | **D**eep **B**elief **N**etwork |
| **DDPG** | **D**eep **D**eterministic **P**olicy **G**radient |
| **DL** | **D**eep **L**earning |
| **ESN** | **E**cho **S**tate **N**etwork |
| **GAN** | **G**enerative **A**dversarial **Network** |
| **GOFAI** | **G**ood **O**ld-**F**ashioned **AI** |
| **GPU** | **G**raphics **P**rocessing **U**nit |
| **GPT** | **G**enerative **P**re-trained **T**ransformer |
| **IndRNN** | **Ind**ependently **R**ecurrent **N**eural **Network** |
| **HMM** | **H**idden **M**arkov **M**odel |
| **LSTM** | **L**ong **S**hort-**T**erm **M**emory (RNN) |
| **MARL** | **M**ulti-**A**gent **R**einforcement **L**earning |
| **MAS** | **M**ulti-**A**gent **S**ystem |
| **MDP** | **M**arkov **D**ecision **P**rocess |
| **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 |
| **RBM** | **R**estricted **B**oltzmann **M**achine |
| **ReLU** | **Re**ctified **L**inear **U**nit |
| **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 |
| **SRN** | **S**imple **R**ecurrent **N**etwork |
| **SVM** | **S**upport **V**ector **M**achine |
| **TDNN** | **T**ime-**D**elay **N**eural **N**etwork |

# List of Symbols

| **General** | |
|---|---|
| $n$ | problem dimension (sample size; number of features) |
| $m$ | number of classes |
| $p$ | number of samples |

| **Neural Networks** | |
|---|---|
| $X$ | $n$-by-$p$ matrix: network input (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 |
| $B^{(i)}$ | vector of $r$ biases for $r$ neurons in $i^{th}$ layer |
| $Z^{(i)}$ | $r$-by-$p$ matrix of activations for $r$ neurons in $i^{th}$ layer for all samples |
| $A^{(i)}$ | $r$-by-$p$ matrix of activities of $r$ neurons in $i^{th}$ layer for all samples |
| $\Delta^{(i)}$ | $r$-by-$p$ matrix of errors on $r$ neurons in $i^{th}$ layer for all samples |
| $Y$ | $m$-by-$p$ matrix: predicted network output for all samples ($Y = A^{(q)}$) |
| $U$ | $m$-by-$p$ matrix: desired network output (targets) for all samples |
| $q$ | number of hidden layers |
| $\tau$ | number of timesteps (context-dependent structures) |
| $a_k^{(i)}$ | activity of $k^{th}$ neuron 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 |
| $b_k^{(i)}$ | bias connected to $k^{th}$ neuron in $i^{th}$ layer |
| $z_k^{(i)}$ | activation of $k^{th}$ neuron in $i^{th}$ layer |
| $f(\cdot)$ | transfer function |
| $f'(\cdot)$ | derivative of the transfer function $f(\cdot)$ |
| $a^{<t>}$ | activity of a (recurrent) neuron at time $t$ |
| $o^{<t>}$ | output of a (recurrent) neuron at time $t$ |
| $c^{<t>}$ | state of a (recurrent) neuron at time $t$ |

**Multi-Agent Systems**

| | |
|---|---|
| $S^{<t>}$ | state at time $t$ |
| $R^{<t>}$ | reward at time $t$ |
| $A^{<t>}$ | action at time $t$ |
| $\Pi$ | policy |
| $\phi$ | set of states (a state space) |
| $\alpha_S$ | set of actions available from state $S$ |
| $\rho(S^{<t>}, A^{<t>})$ | temporal difference for state $S$ and action $A$ at time $t$ |
| $J(\cdot)$ | objective function for Policy Gradients |
| $Z$ | a condition (a boolean expression) |

**Graphical**

| | |
|---|---|
| 🟩 | input node |
| 🔵 | hidden feedforward node |
| 🟣 | hidden recurrent node |
| ▶ | output node |
| $\overset{w}{\otimes}\rightarrow$ | multiplication (signal times $w$) |
| $\overset{b}{\oplus}\rightarrow$ | addition (signal plus $b$) |
| $\rightarrow \boxed{f} \rightarrow$ | function of the signal: $f(\cdot)$ |
| $\xrightarrow{\uparrow}$ | concatenate two signals |
| $\xrightarrow{\downarrow}$ | copy a signal |

# Chapter 1

# Introduction

The remarkable progress over the recent years in the field that is solemnly called *artificial intelligence (AI)* has put the public opinion in an ethical dilemma of either supporting this dishy research direction or hesitating to accept decision making machines at all. People from the outside of the AI community have been justifiably wondering about the purpose of creating machines that seem to be intelligent and even a few experts support the worries and conspiracy theories. In my opinion, the current SotA is still not even close to the general AI capable of a complex human-like behaviour or even gaining consciousness. However, when it comes to solve a single specific task (e.g. face recognition, language translation, speech analysis, etc.), the performances of the latest AI methods are nearly perfect and in many cases they even outperform humans, especially in terms of speed and fatigue resistance. In these cases, the involvement of the AI in our everyday life makes sense and can significantly help people, the Google search engine is the one-for-all example. From my point of view, we should adhere to the following principles when deploying AI systems in the real world:

- *Ethical purpose*; The great power must be used in the right direction.
- *Motivation*; The decision to deploy the AI system must make sense.
- *Zero fault tolerance*; The developed AI system must be reliable.

The first two of them are about the composition of the task and the data origin. The reliability additionally depends on the quality of the AI method.

The first methods, nowadays known as the *Good Old Fashioned AI (GOFAI)*, were based on state space search techniques and on the decomposition to symbolic rules defined by the developer. The rules were deterministic, their behaviour was predictable and in case of a failure, the developer made the required fix at the right place in the system. However, in case of more sophisticated tasks the solution space became too large to be searched completely and hence, the methods were enhanced with heuristics. Since then, although the method has been capable of reaching a solution in a finite time, the warranty of the optimality and rationality has been lost, the behaviour has become unpredictable and targeted failure fixing has been disabled. Even though the methods have evolved, we face the same situation in AI today.

The rise of deep learning over the last fifteen years has made *artificial neural networks (ANNs)* the monopolistic SotA method for classification tasks and training neural networks on various datasets is undoubtedly one of the hottest research directions nowadays.

**The Gap.** The only shortcoming that slows down the deployment of ANNs in several domains (e.g. self-driving cars, healthcare) is their unreliable and unpredictive behaviour in special situations. Due to the trial and error procedure of training ANNs nowadays and due to the complicated architectures, the deep behaviour in trained models is shrouded in mystery and thus, targeted fixes in these models are out of the question and it is impossible to catch up the remaining percents in performance.

Instead of breaking the deadlock by tuning hyper-parameters, this work is devoted to the developement of an alternative method for working with the neural principles. Based on the theory of *multi-agent* systems, the *reinforcement learning* is used to generate tailored network architectures enabled for future targeted adjustments. The proposed method is aimed to be general for any non-sequential classification data and initial experiments have been already performed with expected results. Moreover, new unexplored research directions have been opened for the future work.

## 1.1 Thesis Outline

This work is supposed to serve as a preliminary report of the ongoing research related to the final dissertation thesis. Its purpose is to summarize related studies that may represent the baseline as well as inspiration to some extend. Next, it should state the objective of the global project and explain the proposed method, ideally, including first experiments.

Chapter 2 is devoted to neural networks, starting with a timeline of significant breakthroughs and their inspiration in biological cells (Sec. 2.1). As this work is about building networks, there are the mostly used architectures in Sec. 2.2 sorted into the *feedforward* (Sec. 2.2.1) and *recurrent* (Sec. 2.2.2) categories. The common *backpropagation* algorithm, its extentions and its limitations are described in Sec. 2.3. Section 2.4 contains methods that did not fit to the categories above and also special techniques that help deal with the limitations of the learning algorithm. In Sec. 2.5, the SotA results and top real-world applications are briefly mentioned. The last section (2.6) of the second chapter is devoted to the architecture search algorithms that are the most related to this work.

Chapter 3 is about the second cornerstone of this work - the theory of *multi-agent systems*. The *reinforcement learning* procedure is captured in Sec. 3.1 with the extention to *multi-agent reinforcement learning (MARL)* in Sec. 3.1.2. Commonly used control learning algorithms (RL policies) are described in Sec. 3.1.1.

Chapter 4 defines the overall project objective and by splitting it into partial goals also reveals the current state of the project.

Chapter 5 is devoted to the novelty of the project. Section 5.1 explains the proposed method and Sec. 5.2 provides two initial experiments.

The design choices of the proposed method, eventual shortcomings and directions of the future work are discussed in Chap. 6. Chapter 7 concludes this thesis.

# 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 (McCulloch and Pitts, 1943). 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. When dealing with a specific (not complex) task, the results are often fascinating (see top applications in Sec. 2.5.2) and therefore methods based on the artificial neural network principle are justifiably considered to be the state-of-the-art classifiers.

The artificial neural network collocation can be interpreted as a general term standing for several slightly different classifiers having the neural basis in common and differing mainly in architectures and data structures they are capable of dealing with. There is a wide range of issues in many areas of a human interest turnable into a machine-learning problem, while the key to success (or at least the first step of it) always lies in a correct problem formulation and data representation. Then, based on the task definition and the type of data to be processed, the optimal ANN architecture is chosen. In Sec 2.2, the mostly used network architectures are described and arranged into two categories depending on the dynamics of data processing:

1. Feedforward architectures (*static* systems, Sec. 2.2.1);
2. Recurrent architectures (*dynamic* systems, Sec. 2.2.2).

Besides the architecture-wise sorting of ANNs, the other way is to go through the most popular breakthroughs over time chronologically (Kurenkov, 2020). As stated above, the first signs 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 (see Sec 2.2.1.1). Thus the foundations of the majority of today's neural networks were laid and architectures solely composed of some derivatives of these perceptrons work well and are still being used till today.

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 (Stanford University, 1960). 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 hidden perceptrons learn had not been invented yet. This epoch is known as the "AI winter", as especially skeptical conclusions of the Minsky's work *Perceptrons* (Minsky and Papert, 1969) caused a freeze to funding and publications in AI.

The key *Backpropagation* learning algorithm (see Sec. 2.3) based on the chain rule was firstly derived and implemented to run on computers by Finnish student S. Linnainmaa in 1970 (Linnainmaa, 1970) and later on proposed to be used for neural networks after analyzing it in depth in (Werbos, 1974), who was, interestingly, loosely inspired by Sigmund Freud's psychological theories about modeling the human mind with the concept of a backwards 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* (Minsky and Papert, 1969), the approach was popularized more than a decade later in (Rumelhart, Hinton, and Williams, 1986). Finally, the mathematical proof that multiple layers allow neural networks to theoretically implement any function, and certainly XOR, was given in (Hornik, Stinchcombe, and White, 1989). Since then, ANNs have become dishy again and started to be applied to real-world applications, such as the *Handwritten Zip Code Recognition* problem (LeCun et al., 1989).

Depending on the nature of a particular task, several network architectures and optimization methods have been developed over the years. To briefly list the most interesting breakthroughs in a chronological order:

1982 *Hopfield network* (Hopfield, 1982); 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 (see Sec. 2.2.2.5).

- *Self-organizing maps* (Kohonen, 1982); 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. 2.4.4).

1986 *Boltzmann machine* (Hinton and Sejnowski, 1986); This approach can be seen as a stochastic and generative counterpart of a Hopfield network. The restricted version (RBM) is being used in deep learning for weights pretraining till today (see Sec. 2.2.2.5).

1987 *TDNN - Time Delay Neural Network* (Waibel et al., 1987); Mainly motivated by the speech recognition task, there was a call to consider context dependencies in data. The time-delay network is a special version of a multi-layer feedforward neural network with the ability of context modeling and classification of patterns with shift-invariance (more in 2.2.1.2).

- *Autoencoders* (Bourlard and Kamp, 1987); Based on the neural principles, being unsupervised though, ANN structures started to be used for compression and data encoding tasks (see Sec. 2.4.4).

1990 *Backpropagation through time* (Elman, 1990); 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. 2.2.2).

- *Application in robotics, control engineering and games* (Narendra and Parthasarathy, 1990); At that time, ANNs started to be used as decision makers in the third branch of machine learning - *reinforcement learning* (see Sec. 3.1). The research in (Lin, 1993) 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* (Bromley et al., 1993); 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. 2.4.4).

1995 *Wake-sleep algorithm* (Hinton et al., 1995); G. E. Hinton and his team kept working on some extra tricks for a slightly different belief net setup, which was later on deemed *The Helmholtz Machine* (Dayan et al., 1995). It basically allowed the training of Boltzmann Machines to be done much faster (see Sec. 2.2.2.5).

- *Other (not ANN) methods*; With the idea of the kernel trick (Cortes and Vapnik, 1995) 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 (Ho, 1995) proved to be very effective and had a lovely mathematical theory behind them.

1997 *LSTM - Long Short-Term Memory* (Hochreiter and Schmidhuber, 1997); Most likely 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 (more in Sec. 2.2.2.1).

- *BRNN - Bidirectional Recurent Neural Network* (Schuster and Paliwal, 1997); 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. 2.2.2.3).

1998 *CNN - Convolutional Neural Network* (LeCun et al., 1998); 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. 2.2.1.3).

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

2006 *A fast learning algorithm for deep belief nets* (Bengio et al., 2006); 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.

2014 *GRU - Gated Recurrent Unit* (Chung et al., 2014); Alongside the LSTM cell, GRU is the other gating mechanism that is commonly being used in recurrent structures nowadays. 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. The GRU cell is described in Sec. 2.2.2.2.

- *Attention mechanism*; When processing a large amount of information, attending to a certain part of it is one of the most powerful concepts in deep learning nowadays. Important parts of the input data are enhanced and the rest is faded out. The importance of individual parts is learned and depends on the context. The mechanism was firstly used in (Bahdanau, Cho, and Bengio, 2014) for sequence-to-sequence learning (more in Sec. 2.4.2).
- *GAN - Generative Adversarial Network* (Goodfellow et al., 2014); 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. 2.4.4).

2017 *Transformer* (Vaswani et al., 2017); The latest significant contribution in the field of ANNs was introduced in 2017 and is primarily being used for natural language processing tasks such as translation or text summarization. It is designed to handle sequential data, but unlike RNNs, it does not require the data to be processed in order. This feature allows for parallelization and so saves training times (more in 2.2.2.4). One the massively used concepts is called *multi-headed self-attention.* The recent trend is to use pretrained systems based on *Transformers* such as *BERT (Bidirectional Encoder Representations from Transformers)* or *GPT (Generative Pre-trained Transformer)*, which have been trained on huge and general language datasets and can be fine-tuned to specific tasks.

A complete list of important ANN techniques described in more detail is presented in sections below (mainly Sec. 2.2 and Sec. 2.4), 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 (Benistant et al., 2016). 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.

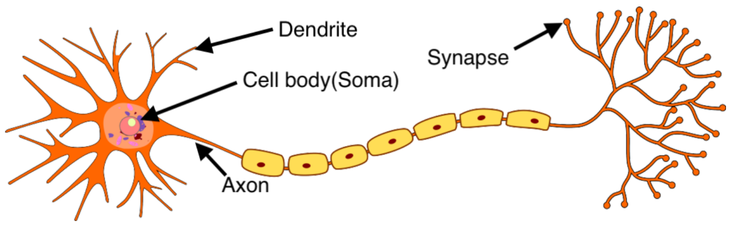


FIGURE 2.1: A biological neural cell. (Wikimedia Commons, 2007)

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 (Dent Neurologic Institute, 2021):

- *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 $10^{15}$ bytes.
- *Asynchronous processing and fault tolerance* - minor failures will not result in memory loss. The architecture is decentralized.
- *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.

## 2.2 Architectures

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.

The basic sorting of machine learning problems is illustrated in Fig. 2.2. As a matter of fact, each of ML problems is described by numerical data and its arrangment in terms of structure as well as eventual dependencies in it.

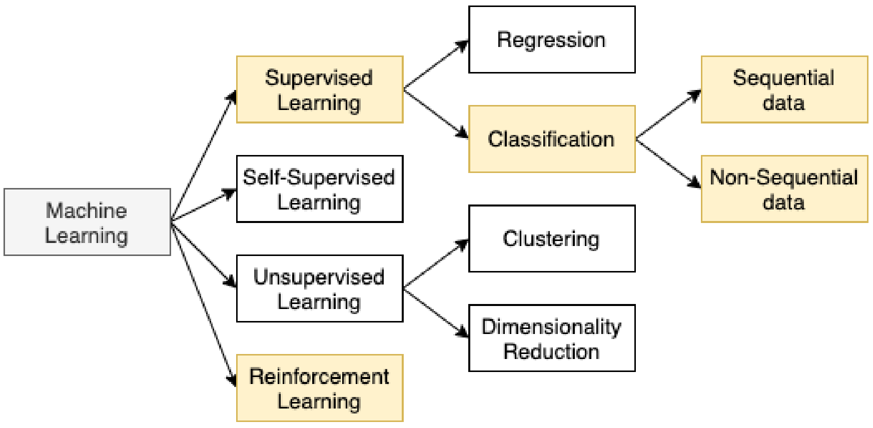


FIGURE 2.2: Task types in machine learning (yellow $\sim$ focused in this work).

In this work, the proposed method is targeted for non-sequential classification data (see Sec. 5.1). In the following sections, the most popular network architectures are sorted into two categories depending on their dynamics:

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

where $z_i^{<t>}$ is the state of $i^{th}$ neuron at time $t$. Static systems (architectures) are commonly called *feedforward* (Sec. 2.2.1) and dynamic architectures, graphically illustrated with loops in them, are called *recurrent* (Sec. 2.2.2).

### 2.2.1 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* (Rosenblatt, 1958). In Fig. 2.3, with the reference to the biological template in Fig. 2.1, there are *dendrites* 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 in App. A1).

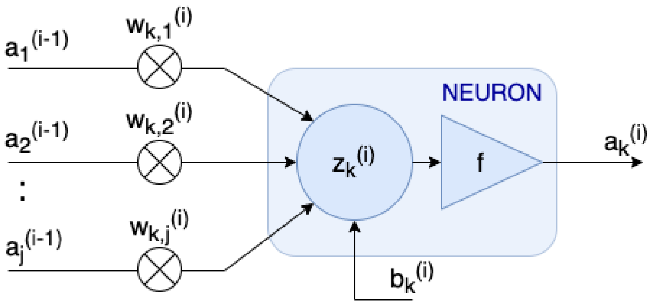


FIGURE 2.3: An artificial neuron.

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. 2.2.1.1) to get the neuron *activity* (Eq. 2.3).

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

#### 2.2.1.1 Multi-Layer Perceptron (MLP)

The default (vanilla) neural network consists of multiple nodes arranged into layers. In Fig. 2.4, there is an example of such a structure with 2 input nodes (green), 1 output node (red) and one hidden layer of 3 nodes.

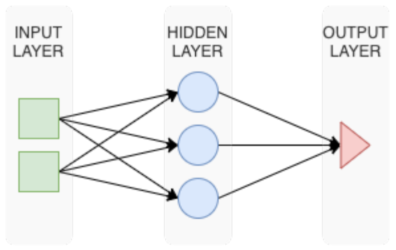


FIGURE 2.4: Example of a feedforward (MLP) architecture.

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 (Hornik, Stinchcombe, and White, 1989), 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. 2.3.

**Transfer (activation) functions.**[1] The learning algorithm (Sec. 2.3) needs the activation function to be (easily) differentiable. The most common activation functions are *hyperbolic tangent*, *sigmoid* and *ReLu* (Fig. 2.5).

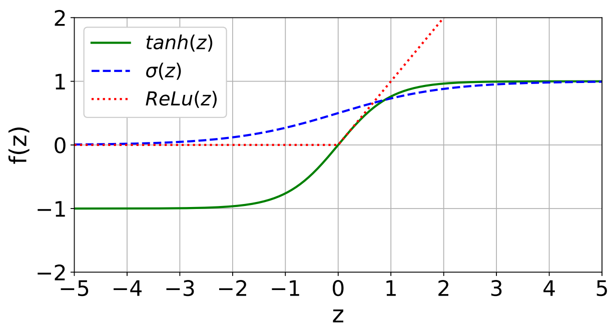


FIGURE 2.5: Common transfer (activation) functions.

#### 2.2.1.2 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.

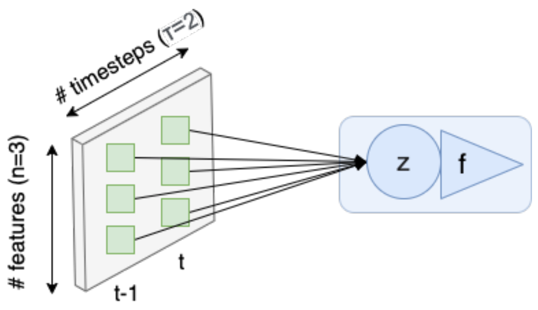


FIGURE 2.6: Single TDNN cell connected to the input layer.

[1]The illustrated transfer functions are used in other architectures as well.

The idea was firstly presented in (Waibel et al., 1987)[2] on the task of phoneme recognition. In Fig. 2.6, 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. 2.3), but there is a difference in the arrangement of its inputs. As shown graphically in Fig. 2.6 and mathematically in Eq. 2.4, there are additionally past (time-delayed) features included. There is a window of $\tau$ timesteps 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.

$$z = \sum_{i=1}^{n} \sum_{j=0}^{\tau-1} [x_i^{<t-j>} \cdot w_{i,j}] + b \tag{2.4}$$

For two-dimensional signals (such as time-frequency patterns or images) there is a 2D context window. Usually there are multiple TDNN units ($\psi$ in Fig. 2.7) arranged into layers, while higher layers generally model coarser levels of abstraction as they have inputs from wider context windows than lower layers.

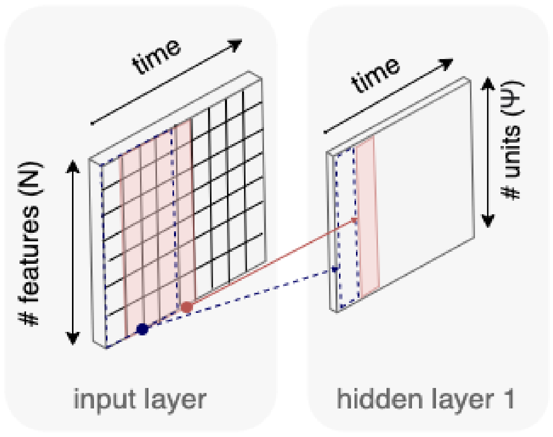


FIGURE 2.7: A time-shifted window over the input data stream for a TDNN network.

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. 2.3), 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. 2.8, there are two design choices illustrated: 1/ for time $t$, the contextual window may include future timesteps as well as past timesteps

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

(here <t-6, t+3>); 2/ the number of timesteps can be subsampled in order to reduce the number of operations (Peddinti, Povey, and Khudanpur, 2015).

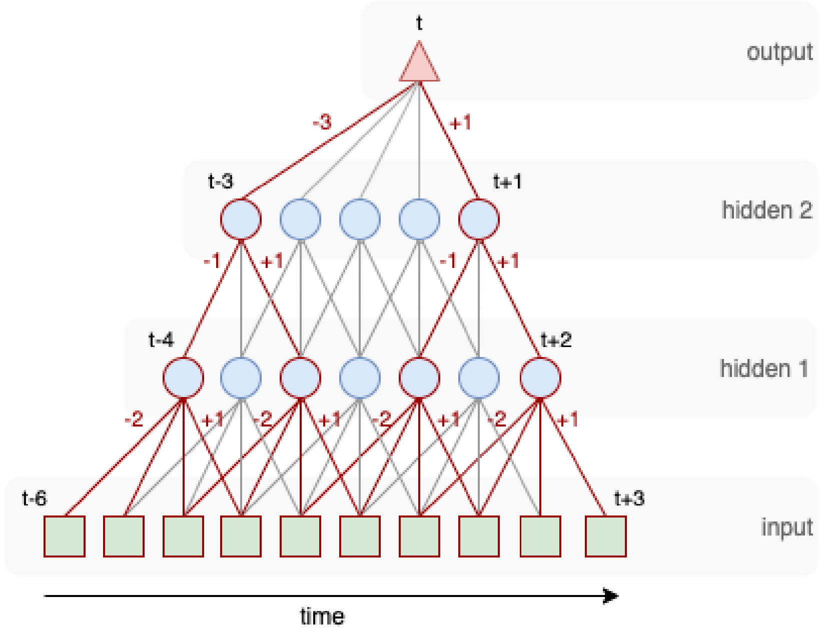


FIGURE 2.8: The TDNN principle with subsampling (red) and without subsampling (red + gray).

#### 2.2.1.3 Convolutional Network (CNN)

Following the theory of receptive fields in the human visual cortex (Hubel and Wiesel, 1959), there was the idea of so-called *neocognitron* in (Fukushima, 1980), which can be considered the origin of the CNN architecture. However, the standard reference is from (LeCun et al., 1998), 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. 2.2.1.2) 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. 2.2.1.1) has two points:

- *Parameters reduction* - using the MLP, a typical 256x256 image on the input results in $56,000 \cdot \psi$ parameters, where $\psi$ 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.

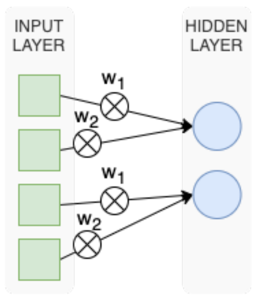
Compared to the MLP approach, many synapses are actually removed and the decision which synapses remain is based on our understanding of the space-importance property.

[3]The approach is identical to the TDNN (Sec. 2.2.1.2) in case of stride = 1 (1D data).

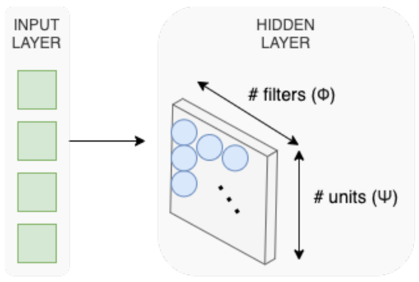
Fig. 2.9 explains the principle of the convolution on a 1D input. There are several hyper-parameters to be set when a CNN is designed:

- *Filter size* - the set of shared weights is called a *filter*. In Fig. 2.9a, 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. 2.9a, stride = 2.
- *Padding* - optionally, the input space can be padded by zeros around its boundaries. There is no padding in the example in Fig. 2.9a.
- *Number of filters* - adding more filters is illustrated in Fig. 2.9b. Each filter is then defined by its own set of weigths and is shared by connections to $\psi$ units in the following layer.

The number of units $\psi$ in the following layer depends on stride and padding.



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



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

FIGURE 2.9: 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 $\psi_1$, height $\psi_2$, number of filters $\phi$).

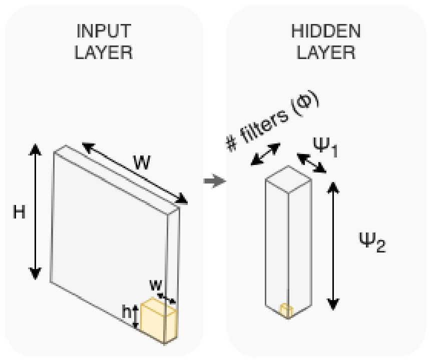


FIGURE 2.10: CNN principle (2D data). The yellow-marked $w \times h$ filter in the input layer corresponds to the little yellow cube in the hidden layer.

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 $\psi_1$, height $\psi_2$, time $\psi_3$ and number of filters $\phi$). As in the 1D case, the number of filters is chosen and the rest depend on the *stride* and *padding* parameters.

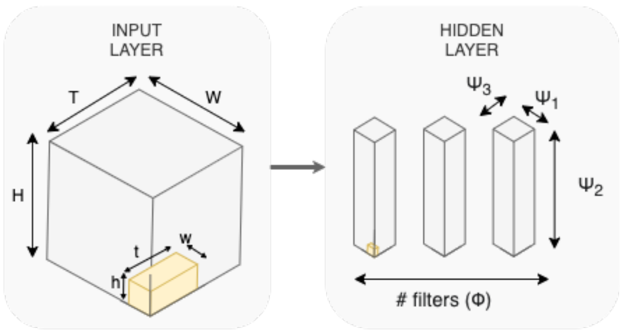


FIGURE 2.11: CNN (3D data). For example a video (width, height, time).

Convolutional layers are commonly combined with the *max-pooling* mechanism and finished by standard feedforward layers (Sec. 2.2.1.1). There are several well-known architectures listed in Sec. 2.5.1.

**Pooling** (Yamaguchi et al., 1990). This method is typically applied consequently to convolutional layers (see Sec. 2.2.1.3) 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. 2.12.

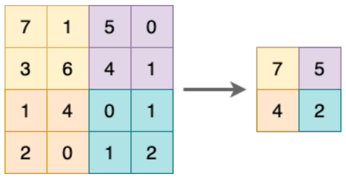


FIGURE 2.12: Principle of the max-pooling method.

### 2.2.1.4 Residual Network (skipping connections)

This method was firstly used in (He et al., 2015) in order to deal with the *vanishing gradient problem*. In case of deep structures with many hidden layers, the gradient updates propagated by the chain rule might be exponentially decreasing and thus the early layers do not update well.

This approach provides an alternative path for the backpropagation algorithm by adding connections that skip subsequent layers. There are two fundamental versions: 1/ *addition* (*ResNet*) and 2/ *concatenation* (e.g. *DenseNet*). More in (Adaloglou, 2020).

### 2.2.2 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 (Sec. 2.2.2.5 - 2.2.2.5), but first, we start with the common and nowadays mostly used RNN approach based on the idea from 1986 (Rumelhart, Hinton, and Williams, 1986). Since then, the crucial breakthroughs that have made RNNs the SoTA tool for context modeling are:

- *the backpropagation-through-time algorithm* (Elman, 1990) - capability of learning using the standard backpropagation algorithm (Sec. 2.3);
- *the LSTM cell* (Hochreiter and Schmidhuber, 1997) - capability of learning long-term dependencies (Sec. 2.2.2.1);
- *using RBM for weights initialization* (Hinton, 2002) - capability of learning for deep RNNs;
- *Transformer* (Vaswani et al., 2017) - mainly speeding up the learning process with the *attention* mechanism (Sec. 2.4.2) and positional embeddings enabling parallel computations (Sec. 2.2.2.4).

In a form of a directed graph along a (usually temporal) sequence, such a network is capable of dealing with contextual dependencies in data. Apart from the TDNN approach (Sec. 2.2.1.2), 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);

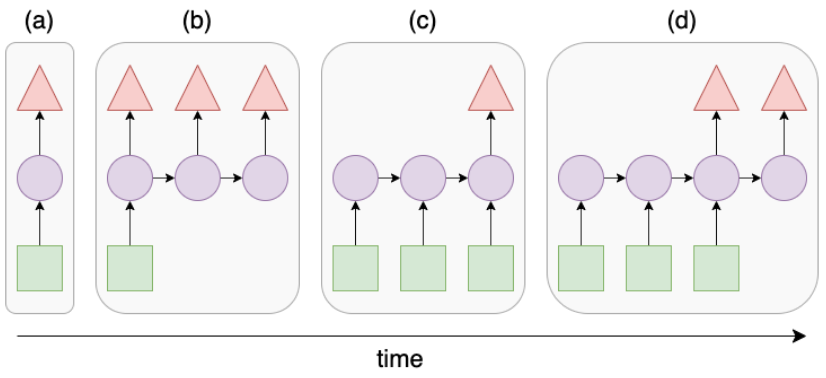


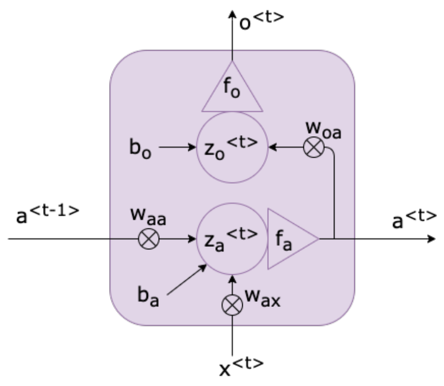
FIGURE 2.13: Sequence data - task types (Karpathy, 2015).

Assuming 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 (Amidi and Amidi, 2018). For each timestep $t$, the cell activation $a^{<t>}$ and the output $o^{<t>}$ are computed as follows:

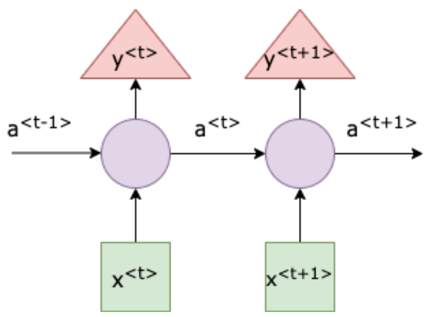
$$a^{<t>} = f_a(w_{aa} \cdot a^{<t-1>} + w_{ax} \cdot x^{<t>} + b_a) \tag{2.5}$$

$$o^{<t>} = f_o(w_{ya} \cdot a^{<t>} + b_o) \tag{2.6}$$

where $w_{aa}$, $w_{ax}$, $w_{ya}$, $b_a$, $b_o$ are parameters that are temporally shared and $f_a$, $f_o$ are chosen transfer functions (see Sec. 2.2.1.1).



(A) Default RNN - cell body.



(B) Information flow in the default RNN.

FIGURE 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 (defining the task type - see Fig. 2.13). 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. 2.2.2.1 - 2.2.2.4) in order to deal with the main drawbacks of the default version:

- The inability of learning and accessing long-term dependencies in data have been addressed by the idea of gates in the cell body (LSTM - Sec. 2.2.2.1 and GRU - Sec. 2.2.2.2).
- The default RNN version cannot consider any future input for the current state - a *bidirectional* version (BRNN - Sec. 2.2.2.3) can.
- The computation is relatively slow, which causes problems with training on large datasets. The computation time can be effectively reduced using the latest popular approach called *Transformer* (Sec. 2.2.2.4).

#### 2.2.2.1 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. 2.3) and it makes the default RNN incapable of capturing long term dependencies in the data sequence.

Originally introduced in (Hochreiter and Schmidhuber, 1997), 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. 2.5), $f_f(\cdot) = \sigma(\cdot)$;

$$g_f^{<t>} = \sigma(w_f \cdot [a^{<t-1>}, x^{<t>}] + b_f) \tag{2.7}$$

- *input gate* $g_i^{<t>}$ - decides what information is stored in the cell state using the sigmoid transfer function (Fig. 2.5), $f_i(\cdot) = \sigma(\cdot)$;

$$g_i^{<t>} = \sigma(w_i \cdot [a^{<t-1>}, x^{<t>}] + b_i) \tag{2.8}$$

- *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. 2.5), $f_c(\cdot) = \tanh(\cdot)$;

$$g_c^{<t>} = \tanh(w_c \cdot [a^{<t-1>}, x^{<t>}] + b_c) \tag{2.9}$$

- *output gate* $g_o^{<t>}$ - decides what information is sent to the output using the sigmoid transfer function (Fig. 2.5, $f_i(\cdot) = \sigma(\cdot)$) and combined with the cell state generates the activation of the cell (Eq. 2.12);

$$g_o^{<t>} = \sigma(w_o \cdot [a^{<t-1>}, x^{<t>}] + b_o) \tag{2.10}$$

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

$$c^{<t>} = g_f^{<t>} \times c^{<t-1>} + g_i^{<t>} \times g_c^{<t>} \tag{2.11}$$

$$a^{<t>} = g_o^{<t>} \times \tanh(c^{<t>}) \tag{2.12}$$

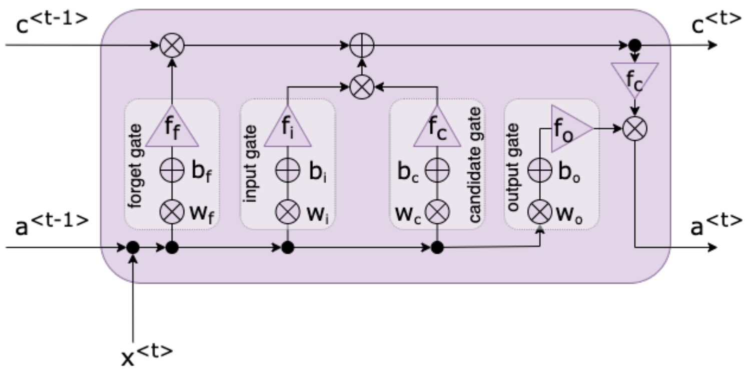


FIGURE 2.15: Long Short-Term Memory (LSTM) cell.

#### 2.2.2.2 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 (Greff et al., 2015). The most popular modified version of the general LSTM template is called GRU (Chung et al., 2014). 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. 2.5), $f_r(\cdot) = \sigma(\cdot)$;
$$g_r^{<t>} = \sigma(w_r \cdot [a^{<t-1>}, x^{<t>}] + b_r) \tag{2.13}$$
- *update gate* $g_u^{<t>}$ - the update to the activation of the cell is expressed as follows ($f_u(\cdot) = \sigma(\cdot)$):
$$g_u^{<t>} = \sigma(w_u \cdot [a^{<t-1>}, x^{<t>}] + b_u) \tag{2.14}$$
- *candidate gate* $g_c^{<t>}$ - the new candidate values are given as follows ($f_c(\cdot) = \tanh(\cdot)$):
$$g_c^{<t>} = \tanh(w_c \cdot [g_r^{<t>} \times a^{<t-1>}, x^{<t>}] + b_u) \tag{2.15}$$

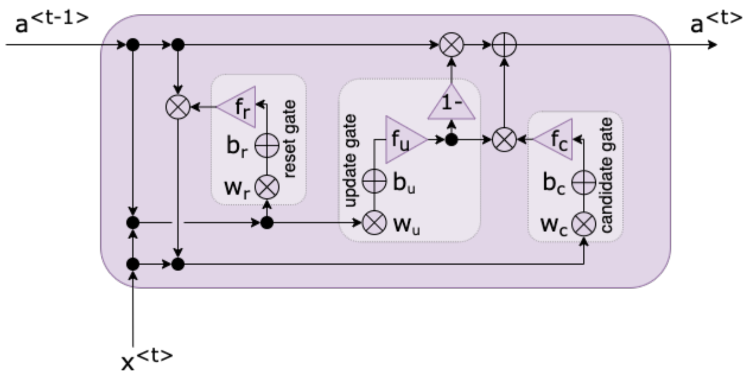


FIGURE 2.16: Gated Recurrent Unit (GRU) cell.

Finally, the new activation value is expressed as follows:
$$a^{<t>} = (1 - g_u^{<t>}) \times a^{<t-1>} + g_u^{<t>} \times g_c^{<t>} \tag{2.16}$$

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. 2.3.

Finding a way of learning deep RNN networks (based on RBM pre-training - Sec. 2.2.2.5) was the key step to make them the SoTA in sequential learning. The next significant improvements came with including the *attention* mechanism (Sec. 2.4.2), using RNNs as a part of Generative Adversarial Networks (GANs - Sec. 2.4.4) and also using the *bidirectional* architecture.

#### 2.2.2.3 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 (Schuster and Paliwal, 1997) 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.

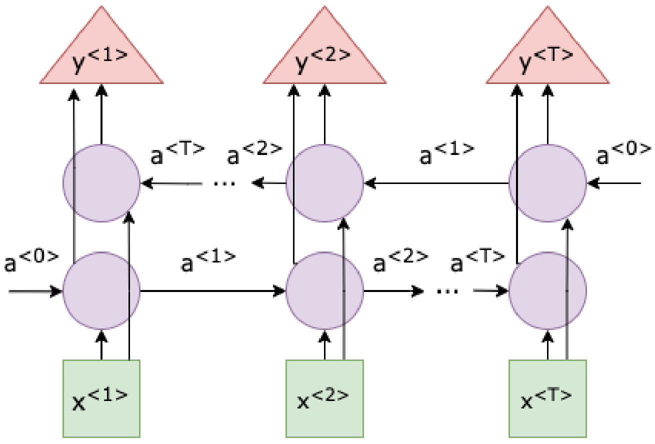


FIGURE 2.17: Bidirectional Recurrent Neural Network (BRNN) - the purple cells can be e.g. LSTM or GRU.

#### 2.2.2.4 Transformer

The approach proposed in (Vaswani et al., 2017) utilizing the *attention* mechanism (Sec. 2.4.2) and positional embeddings has directly become the model of choice (especially) for NLP problems, replacing the LSTM/GRU methods.

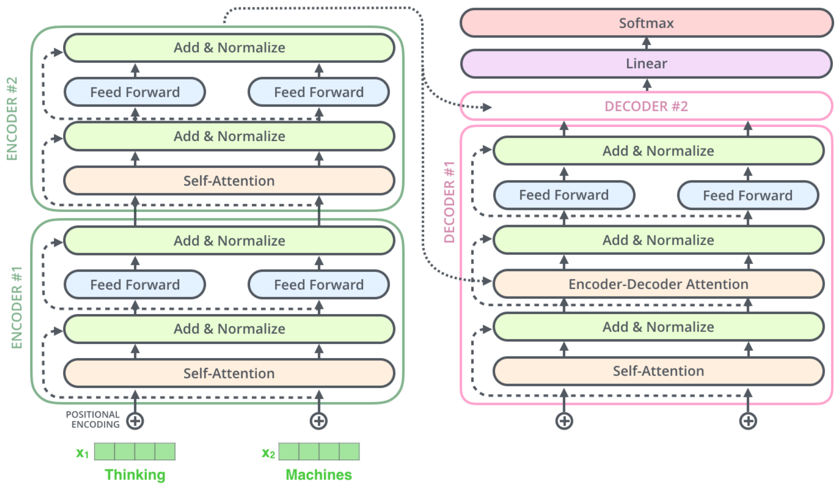


FIGURE 2.18: Transformer architecture (Alammar, 2018).

The key feature is that *Transformers* do not require the sequential data to be processed in order, which opens up parallelization possibilities and so boosts the speed of the training phase. This allows training on larger datasets than it was possible before, such as the Wikipedia Corpus (Davies, 2015). Commonly, a pre-trained model, such as BERT or GPT (see Sec. 2.5.1), is taken and then fine-tuned to a specific task. As briefly indicated in Fig. 2.18, the architecture is based on a set of encoders and decoders. A detailed explanation is provided in (Alammar, 2018). The latest upgrade including an extra convolutional layer called *Conformer* was presented in (Gulati et al., 2020).

#### 2.2.2.5 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** (Hopfield, 1982)

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.

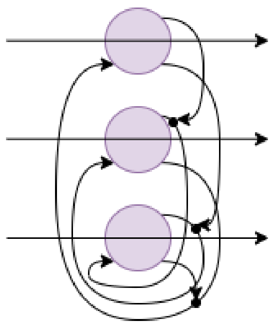


FIGURE 2.19: A Hopfield network of three units.

As long as the connections are symmetric ($w_{ij} = w_{ji}$), there is so-called global *energy* function $E$ (Eq. 2.17) and each configuration of the network is mapped to a certain energy value.

$$E = -\sum_{i<j} s_i \cdot s_j \cdot w_{ij} - \sum_i b_i \cdot s_i \tag{2.17}$$

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*: $\Delta w = s_i \cdot s_j$ (Hebb, 1949) 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.

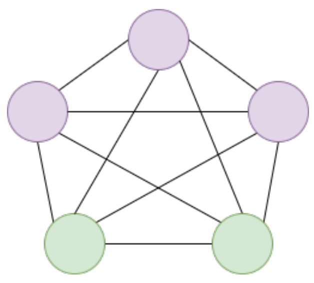
**Boltzmann Machine** (Ackley, Hinton, and Sejnowski, 1985)

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 (Hinton, 2007). 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. 2.17 ($E(v) = E$) and the probability of the *Boltzmann equilibrium* (or stationary distribution) is given as the energy relative to energies of all possible binary state vectors:

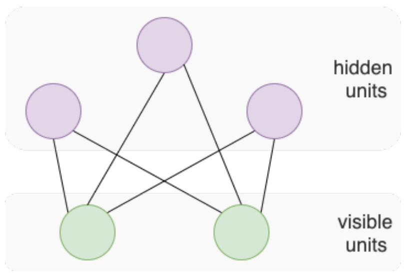
$$P(v) = \frac{e^{E(v)}}{\sum\limits_{u} e^{-E(u)}} \tag{2.18}$$

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



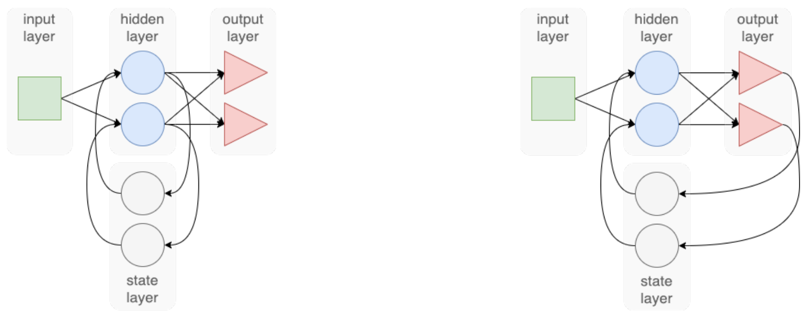
(B) Restricted Boltzmann Machine

FIGURE 2.20: A (Restricted) Boltzmann Machine example.

The restricted version - RBM (Smolensky, 1986), 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 (Hinton, 2002), 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 initialization for deep neural networks, as highest level features are typically much more useful for classification than raw data vectors.

**Elman/Jordan Network** (Elman, 1990), (Jordan, 1997)

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 (Jones, 2017). 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.



(A) Elman network. (B) Jordan network.

FIGURE 2.21: Simple recurrent networks (SRN).

Multiple state layers can possibly be subsequently added and the learning is done by the backpropagation algorithm (the BPTT version, Sec. 2.3).

**Echo State Network (ESN)** (Jaeger, 2007)

This recurrent architecture has a sparsely connected hidden layer (with typically 1% connectivity) and the hidden weights once randomly initialized remain unchanged during training. The weights of synapses connected to the output layer are the only trainable parameters. They are mostly known as *Liquid State Machines* under the field of *Reservoir Computing*.

**Independently recurrent neural network (IndRNN)** (Li et al., 2018)

Apart from to the standard composition of the RNN (Fig. 2.14b), where each cell is fully-connected within the same layer, this method is based on skipping connections (Sec. 2.2.1.4) and each cell only gets its own past state as the context information.Thus the learning is regulated to avoid the exploding/vanishing gradient problem and to capture long-term dependencies.

**Recursive Neural Networks** (Goller and Kuchler, 1996)

This network is created by applying the same set of weights recursively over a variable-sized structured input and are capable of predicting another structure or a scalar. The approach is known for its application in NLP and first was introduced to learn distributed representations.

**Neural Turing machines** (Graves, Wayne, and Danihelka, 2014)

This approach is about coupling RNNs into external memory resources, which they can interact with by attentional processes. The resulting system is end-to-end differentiable, allowing it to be trained with the gradient descent algorithm, while keeping the features of standard *Turing machine*.

**Memristive Networks** (Caravelli, Traversa, and Di Ventra, 2017)

Apart from the other methods, this one is interestingly about a physical device rather than just a theory. The *memristors* (memory resistors) are made of a thin film material with a special way of resistance tuning. Networks runable on these materials behaves like the *Hopfield* networks and they have a more interesting non-linear behaviour compared to standard circuits.

## 2.3 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 (Linnainmaa, 1970) 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. A1.

By default, the algorithm was derived for feedforward architectures (Sec. 2.2.1) and the overall procedure follows these steps:

1. *forward propagation* of a batch of samples;
$$A^{(1)} = f(W^{(1)} \cdot X + B^{(1)}) \tag{2.19}$$
$$A^{(i)} = f(W^{(i)} \cdot A^{(i-1)} + B^{(i)}) \tag{2.20}$$
$$A^{(q)} = Y = f(W^{(q)} \cdot A^{(q-1)} + B^{(q)}) \tag{2.21}$$

2. *error calculation* based on the chosen loss function $\mathcal{L}_{ff}$ ;
$$\mathcal{L}_{ff} = \frac{(U-Y)\times(U-Y)}{2} \tag{2.22}$$

3. *backpropagation* of the prediction error;
$$\Delta^{(q+1)} = (U-Y)\times f'[Z^{(q+1)}] \tag{2.23}$$
$$\Delta^{(i)} = \left[\left[W^{(i+1)}\right]^T \cdot \Delta^{(i+1)}\right] \times f'[Z^{(i)}] \tag{2.24}$$

4. *finding the optimal updates* - taken over from (Bulín, 2017); Every sample $\xi$ 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 $\xi^{th}$ column of the $\Delta^{(i)}$ matrix, which corresponds to the $\xi^{th}$ sample. Analogically, $A^{(i-1)}_{(\xi)}$ is the $\xi^{th}$ column of the activation matrix $A^{(i-1)}$ in the $(i-1)^{th}$ layer. Then we get the votes as:
$$dW^{(i)}_{(\xi)} = A^{(i-1)}_{(\xi)} \cdot \left[\Delta^{(i)}_{(\xi)}\right]^T \tag{2.25}$$
$$dB^{(i)}_{(\xi)} = \Delta^{(i)}_{(\xi)} \tag{2.26}$$

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`):
$$dW^{(i)} = \sum_{\xi}^{batch\_size} dW^{(i)}_{(\xi)} \tag{2.27}$$

The same is analogically applied to biases. The `learning_rate` value ($\mu$), 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):

$$W^{(i)<t+1>} = W^{(i)<t>} + \mu \cdot dW^{(i)<t>} \tag{2.28}$$

$$B^{(i)<t+1>} = B^{(i)<t>} + \mu \cdot dB^{(i)<t>} \tag{2.29}$$

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. 2.2.2), thanks to the method known as *backpropagation-through-time (BPTT)* from (Elman, 1990), the same procedure can be analogically applied. As shown in Fig. 2.22, the RNN layer can be unrolled over a limited number of time steps $T$ and considered as subsequent feedforward layers.

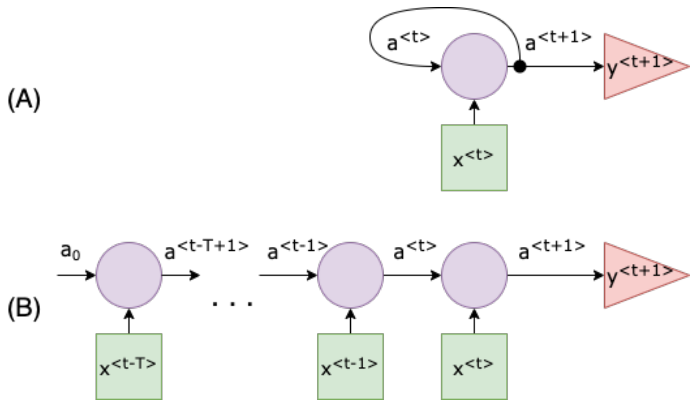


FIGURE 2.22: BPTT unfolding an RNN through time.

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

$$\mathcal{L}_{rnn} = \sum_{t=1}^{T} \mathcal{L}_{ff}^{<t>} \tag{2.30}$$

$$\frac{\partial \mathcal{L}_{rnn}}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_{rnn}^{<t>}}{\partial W}\bigg|_{t} \tag{2.31}$$

## Limitations

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.

## 2.4 Optimization Methods

Network architectures described in Sec. 2.2 together with the learning algorithm (Sec. 2.3) are the baseline in the field of ANNs. This section goes more into detail and focuses on related methods that support the baseline and help to overcome the learning limitations. Section 2.4.4 is devoted to special methods that do not belong to any of the previous categories, but still are interesting for the goals of this work.

### 2.4.1 Fundamental Techniques

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.

**Learning rate.** The learning rate ($\mu$) is a common learning hyper-parameter. It helps to converge to the solution by little steps (Fig. 2.23). The value is usually being tuned during the training.

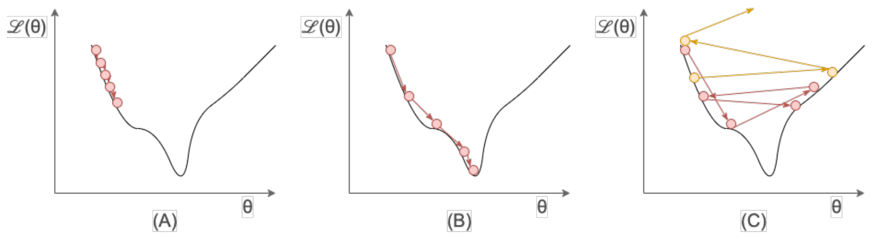


FIGURE 2.23: Learning rate: (A) too low; (B) optimal; (C) too high (red) / way too high (orange).

**Momentum.** The momentum mechanism can be added to the step of parameters update (Eq. 2.28). The purpose is to prevent oscillations and to keep traveling in the same direction along the gradient. Assuming $\alpha$ to be the momentum rate, the change of Eq. 2.28 is shown in Eq. 2.32:

$$W^{(i)<t+1>} = W^{(i)<t>} + \mu \cdot [(1-\alpha) \cdot dW^{(i)<t>} + \alpha \cdot dW^{(i)<t-1>}] \tag{2.32}$$

**Weight decay.** Sometimes the weights become too specialized to the training data and cause so-called *over-fitting*. To prevent it, this method, as one of several *regularization* techniques, makes weights decay in proportion to their sizes. The default update formula (Eq. 2.28), $\lambda$ being a decay factor, is adjusted as follows:

$$W^{(i)<t+1>} = W^{(i)<t>} + \mu \cdot (dW^{(i)<t>} - \lambda W^{(i)<t>}) \tag{2.33}$$

**Optimizers.** There are several optimization techniques for the GDA, to name some of them: *RMSProp*, *Adam*, *Nesterov*, *Adagrad*, *Adadelta*. A detailed explanation is provided in (Ruder, 2016).

**Loss functions.** The learning can be highly influenced by a correct choice of the evaluation metric and the loss function. There are many on the menu, well described in (Chollet et al., 2015).

### 2.4.2 Attention

The mechanism presented in (Bahdanau, Cho, and Bengio, 2014) allows an RNN to focus on specific parts of the input that are considered important. In Fig. 2.24, there is an illustrative example on the task of image captioning.

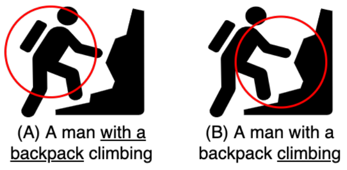

(A) A man with a backpack climbing
(B) A man with a backpack climbing

FIGURE 2.24: Illustration of the attention mechanism on the image captioning task (red circle ∼ focus).

In the background, the attention mechanism is parametrized by a simple feedforward network. The additional trainable parameters $\alpha^{<t,j>}$ express the amount of attention the output $y^{<t>}$ should pay to the activation $a^{<j>}$. Then, the context $c^{<t>}$ at time $t$ is defined as:

$$c^{<t>} = \sum_{j}^{T} \alpha^{<t,j>} \cdot a^{j} \quad \text{where} \quad \sum_{j}^{T} \alpha^{t,j} = 1 \tag{2.34}$$

For the output $y^{<t>}$ at time $t$, the attention weight responsible for context at time step $j$ is then computed as follows:

$$\alpha^{<t,j>} = \frac{\exp(\epsilon(a^{<t>}, y^{<j>}))}{\sum_{j'}^{T} \exp(\epsilon(a^{<t>}, y^{<j'>}))} \tag{2.35}$$

where $\epsilon(a^{<t>}, y^{<j>})$ is the score of the mentioned feedforward network at time $t$ and context step $j$. There are several metrics for getting the score $\epsilon$ in the original paper. The attention mechanism is considered a revolutionary idea, as it is the key ingredience of the *Transformer* approach (SoTA method on sequential data, Sec. 2.2.2.4). A well written survey of attention applications can be found in (Chaudhari et al., 2019).

### 2.4.3 Dropout

Apart from other regularization methods ($L1$ - Laplacian, $L2$ - Gaussian), the *dropout* mechanism (Hinton et al., 2012) is another method addressing the overfitting problem.

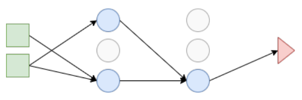


FIGURE 2.25: Example of a dropped-out network.

As illustrated on the example in Fig. 2.25, 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 testing phase then, all nodes are considered as usual.

### 2.4.4 Special Architectures and Methods

The following methods do not belong to any of the two categories (*feedforward / recurrent* - Sec. 2.2) in terms of the architecture type, purpose or learning mechanism. Their backgrounds are related to this work though.

**Generative Adversarial Network (GAN)** (Goodfellow et al., 2014)

As shown in Fig. 2.26, 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.

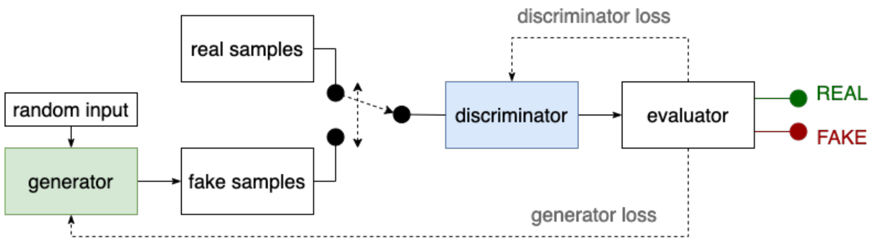
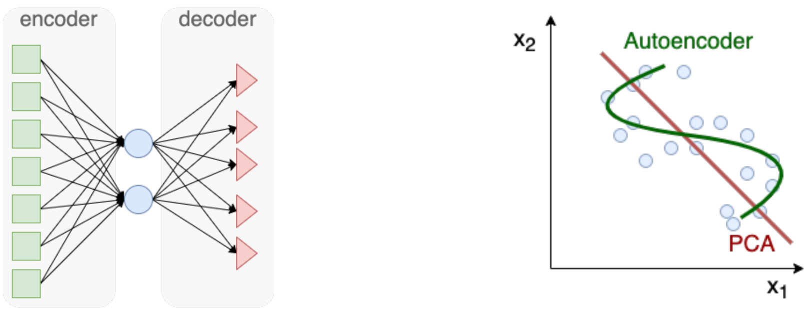


Figure 2.26: The Generative Adversarial Network concept.

**Autoencoder (AE)** (Bourlard and Kamp, 1987), (Kramer, 1991)

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.



(a) Autoencoder (a bottleneck). (b) Autoencoder vs. PCA.

Figure 2.27: Autoencoders (handling nonlinear relations).

The dimensionality of the (coded) hidden layer (also called a *bottleneck*; blue in Fig. 2.27a) 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. 2.27b).

**Self-Organizing Map (SOM)** (Kohonen, 1982)

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. 2.28). 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 (Benistant et al., 2016).

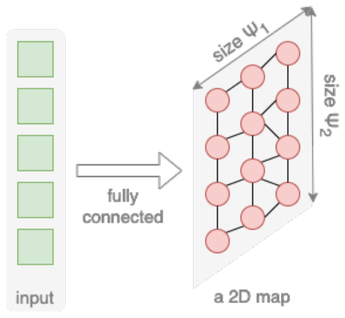


FIGURE 2.28: Kohonen's Self-Organizing Map example.

**Siamese network** (Bromley et al., 1993)

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 (Benistant et al., 2016).

## 2.5 Popular Results

Regarding methods from previous sections, this section is devoted to a summary of best results or breakthroughs generated over the years.

### 2.5.1 Pre-trained models

There are several popular architectures addressing various tasks, mostly in two domains: 1/ *NLP*, more and more using self-supervised training, and 2/ *Computer vision*, typically based on supervised learning. Those models derived from training on large datasets, can be fine-tuned for specific tasks.

#### 2.5.1.1 Natural language processing (NLP) models

By now, the *Transformer* architecture (Sec. 2.2.2.4) has been the latest hit in the field of NLP.

**GPT-3** by OpenAI (Brown et al., 2020)

Generative Pre-trained Transformer 3 is a language model with 175 billion parameters (the largest model so far). It has achieved strong performance on tasks like translation, answering questions as well as writing news articles or generating codes.

**BERT** by Google (Devlin et al., 2018)

Bidirectional Encoder Representations from Transformers allows to build question answering models in a few hours on a single GPU. It is trained on about 800 million words.

**CodeBERT** by Microsoft (Feng et al., 2020)

This model has been fine-tuned on millions of codes of six programming languages on Github. It achieved an excellent performance on searching natural language codes and generation of code documentation.

**RoBERTa** by Facebook (Liu et al., 2019)

Being a shortcut for *Robustly Optimized BERT Pretraining Approach*, this model has been trained to predict intentionally hidden sections of text. It is considered an optimised version of the BERT model trained on more data for an extended amount of time.

#### 2.5.1.2 Computer vision models

In the field of computer vision, models are commonly evaluated on the *ImageNet* dataset (Deng et al., 2009), consisting of about 14 million hand-annotated images. These methods have gradually attracted the community:

- *AlexNet* (Hinton et al., 2012) - introducing the *dropout* mechanism (Sec. 2.4.3);
- *GoogleNet/Inception* (Szegedy et al., 2014) - using pruned connections between layers to speed up the learning;
- *VGG* (Simonyan and Zisserman, 2015) - using a multiple stacked kernel of a smaller size (3x3);
- *ResNet* (He et al., 2015) - introducing the *skip-connections* mechanism (Sec. 2.2.1.4) to prevent the *vanishing gradient problem*;
- *EfficientNet - Meta Pseudo Labels* (Pham et al., 2021) - the current SotA method, using a student/teacher pair of networks.

### 2.5.2 Top Applications

In (Chatterjee, 2019), there are several fields of the human interest listed, where the theoretical methods have successfully been turned into practise:

- *self-driving cars* (Uber AI Labs at Pittsburg);
- *virtual assistants* (Siri, Amazon Alexa, Google Assistant);
- *natural language processing* (Transformer-based models);
- *visual recognition* (ImageNet-based models);
- *other domains/tasks*: healthcare, fraud detection, personalisations, colorization of black and white images, adding sounds to silent movies, game playing, election predictions, deep dreaming.

## 2.6 Architecture Search

Previous sections (2.2 - 2.5) provide a comprehensive summary of various ANN architectures, together with methods mostly addressing the shortcomings of the general learning algorithm (Sec. 2.3). In most of these cases, the network structure is fixed. This section, with respect to the objectives of this work (see Chap. 4), is devoted to algorithms for searching the network architecture in terms of neurons, synapses and their mutual interconnections. In general, these algorithms are divided into two categories: 1/ the *top-down (pruning)* methods and 2/ the *bottom-up (building)* methods.

### 2.6.1 Top-Down - Pruning Algorithms

These algorithms remove synapses from fully-connected networks, however, apart from the *dropout* optimization technique (Sec. 2.4.3), the dropped-out synapses are not turned back on for the testing phase and instead, the resulting pruned network is used for prediction. The general pruning procedure consists of these steps (corresponding to Fig. 2.29):

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.

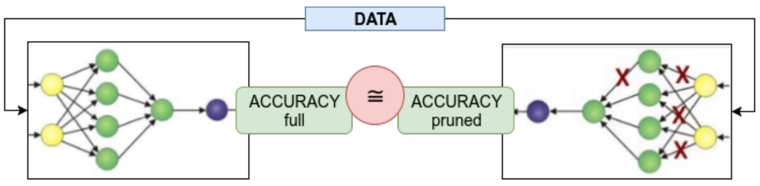


FIGURE 2.29: The principle of network pruning.

A detailed study on this topic is provided in (Bulín, 2017), 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 (Bulín, 2017), the derived algorithm is compared to these related studies:

- *Skeletonization* (Mozer and Smolensky, 1988);
- *Optimal brain damage* (LeCun, Denker, and Solla, 1990);
- *Sensitivity measure* (Karnin, 1990).

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, the resulting network is more efficient and the accuracy is kept, however, the accuracy never improve compared to the original.

### 2.6.2 Bottom-Up - Building Algorithms

These algorithms take elementary units (or blocks of units in some cases) and connect them into a structure for a specific purpose. Not all the following methods are about ANN architecture search exactly, but all of them are related to this work and have at least a similar purpose.

#### Badger Architecture (Rosa et al., 2019)

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.

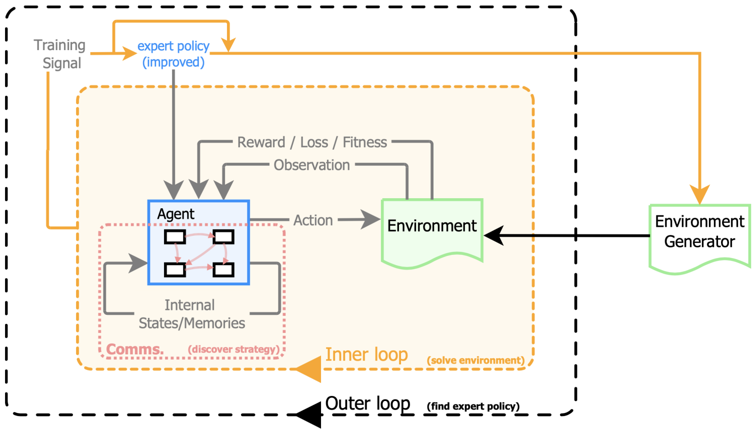


FIGURE 2.30: The inner and outer learning loops in the Badger architecture (Rosa et al., 2019).

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. 2.30. 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.

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 learned to deal with a specific problem, but rather they are learned 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.

**Neural Architecture Search (NAS)** (Zoph and Le, 2016)

Apart from the previous one, 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. 2.31) 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.

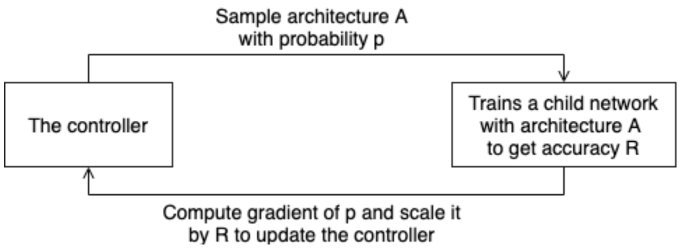

Sample architecture A with probability p
The controller
Trains a child network with architecture A to get accuracy R
Compute gradient of p and scale it by R to update the controller

FIGURE 2.31: The Neural Architecture Search principle (Zoph and Le, 2016).

An adjusted version of the algorithm determined to generate models for mobile devices is called *MnasNet* (Tan et al., 2018). This version is mainly focused on the trade-off between accuracy and inference latency. A few observations related to this work:

- The controller network is trained by reinforcement learning, using the accuracy of the generated network as a reward.
- 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. 2.2.1.4).

**Evolving Neural Networks (NEAT)** (Stanley and Miikkulainen, 2002)

This approach uses *evolutionary* optimization to construct deep learning architectures that are, based on the published results, more compex 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* (Miikkulainen et al., 2017) 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.

# Chapter 3

# 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 (Nash, 1950) 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* refers to any environment containing multiple independent units (agents) that interact with each other and with the environment (Fig. 3.1). The theory is general and applicable to many domains, for example human teams (companies), distributed software systems or communication networks.

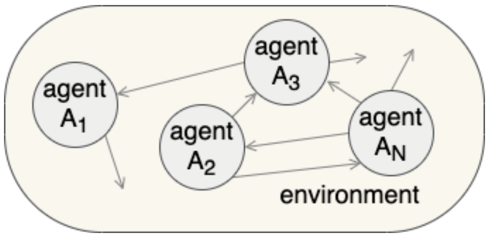


FIGURE 3.1: A multi-agent system - multiple ($N$) agents interacting with each other and with the environment.

The key feature of such systems is the *emergence principle*, a phenomenon that occurs when a system is observed to have properties its parts 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. To list the characteristics of a multi-agent system:

- Agents have a local view only, the whole system and the addressed problem are too complex for them.

- Agents are independent, self-aware and autonomous to some extend.
- Controlling the agents is decentralized, none of them is designated as the one in charge.
- By default, operations are executed asynchronously.
- *Decomposition* - complex tasks are decomposed to elementary tasks addressed by elementary components.
- *Reactivity* - each action of any entity is a reaction on its current state.

A solid overview of multi-agent technologies is presented in (Dorri, Kanhere, and Jurdak, 2018). 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.

## 3.1 Reinforcement Learning

Alongside *supervised* and *unsupervised* learning methods, *reinforcement learning* is the third fundamental ML paradigm and due to its nature it fits the best for learning in multi-agent systems. The approach is based on the Pavlov's theory of classical conditioning known as learning through association. The well-known experiment shows that using positive or negative stimuli, a dog learns to response appropriately to a given situation.

In case of artificial systems, the same theoretical principle is used to train an agent operating in an environment (Fig. 3.2). The method is defaultly derived for a single agent learning (an extention to learning multiple agents in the same environment is provided in Sec. 3.1.2).

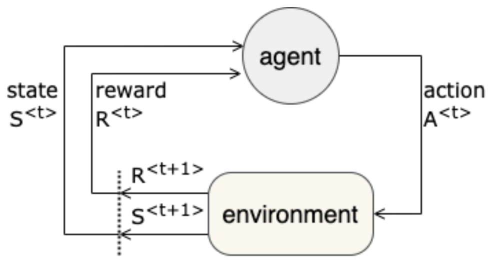


FIGURE 3.2: Reinforcement learning loop (single agent).

Apart from supervised learning, this approach does not require labeled data, nor even an explicit correction of suboptimal actions. Instead, using positive and negative feedback signals, the goal is to find a suitable strategy (action model) that would maximize the total cumulative *reward* of the agent. The task is defined by the following terms:

- *environment* - world of operation responsible for informing the agent about its current state and rewarding it for taken actions;
- *state* $S^{<t>}$ - situation of the agent in the environment at time $t$;
- *action* $A^{<t>}$ - selected agent's move at time $t$ based on state $S^{<t>}$;

- *reward* $R^{<t>}$ - feedback signal from the environment at time $t$ based on agent's reaction $A^{<t-1>}$ to state $S^{<t-1>}$;
- *policy* $\pi$ - method mapping agent's current state into action;
- *value* $V^{\pi}$ - future reward an agent would receive by following policy $\pi$.

The key feature of an agent is the policy responsible for decision making. It is commonly evolving during the learning phase and as the policy is being generated, the agent faces a dilemma of exploring unknown states while maximizing its reward at the same time. The problem of *exploration vs. exploitation trade-off* is illustrated in Fig. 3.3. By default, the agent developes ist decision making strategy (policy) from the beginning of the training process. It might seem that the best choice is to use (*exploit*) the learned experience at each time step in order to maximize the cumulative reward. However, the learned rules can easily be suboptimal (not optimal) from the global point of view, as there are more *state-action* pairs in the environment that remain unexplored. Therefore, it is important (and challenging) to carefully switch between *exploring* and *exploiting*.

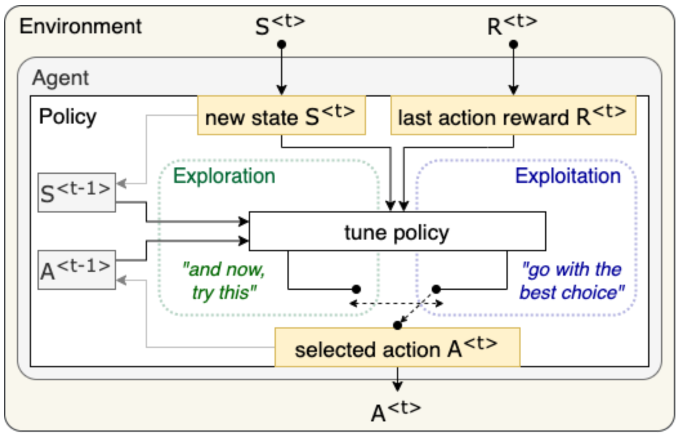


FIGURE 3.3: The exploration vs. exploitation trade-off.

**Markov Decision Process (MDP).** If each state in a sequence depends solely on the previous state and the transition from that state, it follows the Markov property and the generator of this sequence is called *MDP*,formally defined as a 4-tuple $(\varphi, \alpha_S, p_A, R_A)$, where:

- $\varphi$ is a set of states (a state space);
- $\alpha_S$ is the set of actions available from state $S$;
- $p_A(S, S') = p(S^{<t+1>} = S'|S^{<t>} = S, A^{<t>} = A)$ is the probability that action $A$ in state $S$ at time $t$ turns into state $S'$ at time $t+1$;
- $R_A(S, S')$ is the immediate reward received after transitioning from state $S$ to state $S'$ by taking action $A$.

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 *Partially Observable MDP*. The Markov property is useful especially for environments with longer episodes, as storing the complete past information and using it for making decisions become readily infeasible.

### 3.1.1 Control learning algorithms

The following methods are devoted to developing a strategy of choosing actions based on given states with the objective to maximize the cumulative reward. Recently, the dominant methods are related to using deep learning to make the agent's crucial decisions. A comprehensive list of the key literature in deep RL is well listed in (OpenAI, 2018). In the following, the most popular (model-free) methods are summarized.

**Q-Learning** (Watkins, 1989)

This model-free algorithm is based on updating *Q-values* at each time step, which denotes of choosing action $A$ given state $S$ as follows:

$$\rho(S^{<t>}, A^{<t>}) = R^{<t>} + \gamma \cdot \max_A Q(S^{<t+1>}, A) - Q(S^{<t>}, A^{<t>}) \quad (3.1)$$

$$Q(S^{<t>}, A^{<t>}) = (1-\alpha) \cdot Q(S^{<t>}, A^{<t>}) + \alpha \cdot \rho(S^{<t>}, A^{<t>}) \quad (3.2)$$

where $\rho(S^{<t>}, A^{<t>})$ is so-called temporal difference for state $S$ and action $A$ at time $t$, $\alpha$ is a learning rate saying, how much the new information overrides the old one, and $\gamma$ is a discount factor determining the importance of future rewards. Having a table of values for $A \times S$ combinations, the action $A_\star^{<t>}$ with maximal Q-value is selected given state $S^{<t>}$.

**SARSA** (Rummery and Niranjan, 1994)

This special version of *Q-Learning*, with the name derived as the acronym for the quintuple ($S^{<t>}, A^{<t>}, R^{<t>}, S^{<t+1>}, A^{<t+1>}$), is an *on-policy* algorithm for learning a *MDP* policy. Apart from using the optimal *Q-value* based on the maximal reward of available actions (default - see Eq. 3.1), this approach uses the future reward, received after taking next action $A^{<t+1>}$, to derive the temporal difference $\rho_2$:

$$\rho_2(S^{<t>}, A^{<t>}) = R^{<t>} + \gamma \cdot Q(S^{<t+1>}, A^{<t+1>}) - Q(S^{<t>}, A^{<t>}) \quad (3.3)$$

**Deep Q-Network** (Mnih et al., 2015)

The Q-Learning algorithm is simple enough and still powerful, however, it is impractical for environments with high number of states and possible actions. Even though the amount of memory and time for learning would be enormous, many states would remain unexplored. This approach uses a deep neural network to approximate the Q-value function. The state is given as the input and the Q-value for all possible actions is on the output.

**Policy Gradient Methods.** In essence, policy gradient methods update the probability distribution of actions so that actions with higher expected reward have a higher probability value for an observed state. The *gradient ascent* iterative optimization is used to learn the policy that maximizes the cumulative future reward based on the objective function $J(\theta)$:

$$J(\theta) = E[\sum_{t=0}^{T-1} R^{<t+1>}] \quad (3.4)$$

where $\theta$ is the policy parameter and $E(\cdot)$ is the expectation. In case of the *Monte Carlo* method, random samples are taken to collect a trajectory updating the policy parameter. The alternative is the *Deep Deterministic Policy Gradient (DDPG)* algorithm, which concurently learns a Q-function (exploiting the Q-Network principle) and uses it to learn the policy. The DDPG is considered to be a deep Q-learning for continuous action spaces.

**Actor-Critic Methods.** The *Policy Gradient* methods fail in case of 1/ noisy gradients causing unstable learning and 2/ gradients of high variance making the trajectories to have a zero cumulative reward. The *Actor-Critic* methods address the instability and slow convergence problems by subtracting the cumulative reward by a baseline. The *critic* estimates the value function and the *actor* updates the policy distribution. Both parts are parametrized by ANNs. The most popular approaches are:

- *Q Actor-Critic* - the *critic* uses the *Q-value* as the baseline;
- *Advantage Actor-Critic (A2C)* and *Asynchronous Advantage Actor-Critic (A3C)* - see (Mnih et al., 2016);
- *TD Actor-Critic* (Parisi et al., 2018).

**Inverse RL.** One of the recent ideas is to infer the reward function of an agent given its behaviour (Wulfmeier, Ondruska, and Posner, 2016).

**Goal-Conditioned RL.** Another active research area is adding a goal $G$ as the input to communicate a desired aim of the agent by learning so-called contextual policy $\pi(A|S,G)$ (Schaul et al., 2015).

### 3.1.2 Multiple Agents

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 (Zhang, Yang, and Basar, 2019). Regarding this work, MARL has several design choices to be considered.

**Shared vs. individual policy.** As illustrated in Fig. 3.4, all the agents can either share a single decision making system (policy $\pi$), implemented for example in (Rosa et al., 2019) - see Sec. 2.6.2, or each agent can follow its own individual strategy (option B/ in Fig. 3.4).

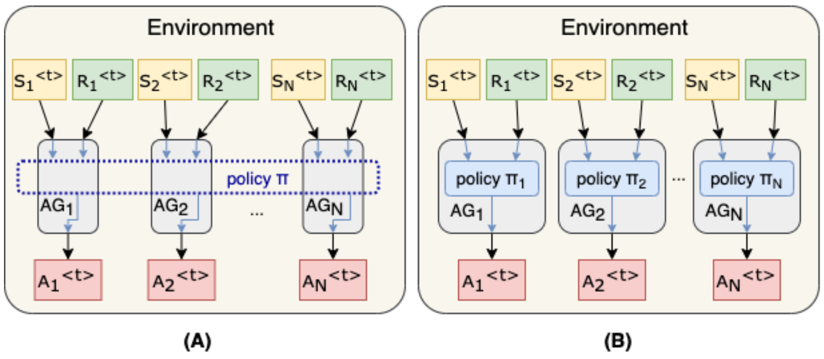

Environment
$S_1^{<t>}$ $R_1^{<t>}$ $S_2^{<t>}$ $R_2^{<t>}$ $S_N^{<t>}$ $R_N^{<t>}$
policy π
$AG_1$ $AG_2$ ... $AG_N$
$A_1^{<t>}$ $A_2^{<t>}$ $A_N^{<t>}$
(A)
Environment
$S_1^{<t>}$ $R_1^{<t>}$ $S_2^{<t>}$ $R_2^{<t>}$ $S_N^{<t>}$ $R_N^{<t>}$
policy $\pi_1$ policy $\pi_2$ ... policy $\pi_N$
$AG_1$ $AG_2$ $AG_N$
$A_1^{<t>}$ $A_2^{<t>}$ $A_N^{<t>}$
(B)

FIGURE 3.4: MARL: A/ shared vs. B/ individual policy.

**Joint vs. individual actions, states and rewards.** Fig. 3.5 sums up possible design choices regarding the definition of actions, states and rewards. In both pictures (A/ and B/ in Fig. 3.5) the actions $A_i$ can be considered and applied either individually (red) or a single joint action $A$ can be compiled and used (purple). Next, a global state $S$ of the environment is usually defined (picture A/ in the figure), however, the task can be designed in a way of multiple independent states $S_i$ for each agent individually (picture B/). Analogically, there are more possibilities of defining the reward.

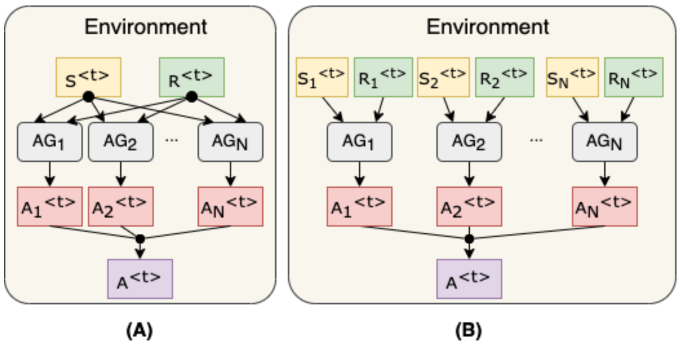


FIGURE 3.5: MARL: joint vs. individual actions, states and rewards.

**The synchronization.** Finally, as the agents cooperate (or compete) in the same environment, they interact one with each other. Then the synchronization and the order of taking their possibly individual actions as well as updating the states must highly affect the global performance. Three possible ways of synchronization are illustrated in Fig. 3.6:

(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.

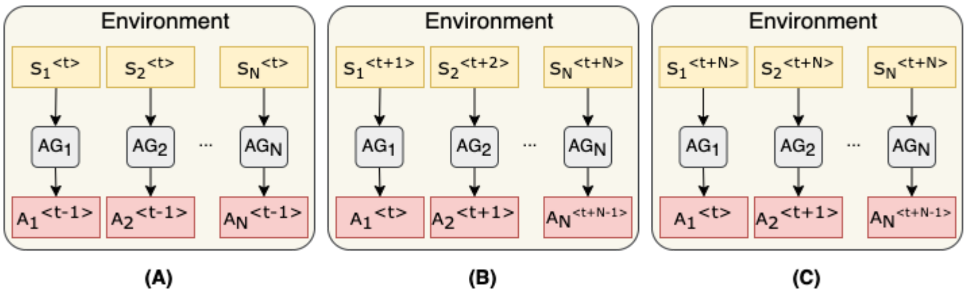


FIGURE 3.6: MARL: the synchronization problem.

# Chapter 4

# Project Objectives

This work is intended to be a preliminary report of the ongoing project assigned for the final dissertation thesis. The overall objective of the whole project is to develop an algorithm for generating a neural network tailored for given data in terms of architecture and performance metric (Fig. 4.1).

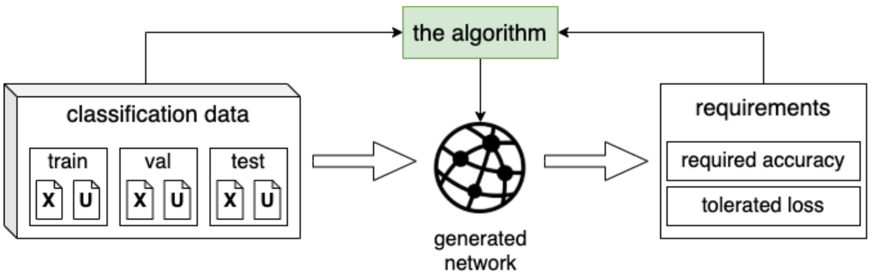


FIGURE 4.1: Global view of the project objective. The proposed algorithm (green) takes the given data and requirements as the input. Its task is to generate a network capable of classifying the data while meeting the requirements.

The network generation process will be based on the common ANN learning procedure and it will extend its features, not replace it completely. The new features will be enabled by including principles of multi-agent systems and reinforcement learning.

The following is a milestone list of the project and the checkboxes on the left indicate the current state.

**Partial goals:**

- ☑ (1) Study of related fields: *neural networks* and *multi-agent systems*;
- ☑ (2) Task composition and a baseline algorithm implementation;
- ☑ (3) Implementation of a GUI for algorithm tuning on 2D problems;
- ☑ (4) Solution of the XOR problem (with the optimal network structure);
- ☐ (5) Solution of a more challenging 2D problem (new rules derivation);
- ☐ (6) Scaling to a multidimensional problem (MNIST);
- ☐ (7) Reaching the SotA performance on MNIST;
- ☐ (8) Application to any classification (yet non-sequential) data.

# Chapter 5

# Multi-Agent based Neural Network

Previous chapters provide theoretical backgrounds, crucial breakthroughs over time and current SotA methods in machine learning areas that are closely related to this project: *neural networks* (Chap. 2) and *multi-agent systems* with *reinforcement learning* (Chap. 3). This chapter is devoted to the new ideas that are used to fullfil the goals of this project (see Chap. 4).

**Project objective.** The overall goal of the project is to develop an algorithm capable of building a neural network architecture tailored for given data. This data is restricted to represent a classification problem of $p$ (<-1,1>-normalized) samples of dimensionality $(n \times 1)$ sorted into $m$ classes. However, these numbers can originally come from any domain (vision, speech, etc.) and hence, despite the data-tailored property of the generated network, the algorithm is considered general. Scores of current SotA architectures will be taken as the baseline to compare the performance of generated networks.

**Key idea.** The development of ANNs has originally been inspired by the biological template (see Sec. 2.1). 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* 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.

Following the principles of multi-agent systems, here we can find a perfect analogy, as there are independent agents capable of primitive moves to interact with each other and with the environment. They only have a local view and the addressed problem of the system as a whole is too complex for them, however, if many agents work together, the system can *emerge* an intelligent behaviour dealing with a global task of a higher complexity.

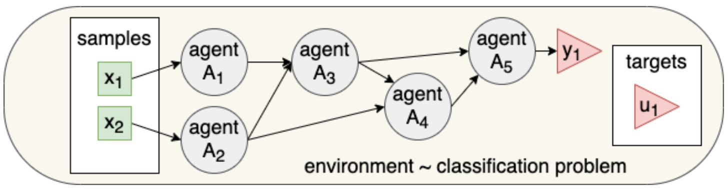


FIGURE 5.1: Neural network as a multi-agent system.

As indicated in Fig. 5.1, the idea is to consider the resulting neural network to be a multi-agent system and individual parts of the network to be the agents. The first step is to arrange the task so that the theory of multi-agent systems, and especially the control learning algorithms (see Sec. 3.1.1), could be applied to build a network architecture for the given data.

**Principles.** There are several design choices to be made in order to arrange the task and to connect the two fields - a neural network dealing with a classification problem and the reinforcement learning in a multi-agent system. The proposed methods adhere to the following principles:

- *Upgrade (not a replacement) of the SotA methods*; The proposed algorithm is supposed to extend the capabilities of the commonly used learning algorithm for ANNs (Sec. 2.3), not to replace it completely. The new method, in a certain setting and assuming unlimited computational resources, must (theoretically) be able to reproduce the SotA results for the given data.
- *Decomposition-and-emergence workflow*; The crucial idea of the workflow is to decompose the global problem as much as possible. The decomposed subproblems (e.g. classification of samples of a specific class) are supposed to be solved by individual parts of the network. The final decisions of the most elementary tasks will be based on rules of lowest-level elements (agents) and the desired behaviour is supposed to emerge from the activity of the agents in the background.
- *Deterministic elementary rules*; The mentioned elementary agents rules for selecting their moves are aimed to be kept simple and deterministic (e.g. not based on another ANN - see control learning in Sec. 3.1.1).
- *Problem-tailored (but flexible) architecture*; The function of the generated network is supposed to be problem-specific (not a complex/adaptive AI - see the related method in Sec. 2.6.2). However, if new data occurs in the given dataset, the network will be flexible enough to learn the new data. The adjustments will be local only, applied to the right places in the network purposefully.
- *Network demystification*; In general, commonly used ANNs are black-boxes nowadays. To some extend, individual parts of the generated network in this work will be observable from the outside (e.g. classification of a certain class, influence of a certain feature) and hence, targeted changes will be enabled.
- *Extendable and adjustable design choices*; The following section (5.1) determines the design choices used for initial experiments of the project. However, there are many possible settings and even ways of how to arrange the task from the ground. Hence, it is expected that some of the design choices might be adjusted with the ongoing research and more experiments. At the same time, these choices must be extendable for new features (e.g. adding the possibility of handling context-, time- as well as position-, dependent data).

## 5.1 Proposed method

As stated above, the following design choices have been determined for the initial experiments (Sec. 5.2), however, might be adjusted with the ongoing research. The overall view of the task is shown in Fig. 5.2.

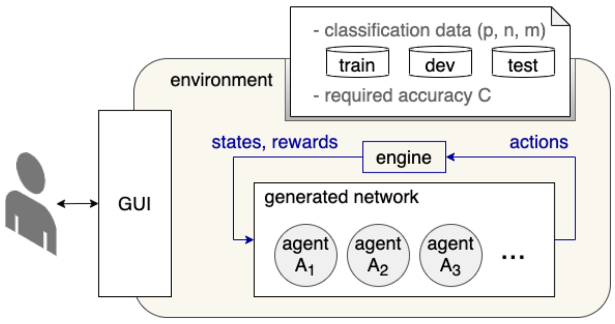


FIGURE 5.2: Global view on the proposed method.

Initially, the network has no agents and the environment has an empty slot waiting for a classification problem, defined by data and a classification accuracy the network is required to reach, to be plugged in. The *engine* (Sec. 5.1.1) is the executive tool of the environment. The *network* (Sec. 5.1.2) is generated from zero and its initialization depends on the problem specification. The graphical user interface (Sec. 5.1.3) allows the user to interact with the environment and to observe the network generation process.

### 5.1.1 Engine

The engine operates with the provided data in order to determine states and rewards for individual agents at each epoch $t$ of the generation process. The algorithm shown in Fig. 5.3 is parametrized by the maximal number of generation epochs $T$ and required classification accuracy of the generated network $C$ (defined with the problem). During the first pass ($t = 0$), all agents are in their *initial* states, take no actions and recieve new states only. The condition $Z_1$ is defined as follows:

$$Z_1 = acc^{<t>} >= C \wedge S_i^{<t>} == S_{optimal} \quad \forall\, i, ..., N \tag{5.1}$$

where, assuming epoch $t$, $acc^{<t>}$ is the classification accuracy of the network (validation data), $S_i^{<t>}$ is the state of $i^{th}$ agent and $N$ is the number of agents. The loop in the yellow box in Fig. 5.3 is processed asynchronously and the agents are ordered randomly. The state determination is based on the *SGD* algorithm, specifically on the loss function - both the global one as well as the error of each agent individually (see Sec. 5.1.2 and Tab. 5.1).

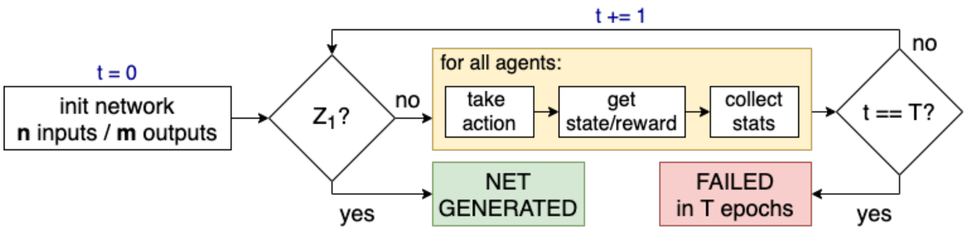


FIGURE 5.3: Network generation algorithm.

### 5.1.2 Agents

There are two types of agents: *neurons* and *synapses*. As the network is initialized, $m$ neurons are generated (representing $m$ classes) and each of them gets $n$ incoming synapses (one for each feature of the $n$-dimensional data). Then the generation process (Fig. 5.3) is started and the only responsibility of the agents at every epoch $t$ is to choose an action based on the given state and the reward from the environment (engine). Each agent has its own independent policy that is pretty basic currently and therefore it is a subject to optimize with the ongoing research of the project.

The default rules used for the initial experiments (Sec. 5.2) are summarized in Tab 5.1. In this default case, no reward from the environment is considered and each state has only one strictly determined action to be performed. Hence, the crucial part is to determine the state of each agent correctly.

| *agent* | state: | *initial* | *useless* | *missing* | *incapable* | *tuning low* | *tuning high* | *optimal* |
|---|---|---|---|---|---|---|---|---|
| neuron | condition: | $t = 0$ | Eq. 5.2 | X | Fig. 5.4 | $\Delta w > 0$ | $\Delta w < 0$ | Eq. 5.5 |
| | action: | - | remove self | X | add neuron in front | increase weight | decrease bias | - |
| synapse | condition: | $t = 0$ | Eq. 5.2 | Eq. 5.3 | Fig. 5.4 | $\Delta b > 0$ | $\Delta b < 0$ | Eq. 5.5 |
| | action: | - | remove self | return | - | increase weight | decrease weight | - |

TABLE 5.1: Default agents policies: one action per state (no reward considered) with conditions to get into corresponding states. The standard learning algorithm (Sec. 2.3) considers gray columns only.

Determination of the state is based on the backpropagation algorithm (Sec. 2.3), however, apart from the common approach (gray columns in Tab. 5.1), this method allows an agent to use information about the error on other agents as well as over the past epochs to aquire its own state. Based on simple rules, agents are allowed to choose out of more actions than just tuning the weights or biases. In the following equations, $\Delta w_i^{<t>}$ refers to the suggestion of the SGD algorithm to tune weight $w$ of the $i^{th}$ agent (synapse) at time $t$ ($|\Delta b_i^{<t>}$ works analogically for biases of agents-neurons).

- *initial*; No agent's action is required and the state is undoubtedly changed during the first iteration.
- *useless*; An agent is considered useless if its weight (bias) is not being touched during the training (Bulín, 2017) and the consequent action removes the agent from the network (in case of neurons, also their connections are removed). For initial experiments (Sec. 5.2): $\epsilon_1 = 20$ and $\epsilon_2 = 0.1$.

$$Z_{useless}^{<t>} = t > \epsilon_1 \wedge \sum_{\tau=0}^{t} |\Delta w^{<\tau>}| < \epsilon_2 \qquad (5.2)$$

- *missing*; As long as a removed neuron can be replaced by a new one, only synapses can aquire the *missing* state. A removed synapse is not used in the network, however, it holds its *useless* state and can be returned if its target neuron $S_N$ gets into the *incapable* state.

$$Z_{missing}^{<t>} = S^{<t-1>} == S_{useless} \wedge S_N^{<t>} = S_{incapable} \tag{5.3}$$

- *incapable*; A neuron is considered incapable of dealing with its task if the tuning suggestions of its bias $\Delta b$ as well as $\Delta w_i$ of all incoming synapses do not converge to zero over last $\epsilon_3$ epochs (set to $\epsilon_3 = 30$ for experiments in Sec. 5.2). Then an in-front neuron is added as depicted in Fig. 5.4, incoming synapses are copied and a single output is connected to the calling neuron in need. Hence the feedforward flow in the network is guaranteed.

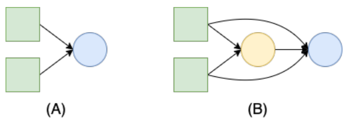


FIGURE 5.4: Adding a new neuron (yellow) in-front of the calling one (blue).

- *tuning*; If the agent does not meet conditions of any other state, it is in the *tuning* state. Depending on the polarity of the SGD suggestions $\Delta b$ (resp. $\Delta w$), *tuning-low* and *tuning-high* states are distinguished. The parameter update is computed with a learning rate ($\mu = 0.03$ in the experiments - Sec. 5.2) as follows (and analogically for $b/db$):

$$w^{<t+1>} = w^{<t>} + \mu \cdot dw^{<t>} \tag{5.4}$$

- *optimal*; When all the agents reach the *optimal* state and the required classification accuracy $C$ is reached, the generated network is considered optimal for the given data (Eq. 5.1). The agent is in the *optimal* state, if the suggested change obtained from the SGD is below a threshold $\epsilon_3$ ($\epsilon_4 = 0.01$ for experiments in Sec. 5.2). The optimal state can be left again due to interactions of other agents (see Fig. 5.5).

$$Z_{optimal}^{<t>} = |dw^{<t-1>}| < \epsilon_3 \tag{5.5}$$

The behaviour of the agents can be described by the finite state machine illustrated in Fig. 5.5. Their goal is to reach the *optimal* state by tuning their parameters so that their errors are driven to zero. Eventually, neurons are allowed to call for adding a new neuron if the situation seems unsolvable. All agents can be removed from the network if they feel useless and removed synapses are allowed to return back to their places if needed.

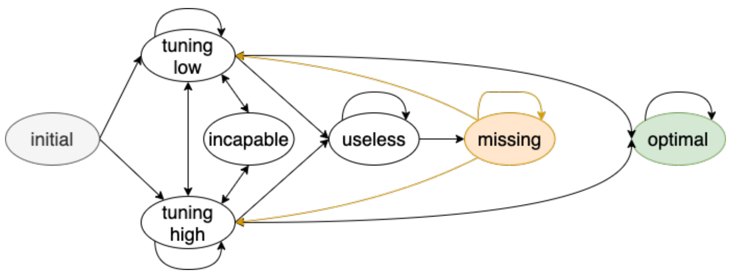
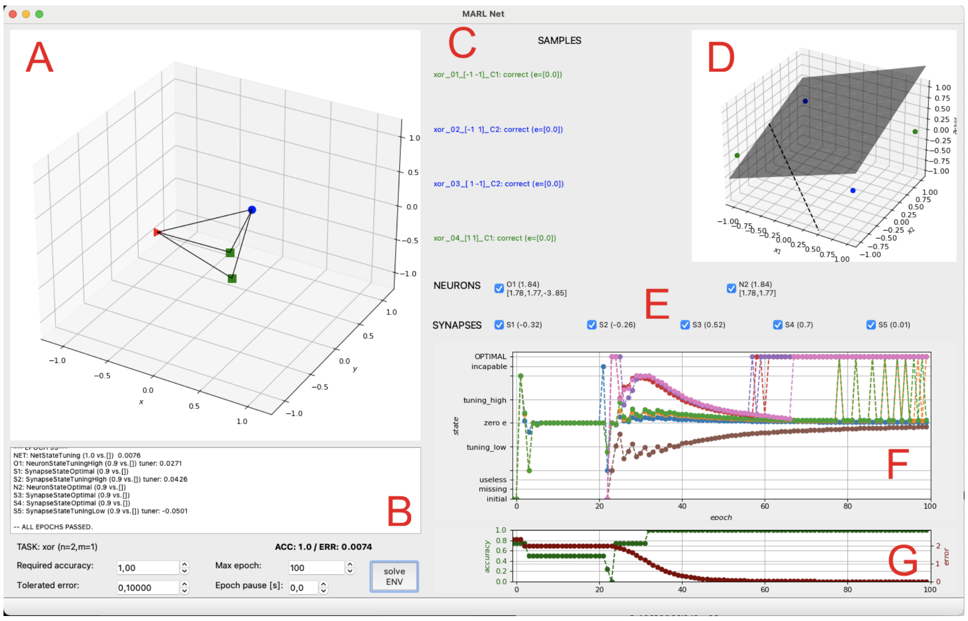


FIGURE 5.5: Agents states and allowed transitions. Orange parts are reachable by synapses only (not by neurons).

### 5.1.3 Graphical user interface (v1)

The purpose of the user interface is especially to help derive the algorithm principles on low-dimensional classification problems. The version shown in Fig. 5.6 allows to observe the network generation process for 2D data.



xor _01_[-1 -1]_C1: correct (e=[0.0])
xor _02_[-1 1]_C2: correct (e=[0.0])
xor _03_[ 1 -1]_C2: correct (e=[0.0])
xor _04_[1 1]_C1: correct (e=[0.0])

NEURONS: ☑ O1 (1.84) [1.78,1.77,-3.85] ☑ N2 (1.84) [1.78,1.77]
SYNAPSES: ☑ S1 (-0.32) ☑ S2 (-0.26) ☑ S3 (0.52) ☑ S4 (0.7) ☑ S5 (0.01)

```
NET: NetStateTuning (1.0 vs.[])  0.0076
O1: NeuronStateTuningHigh (0.9 vs.[]) tuner: 0.0271
S1: SynapseStateOptimal (0.9 vs.[])
S2: SynapseStateTuningHigh (0.9 vs.[]) tuner: 0.0426
N2: NeuronStateOptimal (0.9 vs.[])
S3: SynapseStateOptimal (0.9 vs.[])
S4: SynapseStateOptimal (0.9 vs.[])
S5: SynapseStateTuningLow (0.9 vs.[]) tuner: -0.0501

-- ALL EPOCHS PASSED.
```

TASK: xor (n=2,m=1) ACC: 1.0 / ERR: 0.0074
Required accuracy: 1,00 Max epoch: 100
Tolerated error: 0,10000 Epoch pause [s]: 0,0

FIGURE 5.6: Graphical user interface (v1) for interaction.

There are six independent areas in the GUI (marked A-G in Fig. 5.6) capable of the following:

(A) 3D projection of the generated network; Positions of the nodes are generated randomly, however, the colors and shapes match the notation in this work (green square $\sim$ input; blue circle $\sim$ hidden; red triangle $\sim$ output).

(B) Process log and hyperparameters settings;

(C) Samples information (correctly classified/misclassified, sample error);

(D) Visualization of individual neurons and the samples in a 3D space;

(E) Parameters of the neurons (biases and incoming weights) and synapses (weights) with the option to remove individual agents from the figures;

(F) Evolution of the states of individual agents over time;

(G) Evolution of the classification accuracy and the loss funcion over time.

## 5.2 Initial Experiments

The following 2-dimensional classification problems are used to verify the default agent rules (the RL policy) presented in Sec. 5.1 and to help derive more sophisticated rules in the future. These toy problems help to find principles and to prove concepts with the hope of scaling on real multidimensional tasks in the future.

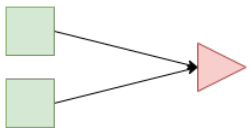
### 5.2.1 A basic 2D problem

This elementary (linearly separable) task defined by Tab. 5.2 can be solved by a network of a single output neuron.

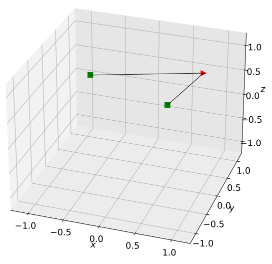
| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 1 | 1 |
| -0.5 | 0 | 1 |
| 0.5 | 0 | 0 |
| 0 | -1 | 0 |

TABLE 5.2: A basic 2D problem definition.

The environment consists of three agents (the neuron and two incoming synapses). The goal was to use the proposed algorithm to reach their *optimal* states ending up with the expected minimal network structure (Fig. 5.7a).



(A) Known minimal architecture.



(B) Generated architecture.

FIGURE 5.7: The 2D basic problem minimal architecture.

As shown in Fig. 5.8, the generation process finished after 85 epochs with the expected architecture (Fig. 5.7b). In case of *tuning* states, Fig. 5.8 indicates the decreasing error value driven to zero. The neuron (blue) reached the *optimal* state after 6 epochs, followed by the synapses (green ∼ epoch 40 and orange ∼ epoch 85). Then the process ended and the neuron was perfectly aligned in the 2D space to separate the two classes (Tab. 5.2) with a negligible loss value - (see Eq. 5.5) and the maximal classification accuracy.

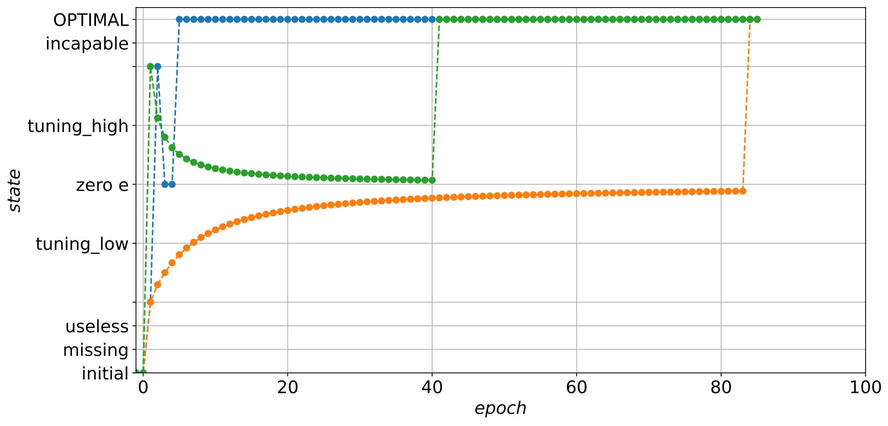


FIGURE 5.8: Evolution of the states over epochs (2D basic).

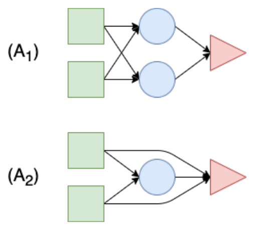
### 5.2.2 The XOR problem

This well-known problem (Tab. 5.3) requires an extra hidden neuron to be solved as the classes are linearly inseparable in the 2D space.

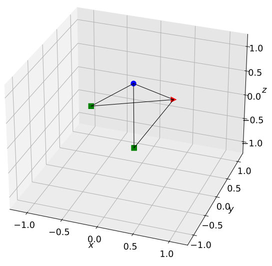
| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

TABLE 5.3: XOR function.

There are two known minimal network architectures (Fig. 5.9a) that are considered minimal for this task. Apart from the previous problem, here we need to add one extra neuron into the initial network. The goal was to end up with one of the expected architectures using the proposed algorithm.



(A) Known minimal architectures.



(B) Generated architecture.

FIGURE 5.9: The XOR problem minimal architecture.

In each out of 50 experimental runs, the algorithm finished with the expected architecture $A_2$ (see Fig. 5.9). In Fig. 5.10 there is the evolution of agents states in one selected observation. Following the basic rules defined in Sec. 5.1.2, the key neuron was added after 35 epochs, when the first neuron was declared *incapable* of solving the task. Since then, all the agents have been one by one driven to the *optimal* state. The generation process finished after 108 epochs with a minimal loss value (see Eq. 5.5) and the maximal classification accuracy produced by a network of the desired architecture.

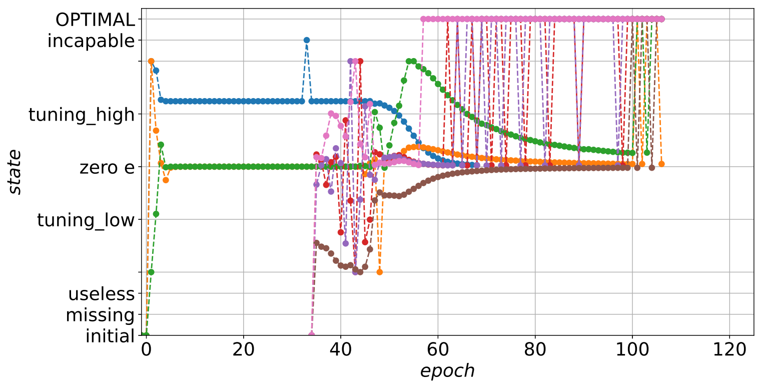


FIGURE 5.10: Evolution of the states over time (XOR).

# Chapter 6

# Discussion

Building a neural network from zero is a challenging task, but the motivation for working on it is huge and the proposed method is solid. The mainstream way of using ANNs is basically a trial and error procedure as the red block in Fig. 6.1 consists of tuning hyper-parameters, data augmentation, deployment of pre-trained models and an engagement of faster computers.

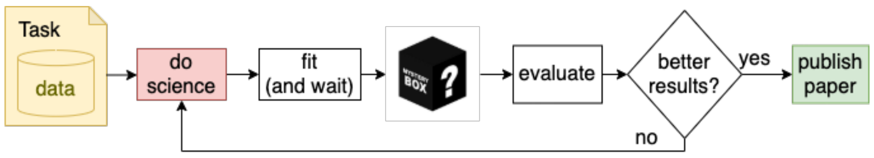


FIGURE 6.1: Mainstream way of using ANNs nowadays.

From time to time, a new optimization method (see Sec. 2.4) appears and helps fight the limitations of the standard learning algorithm (Sec. 2.3), usually by a kind of interference to network structures, for instance the *skipping connections* method (Sec. 2.2.1.4), the *dropout* method (Sec. 2.4.3) or even revolutionary architectures (Sec. 2.2). However, no matter how much these methods raise the performance, the nature of the procedure in Fig. 6.1 is mostly kept and no targeted changes can be applied to the trained models. This makes us feel that despite some extraordinary results, we still might not be exploiting the full potential of neural principles. Interestingly, this assumption is supported by conclusions of pruning methods (Sec. 2.6.1) as well. Although they are not capable of improving the performance, they show that a significant amount (up to 90%) of synapses in (still widely used) fully-connected networks is completely redundant.

**Proposed method summed up.** Apart from using the brute force in terms of unnecessarily complicated architectures, larger datasets and increasingly powerful computers, this work suggests to start the other way around and to focus on building architectures tailored for given data. The proposed method (Chap 5) elaborates on the multi-agent systems theory (Chap. 3). Individual parts of the network are treated as agents directed by reinforcement learning to cooperate on the given task. Despite the elementary nature of individual agents following deterministic rules, multiple agents working together as a whole are believed to be capable of *emerging* the desired (sophisticated) behaviour of the generated classifier.

**State of the project.** The backbone of the algorithm has been already implemented (Sec. 5.1) and initial experiments have been performed (Sec. 5.2).

The first version of the implementation is planned to be extended in the following research. The crucial part of the whole work is the behaviour of the agents controlled by traceable rules. The derivation of these rules has been launched in a 2D space, supported by a user interface (Sec. 5.1.3), and applied on two basic experiments. Both of them finished successfully with expected results and proved the end-2-end preparedness of the implementation for further extention.

**Possible pitfalls.** The key step of the project will be to prove the *scalability* of the method on more complicated (multidimensional) tasks. Next, the experiments will have to focus on preventing the *overfitting* problem carefully examined with *dev* and *test* data sets. Finally, the design of the basic rules (see Sec. 5.1) will be crucial to meet the goals of the work, while keeping the generated network as *observable* as possible.

**Research directions.** There are many unexplored directions and design options that are worth experimenting with:

- *Definition of a state/reward*; The baseline design of the RL task is a subject to think about. By now, the states are defined agent-wise (see Fig. 5.5) instead of using the usual state of the environment. Moreover, there are several options how to use the loss function to define the state and eventually to combine it with the reward function.
- *Definition of agents*; With the ongoing research, we might conclude that considering individual neurons and synapses (as defaultly set in Sec. 5.1) to be the agents is incalculable. Then we could possibly experiment with blocks of neurons, for instance forming structures that we today know as layers or more complicated cells (e.g. LSTM - Sec. 2.2.2.1, which is actually a block of neurons as well).
- *Agents policies*; The decision making policy can be common and shared by the agents (like in the related method in Sec. 2.6.2) or it can be unique for each agent individually. To make a decision, each agent can use its own as well as others information in terms of states, actions, rewards and loss values from the beginning of the training. Moreover, each agent can reason about its contribution to the classification of individual samples (e.g. samples of a specific class) and hence helps prepare the network for future targeted adjustments for new samples.
- *Asynchronous processing*; In the loop in (Fig. 5.3), the agents are processed one by one and their order is set randomly. Finding a way of sorting, respectively of a synchronization, of the agents is another important research direction.
- *Network building (growing) moves*; Fig. 5.4 shows an option how to add a new neuron into a network. However, the new neuron could be placed besides the *calling* neuron (not in front of it) and similarly, further building strategies could be implemented.
- *Data flow direction*; This work is focused on non-sequential classification data and therefore the generated architecture is kept feedforward. However, including loops in the network would significantly extend the scope of possible experiments.

## 6.1 Comparison to related methods

There are two groups of methods addressing the architecture search task - the top-down (*pruning*) algorithms and the bottom-up (*building*) algorithms (Sec. 2.6). The pruned network of top-down methods is generally limited by the initial choice of the network architecture and although these methods help in terms of parameters reduction, they are not expected to improve the performance in terms of classification accuracy. The proposed method of this work would belong to the *bottom-up - building* category and hence it is fair to compare it to methods from Sec. 2.6.2.

**Comparison to Badger.** The closest approach on the market is called *Badger* (Rosa et al., 2019). As well as in this work, they design networks of multiple agents trained with reinforcement learning and their project is also at the stage of prooving the scalability. Apart from this work:

- Related to the authors' long-term goal of building the general AI, the *Badger* architecture is aimed to be adaptive to new environments (problems). In this work, although generated networks are expected to be flexible enough to learn new samples, each will be tailored for one specific classification problem only.
- In the *Badger* architecture, the agents (called experts) share a universal policy for decision making that is based on a trained neural network. This work tends to avoid unobservable decision making systems and focuses on deterministic rules at the lowest level.

**Comparison to architecture search algorithms.** As well as in this work, these methods generate a network architecture that deals with a specific task. However, unlike this work, they are based on searching optimal combination of hyper-parameters and as the space of those combinations is enormous, they need another (hardly observable) ML method to deal with it. In case of NAS (Zoph and Le, 2016), an RNN is trained by reinforcement learning to produce a string specifying the generated classifier in terms of hyper-parameters. The NEAT approach (Stanley and Miikkulainen, 2002) uses evolutionary algorithms to search the space of hyper-parameters.

## 6.2 Outlook

The next steps have been summarized in Chap. 4. The XOR experiment (Sec. 5.2.2) proved the ability of the algorithm to add one needed neuron to solve a linearly inseparable task. The very next step will be to show the ability to add more (but not too many) neurons to solve a more complicated, but still a 2D (and so imaginable and drawable) task. Then, the key step will be to make the method *scalable* to multidimensional tasks, where firstly it will be widely elaborated on the MNIST dataset.

In case of reaching the objectives of this work, a long-term outlook could consist of an extention to context-dependent data by implementing, besides new eventual ideas, the known CNN/RNN principles.

# Chapter 7

# Conclusion

This work elaborates on a new method for working with the neural principles in machine learning that could be an alternative to the mainstream in deep learning. In contrast of training unnecessarily complicated structures that are commonly assembled and tuned by an expertised network architect, the objective of the new method is to generate network architectures tailored for given datasets. The algorithm is designed to work for non-sequential labeled classification data and its purpose is to produce a network that meets the requirements in terms of classification accuracy and tolerated loss.

The timeline of significant breakthroughs in the field of ANNs over the years was studied first, as the main principle of the new method is to exploit the successful ideas, to fit them into the new concept and to extend their capabilities. The backbone is represented by the well-known *backpropagation* algorithm, however, it only supports the decision making in the network instead of taking the full control over the learning.

The network generation process is based on the theory of *multi-agent systems* with *reinforcement learning* and especially on the convenient analogy between a neural network and a multi-agent system. Individual parts (by default neurons and synapses), as well as the agents, are super-elementary and have a local view only, with no idea about the global complex problem they work on together as a whole (the network). The key idea of the new method is to elaborate on the *emergence* phenomenon of multi-agent systems and to use it for network generation, while its parts will be of a primitive nature based on simple deterministic rules. These rules are expected to lead the algorithm to produce a tailored and (to the extend possible) observable network allowed to be, apart from today's trained models, purposefully modified in the future.

The main loop of the algorithm has already been implemented and default rules have been proposed. The baseline version has been sucessfully applied to two classification problems in 2D with expected results. The rules are aimed to be further elaborated on more sophisticated 2D problems with the help of the developed graphical interface. Next, the approach will be applied to a multidimensional classification problem in order to prove its scalability and generality. Regarding various design options for the proposed combination of the two ML fields, neural networks and multi-agent systems, this work opens several unexplored research directions.

# Bibliography

[1] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[2] Donald O. Hebb. *The organization of behavior: A neuropsychological theory.* New York: Wiley, June 1949. ISBN: 0-8058-4300-0.

[3] J. F. Nash. “Equilibrium Points in N-Person Games”. In: *Proceedings of the National Academy of Sciences of the United States of America* 36.48-49 (1950).

[4] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

[5] David H. Hubel and Torsten N. Wiesel. “Receptive Fields of Single Neurons in the Cat’s Striate Cortex”. In: *Journal of Physiology* 148 (1959), pp. 574–591.

[6] Stanford University. *Adaptive "adaline" Neuron Using Chemical "memistors.".* Stanford Electronics Laboratories. Solid State Electronics Laboratory et al., 1960. URL: https://books.google.cz/books?id=Yc4EAAAAIAAJ.

[7] M. Minsky and S. Papert. *Perceptrons.* Cambridge, MA: MIT Press, 1969.

[8] Seppo Linnainmaa. “The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors”. MA thesis. University of Helsinki, 1970.

[9] P. J. Werbos. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”. PhD thesis. Harvard University, 1974.

[10] Kunihiko Fukushima. “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”. In: *Biological Cybernetics* 36 (1980), pp. 193–202.

[11] J. J. Hopfield. “Neural networks and physical systems with emergent collective computational abilities”. In: *Proceedings of the National Academy of Sciences of the United States of America* 79.8 (Apr. 1982), pp. 2554–2558. ISSN: 0027-8424. URL: http://view.ncbi.nlm.nih.gov/pubmed/6953413].

[12] Teuvo Kohonen. “Self-organized formation of topologically correct feature maps”. In: *Biological Cybernetics* 43.1 (Jan. 1982), pp. 59–69. ISSN: 0340-1200. DOI: 10.1007/BF00337288. URL: http://dx.doi.org/10.1007/BF00337288.

[13] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. “A Learning Algorithm for Boltzmann Machines”. In: *Cognitive Science* 9 (1985), pp. 147–169.

[14] G. E. Hinton and T. Sejnowski. “Learning and relearning in Boltzmann machines”. In: *Parallel distributed processing: Explorations in the microstructure of cognition.* Cambridge, MA: MIT Press, 1986, pp. 282–317–.

[15] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-propagating Errors”. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: `10.1038/323533a0`. URL: `http://www.nature.com/articles/323533a0`.

[16] P. Smolensky. “Information processing in dynamical systems: Foundations of harmony theory”. In: *Parallel distributed processing: Explorations in the microstructure of cognition.* Cambridge, MA: MIT Press, 1986, pp. 194–281–.

[17] H. Bourlard and Y. Kamp. *Auto-Association by Multilayer Perceptrons and Singular Value Decomposition.* Manuscript M217. Brussels, Belgium: Philips Research Laboratory, 1987.

[18] Alex Waibel et al. *Phoneme Recognition Using Time-Delay Neural Networks.* Tech. rep. TR-I-0006. Kyoto, Japan: Advanced Telecommunication Research Institute, International Interpreting Telephony Research Laboratories, 1987.

[19] Michael Mozer and Paul Smolensky. “Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment”. In: *Advances in Neural Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988].* Ed. by David S. Touretzky. Morgan Kaufmann, 1988, pp. 107–115. URL: `http://papers.nips.cc/paper/119-skeletonization-a-technique-for-trimming-the-fat-from-a-network-via-relevance-assessment`.

[20] K. Hornik, M. Stinchcombe, and H. White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366.

[21] Y. LeCun et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1 (1989), pp. 541–551.

[22] C. J. C. H. Watkins. “Learning from Delayed Rewards”. PhD thesis. King’s College, Oxford, 1989.

[23] Jeffrey L. Elman. “Finding Structure in Time”. In: *Cognitive Science* 14 (1990), pp. 179–211.

[24] E. D. Karnin. “A simple procedure for pruning back-propagation trained neural networks”. In: *IEEE Transactions on Neural Networks* 1.2 (1990), pp. 239–242. DOI: `10.1109/72.80236`.

[25] Yann LeCun, John S. Denker, and Sara A. Solla. “Optimal Brain Damage”. In: (1990), pp. 598–605. URL: `https://papers.nips.cc/paper/250-optimal-brain-damage`.

[26] K. S. Narendra and K. Parthasarathy. “Identification and control of dynamical systems using neural networks”. In: *IEEE Transactions on Neural Networks* 1.1 (1990), pp. 4–27. DOI: `10.1109/72.80202`.

[27] Kouichi Yamaguchi et al. “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, 1990. URL: `http://www.isca-speech.org/archive/icslp\_1990/i90\_1077.html`.

[28] M. Kramer. “Nonlinear Principal Component Analysis Using Autoassociative Neural Networks”. In: *AIChE Journal* 37 (1991), pp. 233–243.

[29] Jane Bromley et al. “Signature Verification Using A "Siamese" Time Delay Neural Network.” In: *IJPRAI* 7.4 (1993), pp. 669–688. URL: `http://dblp.uni-trier.de/db/journals/ijprai/ijprai7.html#BromleyBBGLMSS93`.

[30] L. J. Lin. “Reinforcement Learning for Robots Using Neural Networks”. PhD thesis. Pittsburgh: Carnegie Mellon University, 1993.

[31] G. A. Rummery and M. Niranjan. *On-Line Q-Learning Using Connectionist Systems.* Tech. rep. TR 166. Cambridge, England: Cambridge University Engineering Department, 1994.

[32] C. Cortes and V. Vapnik. “Support Vector Networks”. In: *Machine Learning* 20 (1995), pp. 273–297.

[33] P. Dayan et al. “The Helmholtz Machine”. In: *Neural Computation* 7 (1995), pp. 889–904.

[34] Geoffrey E. Hinton et al. “The wake-sleep algorithm for unsupervised neural networks”. In: *Science* 268 (1995), pp. 1158–1161.

[35] Tin Kam Ho. “Random Decision Forests”. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1.* ICDAR ’95. USA: IEEE Computer Society, 1995, p. 278. ISBN: 0818671289.

[36] C. Goller and A. Kuchler. “Learning task-dependent distributed representations by backpropagation through structure”. In: *Neural Networks, 1996., IEEE International Conference on.* Vol. 1. IEEE, June 1996, 347–352 vol.1. ISBN: 0-7803-3210-5. DOI: `10.1109/icnn.1996.548916`. URL: `http://dx.doi.org/10.1109/icnn.1996.548916`.

[37] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[38] I. Michael Jordan. “Serial order: A parallel distributed processing approach”. In: *Neural-Network Models of Cognition - Biobehavioral Foundations. Advances in Psychology* (1997), pp. 471–495.

[39] Mike Schuster and Kuldip K. Paliwal. “Bidirectional recurrent neural networks.” In: *IEEE Trans. Signal Process.* 45.11 (1997), pp. 2673–2681. URL: `http://dblp.uni-trier.de/db/journals/tsp/tsp45.html#SchusterP97`.

[40] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE.* Vol. 86. 11. 1998, pp. 2278–2324. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665`.

[41] Geoffrey E Hinton. “Training products of experts by minimizing contrastive divergence”. In: *Neural Computation* 14.8 (2002), pp. 1771–1800.

[42] K. O. Stanley and R. Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: `10.1162/106365602320169811`.

[43] Yoshua Bengio et al. “Greedy Layer-Wise Training of Deep Networks.” In: *NIPS.* Ed. by Bernhard Schölkopf, John C. Platt, and Thomas Hofmann. MIT Press, 2006, pp. 153–160. ISBN: 0-262-19568-2. URL:

http://dblp.uni-trier.de/db/conf/nips/nips2006.html#BengioLPL06.

[44] Geoffrey E. Hinton. “Boltzmann machine”. In: *Scholarpedia* 2.5 (2007), p. 1668. DOI: 10.4249/scholarpedia.1668. URL: https://doi.org/10.4249/scholarpedia.1668.

[45] H. Jaeger. “Echo state network”. In: *Scholarpedia* 2.9 (2007). revision #189893, p. 2330. DOI: 10.4249/scholarpedia.2330.

[46] Wikimedia Commons. *Wikimedia Commons - A collection of freely usable media files.* [Online; accessed 10-February-2021]. 2007. URL: https://commons.wikimedia.org/wiki/Main_Page.

[47] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 2009, pp. 248–255.

[48] Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *CoRR* abs/1207.0580 (2012). arXiv: 1207.0580. URL: http://arxiv.org/abs/1207.0580.

[49] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate.* cite arxiv:1409.0473Comment: Accepted at ICLR 2015 as oral presentation. 2014. URL: http://arxiv.org/abs/1409.0473.

[50] Junyoung Chung et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[51] Ian J. Goodfellow et al. “Generative Adversarial Networks”. In: (June 2014). URL: https://arxiv.org/abs/1406.2661 (visited on 01/08/2017).

[52] Alex Graves, Greg Wayne, and Ivo Danihelka. *Neural Turing Machines.* 2014. arXiv: 1410.5401 [cs.NE].

[53] Christian Szegedy et al. *Going Deeper with Convolutions.* 2014. arXiv: 1409.4842 [cs.CV].

[54] François Chollet et al. *Keras.* https://keras.io. 2015.

[55] Mark Davies. *The Wikipedia Corpus.* [Online; accessed 25-March-2021]. 2015. URL: https://www.english-corpora.org/wiki/.

[56] Klaus Greff et al. “LSTM: A Search Space Odyssey”. In: *CoRR* abs/1503.04069 (2015).

[57] Kaiming He et al. *Deep Residual Learning for Image Recognition.* 2015. arXiv: 1512.03385 [cs.CV].

[58] Andrej Karpathy. *Andrej Karpathy blog: The Unreasonable Effectiveness of Recurrent Neural Networks.* [Online; accessed 19-February-2021]. 2015. URL: https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[59] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[60] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. “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.* ISCA, 2015, pp. 3214–3218. URL: http://www.isca-speech.org/archive/interspeech_2015/i15_3214.html.

[61] Tom Schaul et al. “Universal Value Function Approximators”. In: *Proceedings of the 32nd International Conference on Machine Learning.* Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 1312–1320. URL: `http://proceedings.mlr.press/v37/schaul15.html`.

[62] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 2015. arXiv: `1409.1556 [cs.CV]`.

[63] Ludovic Benistant et al. *Towards data science - A Medium publication sharing concepts, ideas, and codes.* [Online; accessed 10-February-2021]. 2016. URL: `https://towardsdatascience.com/`.

[64] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning.* 2016. arXiv: `1602.01783 [cs.LG]`.

[65] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* [Online; accessed 25-March-2021]. 2016. URL: `https://ruder.io/optimizing-gradient-descent/`.

[66] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. *Maximum Entropy Deep Inverse Reinforcement Learning.* 2016. arXiv: `1507.04888 [cs.LG]`.

[67] Barret Zoph and Quoc V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *CoRR* abs/1611.01578 (2016). arXiv: `1611.01578`. URL: `http://arxiv.org/abs/1611.01578`.

[68] Martin Bulín. “Optimization of Neural Network”. MA thesis. Univerzitní 2732/8, 301 00 Pilsen, Czech Republic: University of West Bohemia, June 2017.

[69] F. Caravelli, F. L. Traversa, and M. Di Ventra. “Complex dynamics of memristive circuits: Analytical results and universal slow relaxation”. In: *Physical Review E* 95.2 (2017). ISSN: 2470-0053. DOI: `10.1103/physreve.95.022140`. URL: `http://dx.doi.org/10.1103/PhysRevE.95.022140`.

[70] Tim Jones. *Recurrent neural networks deep dive.* [Online; accessed 25-March-2021]. 2017. URL: `https://developer.ibm.com/articles/cc-cognitive-recurrent-neural-networks/`.

[71] Risto Miikkulainen et al. “Evolving Deep Neural Networks”. In: *CoRR* abs/1703.00548 (2017). arXiv: `1703.00548`. URL: `http://arxiv.org/abs/1703.00548`.

[72] Ashish Vaswani et al. *Attention Is All You Need.* cite arxiv:1706.03762 Comment: 15 pages, 5 figures. 2017. URL: `http://arxiv.org/abs/1706.03762`.

[73] Jay Alammar. *The Illustrated Transformer.* [Online; accessed 25-March-2021]. 2018. URL: `http://jalammar.github.io/illustrated-transformer/`.

[74] Afshine Amidi and Shervine Amidi. *CS 230 - Deep Learning cheatsheets.* [Online; accessed 19-February-2021]. 2018. URL: `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks`.

[75] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.* cite arxiv:1810.04805Comment: 13 pages. 2018. URL: `http://arxiv.org/abs/1810.04805`.

[76] A. Dorri, S. S. Kanhere, and R. Jurdak. “Multi-Agent Systems: A Survey”. In: *IEEE Access* 6 (2018), pp. 28573–28593. DOI: `10.1109/ACCESS.2018.2831228`.

[77] Shuai Li et al. *Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN*. 2018. arXiv: `1803.04831 [cs.CV]`.

[78] OpenAI. *Key Papers in Deep RL*. `https://spinningup.openai.com/en/latest/spinningup/keypapers.html`. 2018.

[79] Simone Parisi et al. “TD-Regularized Actor-Critic Methods”. In: *CoRR* abs/1812.08288 (2018). arXiv: `1812.08288`. URL: `http://arxiv.org/abs/1812.08288`.

[80] Mingxing Tan et al. “MnasNet: Platform-Aware Neural Architecture Search for Mobile”. In: *CoRR* abs/1807.11626 (2018). arXiv: `1807.11626`. URL: `http://arxiv.org/abs/1807.11626`.

[81] Marina Chatterjee. *Top 20 Applications of Deep Learning in 2021 Across Industries*. `https://www.mygreatlearning.com/blog/deep-learning-applications/`. 2019.

[82] Sneha Chaudhari et al. “An Attentive Survey of Attention Models”. In: *CoRR* abs/1904.02874 (2019). arXiv: `1904.02874`. URL: `http://arxiv.org/abs/1904.02874`.

[83] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: `1907.11692 [cs.CL]`.

[84] Marek Rosa et al. “BADGER: Learning to (Learn [Learning Algorithms] through Multi-Agent Communication)”. In: *CoRR* abs/1912.01513 (2019). arXiv: `1912.01513`. URL: `http://arxiv.org/abs/1912.01513`.

[85] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. “Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms”. In: *CoRR* abs/1911.10635 (2019). arXiv: `1911.10635`. URL: `http://arxiv.org/abs/1911.10635`.

[86] Nikolas Adaloglou. “Intuitive Explanation of Skip Connections in Deep Learning”. In: *https://theaisummer.com/* (2020). URL: `https://theaisummer.com/skip-connections/`.

[87] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: (2020). arXiv: `2005.14165 [cs.CL]`.

[88] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: `2002.08155 [cs.CL]`.

[89] Anmol Gulati et al. *Conformer: Convolution-augmented Transformer for Speech Recognition*. 2020. arXiv: `2005.08100 [eess.AS]`.

[90] Andrey Kurenkov. “A Brief History of Neural Nets and Deep Learning”. In: *Skynet Today* (2020).

[91] Dent Neurologic Institute. *22 Facts About the Brain | World Brain Day*. [Online; accessed 19-February-2021]. 2021. URL: `https://www.dentinstitute.com/posts/lifestyle-tips/22-facts-about-the-brain-world-brain-day/`.

[92] Hieu Pham et al. “Meta Pseudo Labels”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2021. URL: `https://arxiv.org/abs/2003.10580`.

# Appendix A1

# Notation Conventions

Notation conventions used in this study are overtaken from (Bulín, 2017).

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:

$$\Delta_{(1)}^{(q+1)} = \begin{bmatrix} \delta_{11}^{(q+1)} \\ \delta_{21}^{(q+1)} \\ \vdots \\ \delta_{m1}^{(q+1)} \end{bmatrix}$$