Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Tor Istvan Stadler Kjetså

# MLOps - challenges with operationalizing machine learning systems

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
Co-supervisor: Bjarne Grimstad

May 2021

NTNU
Norwegian University of
Science and Technology

Tor Istvan Stadler Kjetså

# MLOps - challenges with operationalizing machine learning systems

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
Co-supervisor: Bjarne Grimstad
May 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

# Assignment

MLOps is the discipline of operationalizing machine learning systems. Machine learning systems are usually complex and particularly vulnerable to errors. They are typically of higher complexity than traditional/non-adaptive software systems. Unpredictable data that changes over time, combined with adaptive software learning from this data, introduces a new set of challenges. Recent years have seen a surge in the technological development of MLOps, and it is in the process of being established as a new scientific field. This means that literature in this area is limited.

This thesis researches testing of machine learning systems and MLOps, and associated challenges. It investigates how machine learning systems are typically developed and what are the components making up their life cycle. This thesis aims to highlight the challenges faced when operationalizing these systems. The thesis takes on a system perspective, but also elaborates on important details concerning the components in such a system. An important contribution of this thesis is gathering relevant information and literature to be presented in a systematic manner, which can be demanding in a new and unestablished field.

This thesis is conducted in collaboration with Solution Seeker, a company delivering AI-as-a-service to the process industry.

Tasks:

- Perform a study on machine learning theory, with emphasis on deep learning, and a study on MLOps.
- Investigate how testing can be performed on machine learning systems.
- Make a systematic overview of a typical machine learning life cycle. Describe the different components in this life cycle.
- Implement an experimental machine learning system; a simple machine learning system applied to a process system, to highlight challenges.
- Discuss testing, challenges, and technology choices and provide advice for good practice regarding operationalizing a machine learning system.

# Preface

I wish to thank my supervisor Sverre Hendseth for helping me make this assignment my own and for encouraging me to pursue what I myself find interesting, using his impressive ability to having me answer my own questions. I also wish to thank my co-supervisor and future colleague Bjarne Andre Grimstad for providing professional insights and consistently pushing me on to the next level.

May, 2021
NTNU, Trondheim

Tor Istvan Stadler Kjetså

# Abstract

There is an increasing demand for machine learning applications within several industries. While good machine learning models exist, there is a widespread struggle in operationalizing them. A lack of tools and established best practices on how to operationalize machine learning system results in many models being left on the shelf. Machine learning systems differ from traditional software in that they are dynamic and stochastic. This poses new challenges, especially in terms of testing. MLOps is the newly emerged discipline of operationalizing machine learning system. However, exactly what it involves is yet to be formally established. This thesis aims to investigate the challenges associated with operationalizing machine learning systems, the current status of MLOps, and propose a set of best practices for how to go about operationalizing a machine learning system. A study is performed on machine learning theory, with emphasis on deep learning to be familiarized with the concept and to provide background for challenges and requirements concerning operationalization. Further, an investigation is conducted on methodologies for testing machine learning systems. Current research on MLOps is explored, and an overview is presented of the typical components in a machine learning life cycle, and how they integrate. A set of modern technologies related to MLOps are reviewed. Lastly, an experimental miniature machine learning system is implemented, providing first-hand experience with machine learning development, and highlighting associated challenges. Some of the investigated testing methodologies are found useful, and others are more specific or experimental. Based on the research and work conducted throughout this thesis, a set of best practices for how to approach the operationalization of machine learning systems is proposed, addressing common challenges, including technology choices. The developed machine learning system exhibits a minimum viable product and can be used for testing some general machine learning-related techniques.

# Terminology

In the machine learning field there exists a plethora of terms and phrases that are not common lingo within ordinary software development. This section aims to elaborate on some of these in order to provide clarity concerning the terminology in this thesis.

**ML:** the term machine learning is often abbreviated as ML.

**ML life cycle:** constitutes the different steps required to build and maintain a machine learning system. A life cycle signifies a continuous process.

**Traditional software/Software 1.0:** software without machine learning.

**Software 2.0:** software that applies machine learning.

**Big Data:** large amounts of data. Refers the increased availability of data during the last decades.

**(ML/Data/Deployment) Pipeline:** a series of transformations or operations applied to data or code between its source and destination.

**Online learning:** frequently updating/retraing a machine learning model using a continuous stream of data.

**ML Ops/MLOps/Model Ops/Model Operations:** are used interchangeably, this thesis refers to the term as **MLOps**.

# Table of Contents

**Assignment** i

**Preface** ii

**Abstract** iii

**Terminology** iv

**1 Introduction** 1

1.1 Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Methodology and structure . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Deep Learning** 6

2.1 Course 1: Neural Networks and Deep Learning . . . . . . . . . . . . . . . . 6

2.2 Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2.1 Setting up the optimization problem . . . . . . . . . . . . . . . . . . 15

2.2.2 Optimization algorithms . . . . . . . . . . . . . . . . . . . . . . . 17

2.2.3 Hyperparameter tuning . . . . . . . . . . . . . . . . . . . . . . . 19

2.2.4 Batch normalization . . . . . . . . . . . . . . . . . . . . . . . . 20

2.3 Course 3: Structuring Machine Learning Projects . . . . . . . . . . . . . . . 22

2.3.1 Orthogonalization . . . . . . . . . . . . . . . . . . . . . . . . . 22

2.3.2 Setting up the goal . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.3.3 Error analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.3.4 Data set distributions . . . . . . . . . . . . . . . . . . . . . . . . 24

2.3.5 Learning from multiple tasks . . . 25
2.3.6 End-to-end deep learning . . . 25
2.4 Course 4: Convolutional Neural Networks (CNNs) . . . 26
2.4.1 Examples of efficient convolutional network architectures . . . 29
2.5 Course 5: Sequence Models . . . 31

**3 Techniques for Testing Machine Learning Systems 38**
3.1 Conventional Software Testing . . . 38
3.1.1 Pre-train testing . . . 39
3.2 CheckList: Three Types of Behavioural Testing . . . 39
3.3 Improving Dependability of Machine Learning Applications . . . 41
3.4 Testing Deep Neural Networks . . . 42
3.5 Developing Bug-free Machine Learning Systems with Formal Mathematics . 45

**4 Employing MLOps 47**
4.1 Machine Learning Pipeline . . . 47
4.2 Machine Learning Operations (MLOps) . . . 49
4.2.1 A four-step model of MLOps . . . 53

**5 Components in the Machine Learning Life Cycle 59**
5.1 Data Operations . . . 64
5.2 Training, Validating and Refinement . . . 64
5.3 Model Evaluation . . . 66
5.4 Deployment . . . 69
5.5 Monitoring . . . 70

**6 Modern Technologies for Machine Learning Operationalization 72**
6.1 Docker and Kubernetes . . . 72
6.1.1 Docker . . . 72
6.1.2 Kubernetes . . . 73
6.2 Run:AI . . . 73
6.3 Apache Kafka . . . 74
6.4 Dataflow . . . 75

6.5 Apache Spark . . . 76
6.6 MLflow . . . 78
6.7 Databricks . . . 79

**7 Miniature Machine Learning System** **81**
7.1 System Specification . . . 81
7.2 Design Choices . . . 81
7.3 Configurability . . . 82
7.4 Process Simulator for Data Generation . . . 82
7.4.1 CSTR with van de Vusse reaction . . . 82
7.4.2 Modelling framework . . . 84
7.5 Neural Network . . . 85
7.6 Setting up Modular Machine Learning Pipelines . . . 86
7.7 Demonstration . . . 87

**8 Discussion** **92**
8.1 Testing . . . 92
8.2 MLOps . . . 93
8.3 Technologies . . . 94
8.4 Miniature machine learning system . . . 96

**9 Conclusion** **97**
9.1 Advice for best practices . . . 97
9.2 Miniature machine learning system . . . 98
9.3 Future Work . . . 99

**Bibliography** **100**
Appendix . . . 113
A Program files for miniature machine learning system . . . 113
B GoogLeNet . . . 132

# Chapter 1

# Introduction

Machine learning is a method of data analysis that automates analytical model building by using algorithms that improve automatically by the use of data. While artificial intelligence (AI) is the broad science of mimicking human abilities, machine learning is a specific subset of AI that trains a machine how to learn. Machine learning itself is not a brand new science, but the ability to rapidly and automatically apply complex mathematical calculations to large amounts of data (big data) is a recent development. As such; machine learning is not a new science, but one that has gained fresh momentum [1].

Deep learning is a subset of machine learning which has emerged from the era of big data. Deep learning algorithms are applied to particularly complex problems because of characteristics that provide good prediction performance [2, 3, 4]. Deep learning is manifesting as an important technique for providing predictive analytics solutions for large-scale data sets, and has, in the last decade, attracted much attention from the academic communities within speech recognition, computer vision, language processing, and information retrieval [2, 5, 6, 7, 8].

Deep learning systems are the type primarily considered in this thesis. Still, most of the research is general and largely applicable to other machine learning systems.

The fresh momentum gained by machine learning has extended its range of applications, and many industries are attempting to adopt this technology and apply it to their domains. This has led to increasing demand for operationalized machine learning systems.

Operationalizing a system, in the context of machine learning, involves migrating machine learning from theory and statistics into production where it can perform real-time predictions to be used by the system into which it is integrated. Deep learning can be used in control systems where there is an abundance of data, but the correlation between input and output is too complex to model with mathematics in a practical manner, or as predictive systems where forecasts are made based on correlations in data that are beyond human comprehension ability. Some examples of this are presented below.

Johansson et al. [9] present an operational system where an ensemble of online machine learning algorithms are used to predict heat demand, to improve a district heating and cooling (DHC) system. Since DHC systems are demand-driven, the ability to predict future demand is a valuable feature in terms of optimizing energy efficiency. The heat demand in all the households and other units controlled by a DHC system is influenced by a lot of

variables and is thus difficult to model using mathematical equations. Using empirical data to train machine learning algorithms has been proven beneficial for this system. Using online learning has also enabled the algorithms to adjust to new data, e.g. seasonal differences.

Lagerquist et al. [10] present the development of convolutional neural networks to predict next-hour tornado occurrence. The National Weather Service (NWS) is responsible for issuing tornado warnings and generally issues warnings at lead times up to 30 minutes. And although the skill of NWS tornado warnings has generally improved over time, critical success index (CSI)[1] and lead time have stagnated in the last decade [12]. What has not stagnated is the amount of data available to forecasters; this has increased significantly and includes dual-polarization radar observations, high-resolution satellite observations, and forecasts from convection-allowing models. This data is a valuable resource and encourages the development of a machine learning system. The predictors during this experiment were proximity sounding and storm-centred radar images. Comparison with ProbSevere, a machine learning model currently used for operational severe weather prediction, suggests that the developed models would be useful operationally.

Nallaperuma et al. [13] propose an expansive smart traffic management platform (STMP) based on several machine learning techniques, including deep learning. During the last decades, the technological landscape of transportation has gradually integrated disruptive technology paradigms into current transportation management systems, leading to intelligent transportation systems (ITS) [14, 15]. Internet of Things (IoT), sensor networks and social media have lead to increased efficiency of data collection, with voluminous and continuous streams of real-time data. Nallaperuma et al. [13] reports that the volatile data generation and the dynamicity of data generated that originates from these sources present challenges for and impede the effectiveness of many existing AI techniques. One of the main challenges is that of concept drift[2]. The solution proposed revolves around frequent updates/retraining of the machine learning model, using the continuous stream of new data, i.e. online learning. The STMP combines this technique with several types of machine learning algorithms and data from a variety of sources to produce a platform that has been successfully demonstrated on 190 million records of smart sensor network traffic data generated by 545,851 commuters and corresponding social media data on the arterial road network of Victoria, Australia.

Operationalizing machine learning requires integrating machine learning models with supporting software and production pipelines to form a complete system. Data must be structured and directed into the machine learning algorithm, and the resulting output must be interpreted and acted upon. The machine algorithm itself simply outputs data. This is where data scientists and software engineers must collaborate to achieve operationalization.

Machine learning systems possess a larger degree of scientific characteristics and mathematical complexity compared to traditional software - they combine data and code in dynamic, adaptive stochastic systems. Such systems invoke a completely new set of demands and challenges that do not apply to traditional software systems.

A distinct challenge is testing. The dynamic nature of machine learning systems necessitates frequent testing. The stochastic nature renders traditional software testing techniques

[1] Also called the threat score (TS), is a verification measure of categorical forecast performance equal to the total number of correct event forecasts (hits) divided by the total number of storm forecasts plus the number of misses (hits + false alarms + misses) [11].

[2] Concept drift primarily refers to an online supervised learning scenario when the relation between the input data and the target variable changes over time [16].

insufficient. Testing is important for achieving quality assurance, and well-defined tests allow for automating processes.

MLOps (abbreviated from Machine Learning Operations) refers to the discipline of operationalizing machine learning systems. At its core, MLOps is a set of practices on how to build, deploy, monitor, and maintain machine learning pipelines in production or operational systems. Being a relatively ripe field in rapid development, MLOps has experienced a lack of attention in terms of establishing guidelines and best practices.

Naturally, most available papers present a successfully operationalized machine learning system; it is less appealing to share the inability to figure out how to put a spectacular machine learning model to use. However, various reports highlight the struggle in operationalizing machine learning.

According to Deeplearning.ai [17], only 22% of companies using machine learning have successfully deployed a model. Research from Gartner shows that only 53% of AI prototypes make it into production, due to a lack of tools [18]. Matt Macaux, Global Field CTO for HPE Ezmeral [19], reports that $\sim 60\%$ models are built, but not operationalized [20], also mentioning the lack of tools as a reason.

MLOps requires both machine learning expertise and software engineering operating in harmony - a gap between software engineers and data scientists hinders its utility [21, 22]

## 1.1 Scope

The scope of this thesis includes a review of the fundamentals of deep learning and a more concise review of some well-established techniques and extensions in the field. How to test machine learning systems is discussed, and associated challenges and possible approaches are addressed through reviews of some methodologies of varying prevalence. Also within the scope is research on MLOps; considering different definitions, discussing its prevalence and how it fits in with related disciplines and evaluating its importance. Complementing this is a systematic overview of a machine learning life cycle accommodating MLOps. Some components of particular interest are discussed in more detail. The scope further includes a short review of a set of modern technologies related to MLOps. Lastly, within the scope is developing a miniature machine learning system. This includes a neural network, a simulator for a CSTR with a Van de Vusse reaction, and supporting modules required to build pipelines.

## 1.2 Methodology and structure

The coverage of MLOps, machine learning pipelines, machine learning deployment, and other closely related fields is scarce in terms of acclaimed literature. Evolution of and discussions about the topics are heavily influenced by ad hoc applications, unpublished articles, and blog posts. This means that a keen eye and a selective attitude is of the essence regarding research in this area. As a result of this, a solid understanding of the fundamentals of deep learning, as well as first-hand exposure to challenges in machine learning development, is considered of increased value as learning resources.

Chapter 2 provides an immersion into deep learning through a series of courses offered by Coursera [4]. The first three courses involve the fundamentals of how deep learning

works, challenges associated with it, and some techniques for how to structure machine learning projects. Much of the content also applies to machine learning in general. The insight provided by these courses is highly relevant for understanding the techniques for and challenges associated with operationalizing machine learning systems. Courses number four and five introduce techniques that extend the basic deep learning functionality. These techniques are widely adopted in deep learning applications and are in many cases essential for their feasibility and performance levels. A detailed explanation of them is not highly relevant for the remainder of this thesis, but a review of their high-level structure and functionality highlights the complexity and computational power required by some machine learning systems - making them demanding to operationalize.

A task that is particularly difficult when performed on machine learning systems is testing. Adaptivity and stochasticity require frequent testing and non-binary test results. Such tests can be difficult to create. The basic idea of machine learning is to have the algorithm build itself, making it inherently difficult to inspect its structure and logic to verify its correctness. Testing of machine learning systems is discussed in chapter 3. This chapter also includes reviews of some papers presenting testing approaches of various prevalence. These reviews are results of an optimistic search for acclaimed testing methods and help to illustrate the variety of use cases and approaches within this area.

Chapter 4 involves research on the current state of MLOps and what it constitutes, including important preliminaries and how to go about employing it, such as machine learning pipelines and organizational philosophies.

Chapter 5 presents typical components in a machine learning life cycle organized in a proposed architecture. Particularly noteworthy components are elaborated on in greater detail and supplemented with references to relevant literature. The illustrative architecture and corresponding component overview are intended to form a basis for structuring a sustainable machine learning life cycle. The chapter works to highlight the benefit of thinking in terms of MLOps from an early development stage and is formulated to appeal to both data scientists and software engineers in order to form a common understanding and bridge the gap between them.

Chapter 6 reviews a selection of technologies in the business of supplying support for machine learning operationalization. There is only so much insight to be gained from reading about a technology as opposed to testing it. However, their solutions and focus areas help highlight some of the challenges associated with operationalizing machine learning from a more practical point of view, contrasting the hitherto theoretically dominated perspective.

Chapter 7 presents the development of a miniature machine learning system and its ultimate functionality. The system consists of a neural net modelling an industrial process, chosen to be a CSTR with a Van de Vusse reaction, in the form of a simulator, along with the building blocks for training and prediction pipelines.

This machine learning system is an integral part of this thesis in the sense that its development was initiated early in the process and has functioned as a touchstone throughout it, by allowing first-hand exposure to the machine learning scene. The process of actually writing code and implementing a system offers a different perspective on challenges and needs as opposed to that of reading research papers. As a result, developing - as a parallel process throughout this thesis - has directed the structure of it.

Along with this influence; the system itself forms the foundation of a valuable tool in terms of testing techniques and technologies. It provides a shortened feedback loop, as the system

can run locally, data can be quickly produced through simulations, and the configurable size of the neural network offers relatively fast training jobs.

In chapters 8 and 9 methodologies for testing are discussed and evaluated. Challenges for machine learning are summarized, and current MLOps approaches are discussed. A brief personal view is offered on modern technologies, complemented with comments on how to approach technology selection. Insight gained machine learning development is presented, and possible additions to form a more complete system are discussed. These reflections contribute to a proposed set of advise for best practice in approaching the operationalization of machine learning systems.

# Chapter 2

# Deep Learning

This chapter provides insight into the fundamentals of machine learning theory - with specific emphasis on deep learning, constituting important groundwork for the remainder of this thesis. To understand the challenges and requirements discussed in later chapters it is important to be familiar with the basics of deep learning. This chapter also highlights some popular extensions to basic deep learning functionality in sections 2.4 and 2.5. These extensions have grown to be established as state of the art techniques, and are vital to several active deep learning focus areas, such as image recognition and natural language processing (NLP). They bring attention to the complexity and computational power associated with deep learning, and some practical applications are mentioned.

The research is conducted through aspecialization[1] on deep learning, named Deep Learning Specialization, offered by Coursera [4]. The specialization is quite detailed and consists of five courses:

- Course 1: Neural Networks and Deep Learning
- Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization
- Course 3: Structuring Machine Learning Projects
- Course 4: Convolutional Neural Networks
- Course 5: Sequence Models

each of which is divided into approximately 4 weekly sub-modules. The most important parts of the courses are summarized throughout this chapter. Everything discussed in this chapter, except where a different source is explicitly specified, is from courses by Coursera [4].

## 2.1 Course 1: Neural Networks and Deep Learning

Deep learning is a sub-science of machine learning; and it refers to training neural networks, often very large ones. The adjective "deep" is used to emphasize the use of multiple layers

[1] Coursera-term for a collection of courses

in a neural network, as illustrated in fig. 2.1.

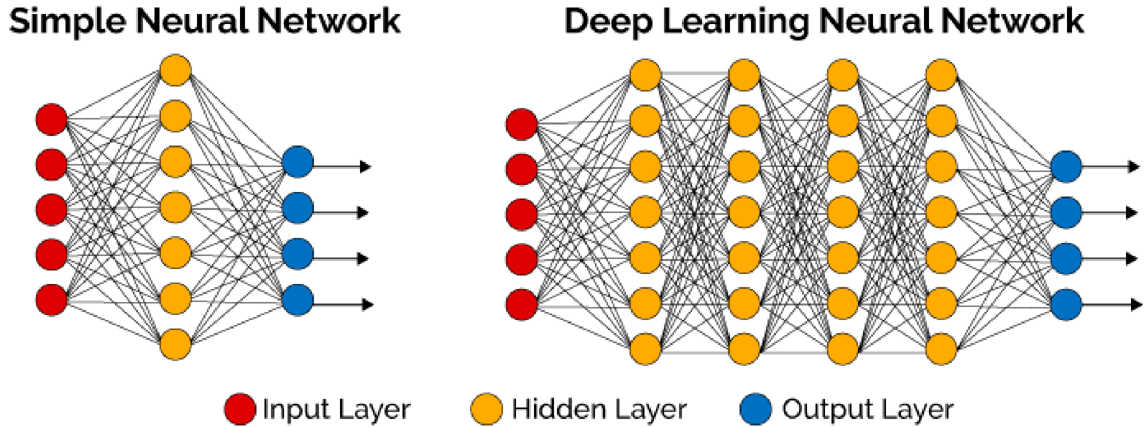



Figure 2.1: A simple and a deep neural network, as illustrated by Vazquez [23].

#### What is a neural network?

A neural network is, as the name implies, a network of neurons. A neuron is a unit that takes a number of inputs $x$, multiplies by weights $w$ and adds biases $b$, as shown in eq. (2.1).

$$\sum_{i=1}^{N} w_i x_i + b_i \tag{2.1}$$

The result is passed to an activation function whose output is the output of the neuron. This output can be either the output of the entire neural network or be passed as an input to the next layer in the neural network. Figure 2.2 illustrates a single neuron.

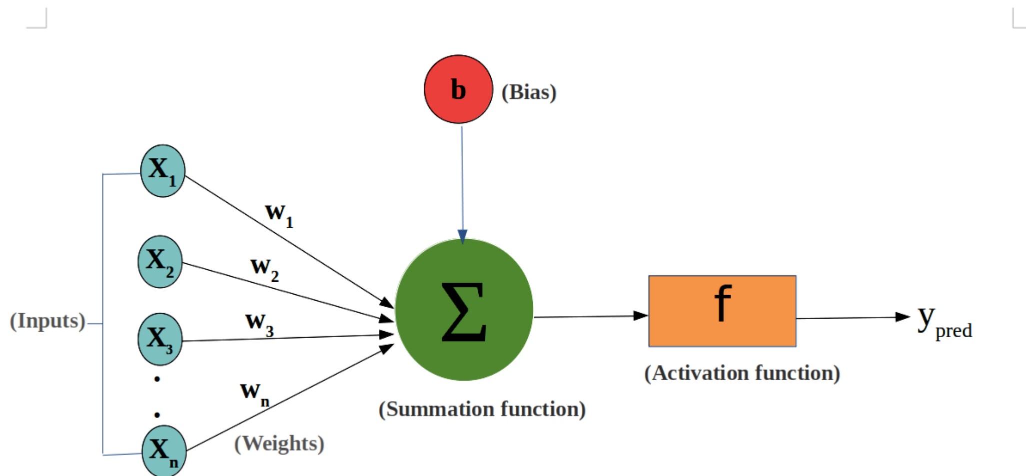



Figure 2.2: The structure of a single neuron, as illustrated by Ganesh [24].

As mentioned, a deep neural network consists of multiple layers. The layers that are neither input nor output layers are referred to as *hidden layers*. This because the outputs of these layers are not visible outside the neural network, but function as part-computations that contribute to the network as a whole.

### Activation function

The activation function is a nonlinear function that essentially determines the significance of the summation in the neuron, represented in a way that is favourable for the succeeding layers of the neural network. The activation function can, in theory, be anything, but not all choices are equally likely to be successful. A simple example of an activation function is the ReLU (Rectified Linear Unit)-function. Say the purpose of a neural network is to predict the price of a house, and the inputs are attributes like area, neighbourhood, parking space, etc. Whatever the inputs, weights and biases sum to, the price of a house cannot be negative. Thus there is no purpose for the activation function of the output layer to output a negative number. The ReLU-function computes $ReLU(z) = max(0, z)$, yielding an output that is always non-negative.

Some common activation functions are:

- *sigmoid:* $\phi(z) = \frac{1}{1+e^{-z}}$
- *tanh:* $\phi(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
- *ReLU:* $\phi(z) = max(0, z)$
- *leaky ReLU:* $\phi(z) = max(a * z, z)$ where $a$ is a small constant, e.g. $a = 0.01$, so that the slope of the function is decreased when below zero.

### The cost function

To measure how well a neural network performs, it is needed to define exactly what it is that the neural network is meant to achieve. This is where the cost function enters. A neural network will work to minimize the defined cost function, and the predictions it produces will be the predictions that the neural network "believes" will produce the lowest cost. The most common cost function is the mean squared error (MSE) between the prediction of the neural network, $\hat{Y}$ and the actual measurement, $Y$, shown in eq. (2.2)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.2}$$

The cost function is usually denoted $J(w, b)$, where $w$ are the weights, and $b$ the biases.
*Note: The loss function refers to the error of a single training example, while the cost function refers to the average of the loss function over the entire training set.*

### Gradient Descent - forward and backward propagation

Gradient descent is an algorithm that the neural network uses to minimize the defined cost function. The gradient descent algorithm starts with an initial guess for values for weights and biases in the neural network. Followingly, *forward propagation* is performed. Forward propagation is the process of the neural network applying its weights, biases and activation functions to a given input to produce an output. As the weights and biases are randomly initialized, the resulting prediction is likely to be far off. To adjust the weights and biases, their derivatives with respect to the cost function are computed for every layer in the neural

network. The weights and biases are then adjusted by "taking a step" in the direction of their respective derivatives. The length of this step is decided by the *learning rate*. This process is referred to as *backward propagation* or *backpropagation*, as it propagates the error backwards in the neural network. By performing forward and backward propagation over several iterations the neural network will eventually end up with weights and biases that produce an optimal (or close to optimal) output with respect to the cost function. Figure 2.3 gives a schematic of gradient descent. A caveat for gradient descent is that it only

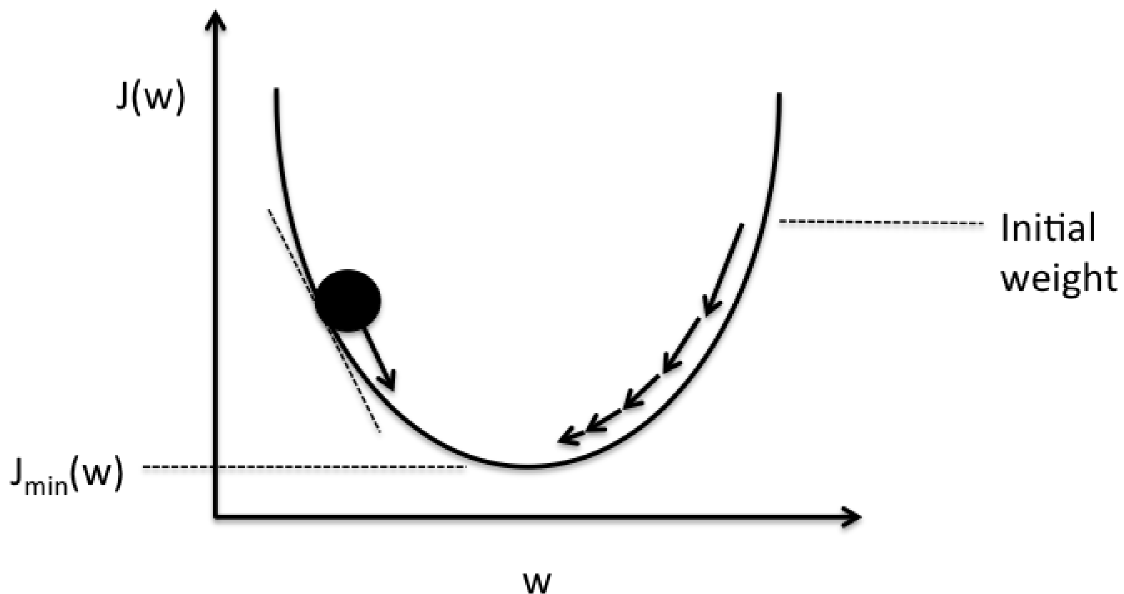



Figure 2.3: Schematic of gradient descent, as illustrated by Moawad [25].

guarantees convergence to a local minimum, and thus works sub-optimally for non-convex functions - which have several local minima.

### Logistic Regression

Logistic regression is a relatively simple learning algorithm used for binary classification problems; when all output labels $Y$ in a supervised learning problem are either zero or one. In such a case one would want the neural network to produce an output prediction $\hat{Y} \in [0, 1]$, based on the input features $X$. $\hat{Y}$ represents the estimated probability for $Y = 1$. With the use of an activation function, the network will convert this estimate into an output of 1 or 0.

### Weight initialization

When initializing the weights in a neural network, they must be randomly initialized, not initialized to zero. Zero-valued weights eliminate the influence of input values on a neuron, since they are multiplied by zero. Succeeding activations will be equal because all hidden units are computing the exact same function. In backpropagation, when computing the

derivatives of each hidden unit, they will also be identical, due to them having the same influence on the output. After each iteration, when the weights are updated, it is proved by induction that the hidden units will still compute the same function. Thus, the functionality is equal to that of using only one hidden unit. This concept is referred to as hidden units being *symmetric*. On the other hand, biases can be initialized to zero.

### Shallow versus deep neural network

Figure 2.1 illustrates a shallow and a deep neural network. Shallow versus deep is a matter of degree. Over the last years, the machine learning community has realized that there are functions that very deep neural networks can learn, which shallower models are often unable to. It might be hard to predict in advance exactly how deep the neural network should be.

### Parameters and hyperparameters

The parameters in a neural network are the weights and biases that are adjusted as the network is trained. There are other characteristics as well, that make up a model, specifying the structure of the neural network and how it is trained. These are called *hyperparameters*. A common hyperparameter is the *learning rate*, which decides how much the weight and biases are adjusted at each iteration. The number of layers in a neural network can also be considered a hyperparameter. Another hyperparameter, discussed in the next section is the *regularization parameter*. Optimal hyperparameter values are not so easy to obtain, and they often differ between the domains to which machine learning is applied. As a result, hyperparameter tuning is usually subject to iterative development. This is discussed in section 2.2

### Supervised Learning, Unsupervised Learning, Reinforcement Learning

There are three primary methods of machine learning, the following definitions are taken from [1]:

- **Supervised learning** algorithms are trained using labelled data. Input data with corresponding output labels are fed into the algorithm. The algorithm then compares its output with the actual label and then adjusts itself to best fit its output with the given labels. The goal is that when an algorithm is trained on enough labelled data, it can correctly predict the label of unlabeled input data, e.g. label an image as a cat or not a cat. Most machine learning examples in this thesis are instances of supervised learning.
- **Unsupervised learning** algorithms are used on unlabeled data. The algorithm is not given any "correct answer", the goal is to explore the data and find some structure in it. This might reveal patterns between different features that the human eye is unable to recognize. An example of unsupervised learning is identifying people/customers that should be treated similarly in marketing campaigns.
- **Reinforcement learning** algorithms use trial and error to discover which actions, in which states, yields the highest reward. A reinforcement learning algorithm consists of three main components: *the agent* is the decision-maker; *the environment* describes

what the agent interacts with; *the actions* make up the possibilities that the agent must decide between. The objective is for the agent to choose actions that yield the highest combined reward over a given period. Reinforcement learning is often used for robotics, gaming, and navigation.

## 2.2 Course 2: Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

#### Splitting data sets

All the data available for creating a neural network must be divided into separate data sets with distinct purposes; a training set, a validation/development set, and a testing set. Setting up the data sets well can help speed up the iterative development process. The training set builds the neural net. The validation set is then used to assess how the net is trained and consider whether to perform any adjustments. The validation set can also be used to determine which model to use from a selection of trained models. An example of this is cross-validation, where the training set is divided into $k$ parts, and the model is trained on $k-1$ of these parts and validated on the last one. This requires $k$ models to be trained from scratch since each of the $k$ parts should act as the validation set once. Not all training procedures involve a validation set, e.g. if the amount of data is scarce, using some of it for validation might not be prioritized.

The test set is used for the final assessment of the model. After the net is subjected to the test set, no adjustments based on the test results should be performed. Doing this is referred to as *data leakage*, where the model is exposed and tailored to the data it should be tested on. Data leakage renders the test results invalid, as they no longer give any information regarding the predictive power of the neural network. As the validation set is used to tune the model, the validation set and the test set should come from the same distribution, to score the model as accurately as possible.

In the previous era of machine learning, it was common practice to split the data with a ratio of 70/30 in a train/test split, and perhaps 60/20/20 in a train/validate/test split. In the modern big data era, where there might be millions of examples to train on, these fractions have changed, and a more common split ratio is around 98/1/1 for train/validate/test.

#### Bias and variance

A model that suffers from high bias is said to have underfitted the data, while a model with high variance has overfitted the data. Underfitting the data signifies not paying much attention to it and thus ending up with an oversimplified model, insensitive to important data features. Overfitting is the exact opposite, where the data is paid too much attention to, and the resulting model is poor at generalizing unseen data.

Bias is the difference between the mean prediction by the model and the correct values. Variance is the variability of model predictions.

Figure 2.4 illustrates combinations of low and high bias and variance with the use of a bullseye target, and fig. 2.5 illustrates the behaviour of an underfitted, an overfitted, and a well-balanced model.

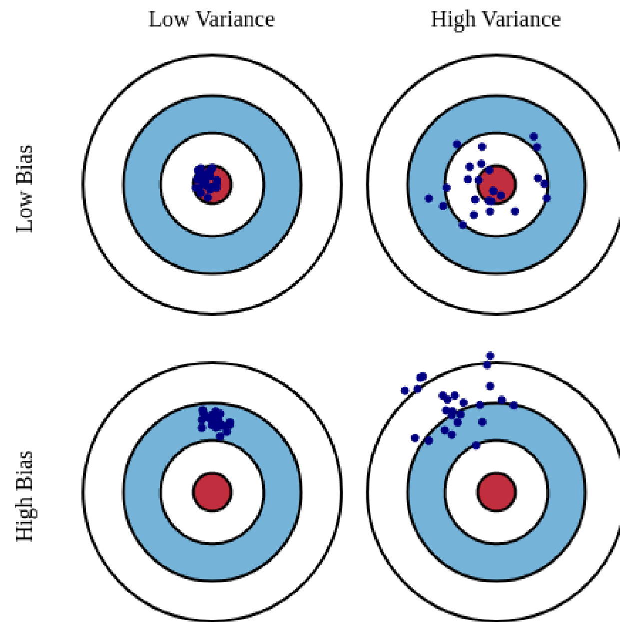



Figure 2.4: Low and high bias and variance, illustrated with bullseyes by Fortmann-Roe [26].

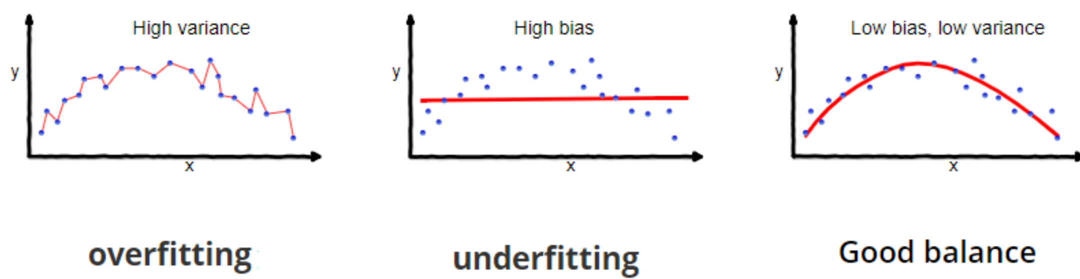



Figure 2.5: Overfitting and underfitting compared with good balance, as illustrated by Singh [27].

The key indicators of bias and variance are the training set and validation set errors. A high training set error indicates that the algorithm suffers from high bias, and has underfitted the training data. A high difference between the training set error and the validation set error indicates that the algorithm suffers from high variance. Since the algorithm performs much worse on unseen data, it has probably overfitted the data and generalized poorly. These assessments are based on the assumptions that the base error[2] is relatively small, and that the training and validation sets are from the same distribution.

At the early stages of machine learning, the bias-variance tradeoff was an important consideration, as one was improved at the expense of the other. This is not as relevant in modern machine learning, as it is possible to improve both without affecting the other. If the algorithm suffers from high bias, then a possible solution is to extend the neural network, by adding more layers. Obtaining more data or altering the architecture of the neural net might also counteract bias. The latter two measures can be effective for addressing high variance as well. Perhaps the most efficient tool for counteracting overfitting, however, is called *regularization*.

### Regularization

Regularization is the practice of regulating the degree to which the model fits the data. The most common regularization technique is called *L2 regularization*. L2 regularization uses the squared L2-norm (Euclidean distance) of the weights vector. This is multiplied with a *regularization parameter*, $\lambda$, and divided by $2m$, where $m$ denotes the number of training examples. To achieve regularization, this term is added to the cost function $J$ to penalize large weight values.

In a neural network, there are several hidden layers, and the weights are represented by a matrix, $W$, containing the weight vectors for each layer, $w^{[l]}$. When implementing L2-regularization for a neural network, it is actually the squared *Frobenius norm* that is applied to the cost function. The squared Frobenius norm is the sum of all the squared L2-norms of the vectors in the matrix. Equation (2.3) shows a cost function where L2 regularization is implemented. Here $J$ denotes the cost function, $\mathcal{L}$ the loss function for each training example, $W$ the weight matrix, $w^{[l]}$ the weight vector for layer $l$, and $b$ the bias vector. $\lambda$ is the regularization parameter, $L$ and $m$ are the number of layers and training examples, respectively, and $\hat{y}^{(i)}$ and $y^{(i)}$ are the prediction and actual value, respectively, for training example $i$. $||W||_F^2 = \sum_{l=1}^{L} ||w^{[l]}||^2$ denotes the squared Frobenius norm of the weight matrix, $W$.

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} ||w^{[l]}||^2 \tag{2.3}$$

It is possible to add the bias of the respective layer $b^{[l]}$ to the regularization term, but this has limited impact and is often not prioritized. A less frequently used regularization technique is L1 regularization, where the squared L2-norm is substituted with the L1-norm

[2]The base error, or the optimal error, is the error one could expect for a near-perfect classifier. For difficult problems, the base error is usually higher.

(Manhattan distance), resulting in eq. (2.4).

$$J(W, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^{L} |w^{[l]}| \tag{2.4}$$

The general difference between L1 and L2 regularization is that L1 regularization is more prone to shrinking some weights to zero, effectively removing some features' impact on the output, yielding a network with lower complexity. This can work well for *feature selection*[3] in cases where there a huge number of features. L2 regularization shrinks the weights overall, but are more incentivised to penalising large values than further shrinking small values, due to the squared term. Thus, L2 regulates the impact of all features but removes fewer.

Another powerful regularization technique is *dropout regularization*. In dropout regularization, each node in each layer is given a probability of whether or not it will be included in an iteration of forward propagation. If the node is not included, all outputs from it are omitted in the proceeding computation and the following backpropagation. The process is repeated for each training example, resulting in a different set of nodes being used at each iteration. This contributes to regulating the importance of each feature and thus prevents overfitting. The dropout is only applied during training, to regulate the weighting - when testing, applying dropout will result in noisy predictions. A downside with dropout is that the cost function is no longer well-defined. Since random nodes are omitted at each iteration, it is difficult to verify that the cost function is monotonically decreasing. Even though no guarantees are offered, it is possible to run through the training set once without dropout, and verify that the cost function is decreasing, first, and then applying dropout, hoping no bugs have been introduced.

An alternative regularization approach is *data augmentation*. To avoid the algorithm overfitting data and focusing on less important features, it is possible to augment the training data to provide a more diverse training set. Say the goal is to train an algorithm that can recognize pictures of cats. If all the images in the training set contain cats facing towards the right, the algorithm might struggle when presented with a cat facing to the left. Simply adding a flipped copy of the cat pictures in the training set will make the algorithm focus less on the orientation of the cat, and more on more important features.

The last regularization technique to be mentioned is *early stopping*. This technique involves running the algorithm on the validation set after each training iteration, and monitoring how the cost function behaves. If the cost function starts increasing halfway through the training procedure, it is possible to halt the training procedure prematurely, when the weights are not finished adjusting. The main caveat with this approach is that it couples the tasks of optimizing the cost function and regulating the algorithm, making the process more complicated.

[3]Feature selection involves selecting a subset of relevant features to train on. Effects are: simplified model, shorter training times, reduced overfitting.

### 2.2.1 Setting up the optimization problem

#### Normalizing inputs

One measurement that can be taken to speed up the training, is normalizing the inputs. This consists of two steps; the first being to subtract the mean from the data, according to eq. (2.5), where $x$ is the training data, and $\mu$ is the mean.

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ x &:= x - \mu \end{aligned} \tag{2.5}$$

The second step is to normalize the variances, as shown in eq. (2.6), where $\sigma^2$ is the variance, and thus $\sigma$ is the standard deviation. $(x^{(i)})^2$ denotes element-wise multiplication. Note that the mean is already subtracted.

$$\begin{aligned} \sigma &= \frac{1}{m}\sum_{i=1}^{m} (x^{(i)})^2 \\ x &:= \frac{x}{\sigma} \end{aligned} \tag{2.6}$$

The complete normalization of the inputs is then described by eq. (2.7).

$$x := \frac{x - \mu}{\sigma} \tag{2.7}$$

It is important to use the same $\mu$ and $\sigma$ when normalizing the test set, as the training set and the test set should be subjected to the same normalization.

The purpose of normalizing the inputs is that it results in a more symmetric cost function, where each iteration yields a greater decrease. This is illustrated in fig. 2.6. The cost function is likely to have significantly more than three dimensions, but this is difficult to illustrate.

#### Vanishing/Exploding gradients

In a deep neural network, layers that are deep into the network are subjected to several, recurring matrix multiplications. If the weight matrices, and followingly the activation functions, take on large values, or simply values that are greater than 1, then the computed values will grow exponentially. This is reflected in backpropagation when computing the gradients, as they are made up of the derivatives of each layer multiplied together. If the derivatives are large ($> 1$), they will accumulate, and the gradients will increase exponentially, referred to as *exploding gradients*. Alternatively, if the derivatives are small ($< 1$), the gradients decrease exponentially, causing *vanishing gradients*. Exploding gradients result in an unstable network, incapable of learning effectively. Vanishing gradients can result in a model with so small gradients that the weights are effectively prevented from altering their values.

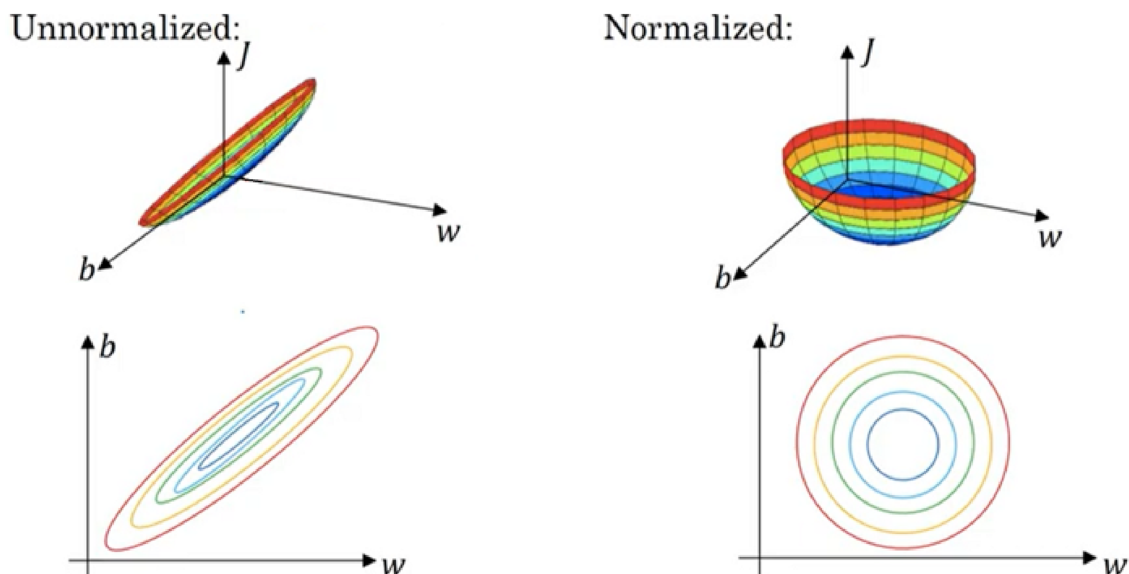



Figure 2.6: The cost function with unnormalized and normalized inputs, illustrated by Ng et al. [4].

A partial solution to this problem is formed through weight initialization heuristics. One such heuristic, which is especially effective for *tanh* activation functions, involves initializing the weights in accordance with eq. (2.8), where $U$ signifies a uniform distribution, and $n^{[l-1]}$ denotes the number of nodes in layer $l-1$, i.e. the number of inputs to layer $l$.

$$W_{ij} = U\left[-\frac{1}{\sqrt{n^{[l-1]}}}, \frac{1}{\sqrt{n^{[l-1]}}}\right] \tag{2.8}$$

Xavier initialization [28] is a modification to this heuristic, where the boundaries are changed, as in eq. (2.9).

$$W_{ij} = U\left[-\frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}, \frac{\sqrt{6}}{\sqrt{n^{[l-1]} + n^{[l]}}}\right] \tag{2.9}$$

He initialization [29] in eq. (2.10) is an alternative which works better for *ReLU*-like activation functions.

$$W_{ij} = U\left[-\frac{\sqrt{2}}{\sqrt{n^{[l-1]}}}, \frac{\sqrt{2}}{\sqrt{n^{[l-1]}}}\right] \tag{2.10}$$

These heuristics have proven to counter the problems of vanishing or exploding gradients. However, they are just heuristics and are not guaranteed to work, but may provide a good starting point.

*Note: Coursera* [4] *refers to eq.* (2.8) *as Xavier initialization, and also presents an alternative version of Xavier initialization that is similar to eq.* (2.9)*, but with* $\sqrt{2}$ *in the nominators instead of* $\sqrt{6}$*. The official paper by* Glorot and Bengio [28]*, however, refers to eq.* (2.8) *as a "common heuristic" and presents Xavier initialization ("normalized initialization", as it is referred to in the paper) as it is shown in eq.* (2.9).

### 2.2.2 Optimization algorithms

Applying machine learning is a highly empirical, and highly iterative process. To find a good model, one must usually train several and pick the best one. For this reason, it is highly beneficial that the training jobs proceed quickly, which can be a challenge in the context of big data. Having good optimization algorithms can significantly help speed up the training jobs.

#### Mini-batch gradient descent

Normal gradient descent iterates through the entire training set before making adjustments to the weights. If the training set is very large, this can be a slow process, since each iteration through the training set takes a long time. By dividing the training set into several *mini-batches*, and making adjustments to the weights after each mini-batches, the model will make more frequent progress, and the training procedure will finish more quickly. This technique is called mini-batch gradient descent. The following optimization algorithms are all usually implemented with the mini-batch characteristic.

#### Exponentially weighted averages

Exponentially weighted averages (EWA) is a key component in several optimization algorithms. It is a technique for analyzing data by creating a series of averages of different subsets of the full data set. These averages provide a smoother, less noisy curve, than the raw data points. The technique can be considered to capture general trends in how the data evolve, which can be beneficial for optimization. The weighted average at each time $t$, $V_t$ is the weighted average at the previous time $V_{t-1}$ times a factor $\beta$ added with the data point at the corresponding time, $\theta_t$ times $1-\beta$. The computation procedure is shown in eq. (2.11).

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t \tag{2.11}$$

The EWA represents the average over approximately the $\frac{1}{1-\beta}$ preceding data points. The higher the value for $\beta \in [0,1]$, the smoother the curve, but the slower the EWA is to adjust to new data.

The initial EWA, $V_0$ is not necessarily a good estimate and might cause a bias that leads to poor EWA. A technique to counter this is bias correction, shown in eq. (2.12). This helps the EWA, especially in the initial phase, since for large values of $t$ the $\beta^t$ term becomes negligible.

$$V_t := \frac{V_t}{1-\beta^t} \tag{2.12}$$

#### Gradient descent with momentum

The idea of gradient descent momentum is to compute an exponentially weighted average of the gradients, and then use this to update the weights and biases instead. This will lead to less drastic changes to the parameters, analogous to them having a type of momentum.

This can help to eliminate oscillations when approximating the local minimum of the cost function. The gradients and weight and bias updates are thus computed in accordance with eqs. (2.13) to (2.16).

$$V_{dw} := \beta V_{dw} + (1-\beta)dw \tag{2.13}$$

$$V_{db} := \beta V_{db} + (1-\beta)db \tag{2.14}$$

$$w := w - \alpha V_{dw} \tag{2.15}$$

$$b := b - \alpha V_{db} \tag{2.16}$$

Bias correction is usually not necessary in this context but can be applied if preferred.

### RMSprop (root mean square prop)

Like gradient descent with momentum, the goal of RMSprop is to dampen out oscillations and focus the descent towards the minimum. RMSprop also utilizes exponentially weighted averages, but a new notation $S_{dw}$ is used to distinguish from gradient descent with momentum. Equations (2.17) to (2.20) show how gradients, weights and biases are updated in RMSprop, where $dw^2, db^2$ signifies element-wise multiplication. The $\epsilon$ is added to the denominator to avoid dividing by very small numbers that cause exhaustive weight and bias updates. $\epsilon = 10^{-8}$ is a good default value [4, 30].

$$S_{dw} := \beta S_{dw} + (1-\beta)dw^2 \tag{2.17}$$

$$S_{db} := \beta S_{db} + (1-\beta)db^2 \tag{2.18}$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw} + \epsilon}} \tag{2.19}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}} \tag{2.20}$$

The intuition is that too-large gradients in sub-optimal directions causing oscillations will be dampened out since the learning rate is divided by a large number, while too-small gradients in the optimal direction will be magnified when divided by a smaller number.

### Adam optimization

The Adam optimization algorithm essentially combines gradient descent with momentum and RMSprop. Adam is an acronym for “Adaptive Movement Estimation”. Adam has

proven to be a widely successful algorithm and is perhaps the most popular one in the machine learning community. $V_{dw}$, $V_{db}$, $S_{dw}$ and $S_{db}$ are computed in accordance with eqs. (2.13), (2.14), (2.17) and (2.18). In order to distinguish between them, $\beta$ is substituted with $\beta_1$ and $\beta_2$ for gradient descent with momentum and RMSprop, respectively. For Adam, bias correction is usually implemented, giving eqs. (2.21) to (2.24), where $t$ denotes the number of iterations.

$$V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t} \tag{2.21}$$

$$V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t} \tag{2.22}$$

$$S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta_2^t} \tag{2.23}$$

$$S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t} \tag{2.24}$$

Finally, the weights and biases are updated with eqs. (2.25) and (2.26).

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}} \tag{2.25}$$

$$b := b - \alpha \frac{S_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}} \tag{2.26}$$

The values for the hyperparameters are suggested to be $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [30].

#### Learning rate decay

Another measure towards faster training procedures is to reduce the learning rate over time. This will maintain fast learning early in the training phase, but also help the algorithm converge when it is nearing the optimum. This can be implemented with eq. (2.27), where the initial learning rate $\alpha_0$ and the decay rate $\zeta$ are tunable hyperparameters. $n_{epoch}$ denotes what number epoch the training procedure is in, i.e. how many times it has passed through the training data.

$$\alpha = \frac{\alpha_0}{1 + \zeta n_{epoch}} \tag{2.27}$$

### 2.2.3 Hyperparameter tuning

As has become apparent by now, neural nets are largely dependent on hyperparameters. Choosing the correct hyperparameter values can be challenging, especially if the neural net

in question depends on a multitude of them. Some hyperparameters are more important than others. Ng et al. [4] names the learning rate *alpha* as the most important one, and then secondarily the number of hidden units, the mini-batch size, and the momentum term $\beta$ if gradient descent with momentum is used as the optimization algorithm. The hyperparameters specific to the Adam optimizer rarely need tuning. Naturally, the regularization parameter $\lambda$, and the dropout rate are also important if regularization is implemented.

A classical approach to hyperparameter tuning is setting up a selection of values for each hyperparameter in an $n$-dimensional grid, where $n$ is the number of hyperparameters. Each cell in this grid represents a unique combination of hyperparameter values. Training thus ensues with each of these combinations to find the best one. This approach is not that popular anymore as one end up wasting many training jobs on tuning hyperparameters that are less important, and additionally, the same values for important hyperparameters are tested multiple times. A better approach is in fact to sample random values.

When sampling at random, the best approach is not always to do so uniformly over the range of valid values. When searching for the number of hidden units to have in a layer, or the number of layers the neural network should have, uniform sampling over a linear scale might be a valid approach. However, in the case of the learning rate $\alpha$, for which $[0.0001, 1]$ is a reasonable range of values, a uniform search over a linear scale will yield $\sim 90\%$ of samples within the range $[0.1, 1]$. Using a logarithmic scale instead yields the same amount of samples in the ranges $[0.0001, 0.001]$, $[0.001, 0.01]$, $[0.01, 0.1]$, and $[0.1, 1]$, which is a much more reasonable approach. The scale on which values are picked is important to keep in mind when sampling hyperparameter values.

Hyperparameter optimization is discussed further in chapter 5.

When maintaining a machine learning system over time, the data might gradually change over time, or the algorithm might be subjected to small alterations. This might cause the best hyperparameter settings to get stale. Thus it is recommended to reevaluate the hyperparameters in regular intervals.

Ng et al. [4] mentions two major schools of thought for hyperparameter tuning. The first one is referred to as the panda approach, where one model is watched carefully over the course of several days. The learning curve is monitored continuously, and the hyperparameters are altered continually based on the observations made. This is a typical approach when computational resources are scarce. It is called the panda approach due to the similarity between how a panda reproduces - it has very few babies, and pay much attention to them.

The second approach is the caviar approach. In the caviar approach several models, with different hyperparameter settings, are trained in parallel, and the most promising one is chosen. This is an analogy to how fish reproduce, laying thousands of eggs, and has thus been named the caviar approach.

### 2.2.4 Batch normalization

One of the most important ideas in the rise of deep learning; *batch normalization*, is an algorithm created by Ioffe and Szegedy [31], to expedite the hyperparameter search problem and make the neural network more robust. As discussed in section 2.2, normalizing the inputs can alter the contour of the cost function so that training proceeds faster. Batch normalization applies this concept to deep neural networks. As the parameters of layers change during training, the distribution of the inputs to the succeeding layer changes. Ioffe

and Szegedy [31] refer to this phenomenon as *internal covariance shift* and explain that it slows down the training process by requiring lower learning rates and careful parameter initialization, and causes models with saturating nonlinearities to be notoriously difficult to train.

The idea behind the algorithm is to normalize layer inputs and integrate the process of doing so as part of the model architecture. Normalization is performed for each mini-batch. Applying this algorithm allows for much higher learning rates and less careful initialization, and, in some cases, provides a degree of regularization. The normalization addresses the internal covariance shift, rendering the learnable parameters less susceptible to extreme alterations when faced with input data containing unfamiliar characteristics, which in turn allows for higher learning rates and reduces overfitting.

It is possible to normalize either $z_i$, the value before the activation function, or $a_i$, the value after the activation function, which is the actual input to the succeeding layer. In practice, normalizing $z_i$ is done more often, and is the approach recommended by Ng et al. [4].

Similarly to when normalizing inputs eqs. (2.5) to (2.7), the outputs of all hidden units before passing through the activation function, $z^{[l](i)}, i \in [1, m]$, where $m$ denotes the number of hidden units in layer $l$, are normalized with zero mean and unity variance, as in eqs. (2.28) to (2.30). $l$, denoting the layer, is from here on omitted for brevity. In eq. (2.30) $\epsilon$ is added to the denominator in case $\sigma = 0$.

$$\mu = \frac{1}{m} \sum_{i}^{m} z^{(i)} \tag{2.28}$$

$$\sigma^2 = \frac{1}{m} \sum_{i}^{m} (z^{(i)} - \mu)^2 \tag{2.29}$$

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{2.30}$$

Often a different distribution is desired for hidden units. If the activation function for the respective hidden unit is a sigmoid function, one might desire a larger variance or a different mean to better utilize the nonlinearity of the sigmoid function, instead of limiting the range to the linear regime. Thus $\tilde{z}^{(i)}$ is computed from eq. (2.31), where $\gamma$ and $\beta$ are learnable parameters, that control the mean and variance of the hidden unit values.

$$\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta \tag{2.31}$$

The $\tilde{z}^{(i)}$s are then substituted for the $z^{(i)}$s for the succeeding computations in the neural network.

One thing to note about batch normalization is that, due to eq. (2.28), any bias $b^{(i)}$ added to $z^{(i)}$ is removed, and might as well be omitted or valued to zero. It is in a way replaced by $\beta$, which adds a constant to the resulting $\tilde{z}^{(i)}$.

At test time, one might need to process one example at a time, instead of mini-batches. This provides a problem in terms of $\mu$ and $\sigma^2$ since it makes no sense to compute these from one example. The solution to this problem is to estimate values for $\mu$ and $\sigma^2$ for each

layer, using the computed values for the respective layers during the training phase. The most common way to do this is by computing an exponentially weighted average for $\mu$ and $\sigma^2$ for each layer, based on the $\mu$ and $\sigma^2$ for computed each mini-batch on the respective layer during training.

## 2.3 Course 3: Structuring Machine Learning Projects

Effective means for structuring machine learning projects are beneficial for operationalization, providing clarity and resulting in effort more efficiently being translated into progress.

A challenge with deep learning is that when a reasonably good, but not sufficient model has been achieved, it is difficult to figure out what to change to further improve the model. This challenge can result in entire teams spending much time on changes that ultimately have no or little impact, effectively laying all the work to waste. To cope with this, there are some strategies for analyzing the problem and identifying the most promising measures for improving the model.

### 2.3.1 Orthogonalization

The concept of orthogonalization involves obtaining a clear overview of what to tune to achieve a certain effect. It is derived from orthogonality, e.g. orthogonal vectors, which have a dot product of zero, meaning that they affect completely separate features in a system. Achieving orthogonalization allows for a systematic tuning process, much more likely to translate effort into progress.

Say that a model is performing below par. Instead of assuming that collecting more training data will increase the performance, it is wise to take a closer look at the model, to try and pinpoint exactly what needs to be improved.

Investigating how well the algorithm performs on the training set is a good start. If the algorithm does not fit well with the training data, then adding more data is not likely to effectively solve the problem. Adding more hidden units and/or layers to the neural network might enable the algorithm to identify more characteristics in the training data, thus increasing training set performance. Another approach could be to switch to a different optimization algorithm.

If the problem lies not with the test set, a natural next step is to investigate the validation set performance. If this is not satisfactory, tuning the regularization parameters could help increase performance. Alternatively, collecting more training data is a valid course of action for improving validation set performance.

If the algorithm performs well on the validation set, but badly on the test set, a likely reason is that the algorithm has overfitted the validation set, and a solution might be to increase the size of the validation set.

Finally, if the algorithm performs well on the training, validation, and test sets, but is not delivering good results for real-world data, then likely the distribution for the validation and test sets is not representative enough, or the cost function is not well-defined.

These steps form a systematic approach, where problems are closely examined to deduce the most promising courses of action.

Early stopping is a technique that aligns badly with orthogonalization. Early stopping affects how well the algorithm fits the training set, while simultaneously being used to improve validation set performance, thus affecting two things.

### 2.3.2 Setting up the goal

Defining concise goals allows for more efficient and frequent evaluations of model performance. This is auspicious in an operationalized system, where models require regular updates.

#### Single number evaluation metric

When evaluating model performance, several metrics provide different insights. This is beneficial in terms of achieving a clearer picture of the model, but it poses problems when comparing different models, some models can score high on some metrics and low on others. Creating a single metric that combines several metrics in a weighted manner that makes sense for the model offers more easily accessible insights into the models, and makes it simpler to compare them. Typical evaluation metrics, as well as some combinations, are discussed in chapter 5.

#### Satisficing and optimizing metric

Combining several metrics into one metric can be difficult. A technique that can be used is distinguishing between satisficing and optimizing metrics. Optimizing metrics are the metrics that should be optimized, the ones where a high a score as possible is desired, such as accuracy. Satisficing metrics are metrics that only need to reach a certain threshold, and are not considered much beyond this threshold. An example of a satisficing metric is the run time of a classifier; as long as the classifier finishes within a certain time period, e.g. 100 milliseconds, it is not necessary to make it run faster.

### 2.3.3 Error analysis

Error analysis involves examining the misclassifications or erroneous predictions made by a model to understand what characteristics the algorithm struggles with. E.g., a classifier for cat pictures has 90% accuracy, and it turns out that out of the 10% misclassified pictures, 70% of them are of dogs. This might motivate working on a solution for distinguishing especially between dogs and cats. On the contrary, if 2% of the misclassified pictures are actually raccoons, then focusing on a solution that distinguishes between raccoons and cats, is perhaps not worth the effort.

#### Cleaning up incorrectly labelled data

Sometimes data can be incorrectly labelled, which naturally has a bad impact on the algorithm. However machine learning algorithms are relatively robust to random errors, so long as only a small percentage of the training set is incorrectly labelled, and thus cleaning up the errors might not be worth the effort. Systematic errors, e.g. consequently labelling images of dogs as images of cats, are worse and should be cleaned up.

### 2.3.4 Data set distributions

It is imperative that the validation and test sets are from the same distribution. If they are not, the algorithm is prone to optimize on false premises, which will lead to poor performance on the test set.

#### Training and testing on different distributions

There are scenarios when the application subject to development is particularly specific, yielding much training data that is related to but not within the primary domain of interest. Imagine a speech recognition system for a music application. Such a system must recognize a lot of song names, and names of bands and artists, as well as commands for playing, pausing, skipping, etc. The amount of speech data in this domain may be limited, while the amount of general speech data is much larger. In this case, it can be beneficial to use all speech data available, and not just the data from the music application domain.

In such a situation it is important that the data from outside the domain is only used in the training set, and not in the validation or test set. Using it in the training set is likely to improve how well the algorithm recognizes different characteristics, which is positive. Using the data in the validation or test set, however, will lead the algorithm to optimize on false premises, and not optimize for the actual implementation.

#### Bias and variance with mismatched data distributions

As discussed in section 2.2, comparing the error on the training set with the error on the validation set can provide insight into whether the algorithm is suffering from bias or variance. These assessments can not be made when the training and validation sets are from different distributions. To address this, a new data set is created from the training set, called the training-validation set. The training-validation set is not explicitly trained on but comes from the same distribution as the training set. Comparing the error on the training set with the error on the training-validation set can now reveal bias or variance. Comparing the training and training-validation sets with the validation set can reveal something else; if the errors differ largely, this indicates data mismatch, signifying that the algorithm is trained well, but for the wrong distribution.

To address data mismatch, it is possible to conduct error analysis and pinpoint the specifics of how the data in the validation and test sets differ from the data in the training set. When the differences are identified, it is possible to conduct artificial data synthesis to perform alterations on the training data, making it more similar to the validation and test data. Taking the example of speech recognition for a music application, it could be that the data in the validation and test set have much background noise, while the training data does not. Sampling similar background noise and synthesizing this with the training data might help to solve this problem.

### 2.3.5 Learning from multiple tasks

#### Transfer learning

Transfer learning is when knowledge that a neural network has gained from solving one task can be applied to solve different tasks. This is a very powerful tool in deep learning. This can be done by taking a trained neural network, removing the last layer which produces the output and the weights that are fed into this layer, and replacing them with a new layer and a new set of initialized weights. The modified network can now be trained on a much smaller data set for the new application. Optionally, the already trained weights can be frozen, so that they are not adjusted by the new training data. A rather humorous and quite magnificent example of transfer learning comes from Japan, where a neural net created to identify pastries turned out to be able to recognize cancer cells [32].

#### Multi-task learning

Multi-task learning is when a neural network is designed to perform multiple tasks, i.e. predict multiple things. When the tasks have the same input and share some low-level features this has its advantages, as compared to training separate neural networks for each task. An example is a self-driving car that needs image recognition for deciding when to stop. There are many reasons why a car should stop; there could be a stop sign, a red traffic light, a pedestrian, etc. The images containing these objects can be very similar in structure; they are likely to contain the sky, other traffic signs, cars, road markings, etc., and some images could contain multiple objects that should trigger a stop signal. A neural net can then be trained to output several predictions on whether a relevant object is in the picture or not.

### 2.3.6 End-to-end deep learning

Many data processing or learning systems consist of multiple stages. The idea of end-to-end deep learning is these stages with just a single neural network. Traditionally, speech recognition required many stages; first a processing algorithm for extracting low-level features of audio, then a machine learning algorithm to identify phonemes[4] in the audio clip. Then the phonemes are combined to form words, which in turn are combined to form transcripts. An end-to-end approach for speech recognition will take the original audio clip as the input and output the transcripts, with a single neural network. End-to-end deep learning is a pure machine learning approach. Allowing the algorithm to fully explore and capture the characteristics and statistics of the data, rather than being forced to reflect human perceptions, such as phonemes, might yield better performance. Eliminating hand-designed components will also simplify the system. This is a double-edged sword, however, since potentially useful hand-designed components are excluded. Another caveat is that end-to-end deep learning requires vast amounts of data to fulfil its potential.

[4] Phonemes are the “basic units” of sound.

## 2.4 Course 4: Convolutional Neural Networks (CNNs)

### Computer vision

Computer vision has gained a lot of traction with the rise of deep learning. Computer vision is used in applications such as facial recognition, which is widely used for unlocking phones, and object detection, used in self-driving cars, to mention a few. Due to the rapid development in this area, the computer vision research community has been inventive and creative in terms of forming new architectures and algorithms for neural networks. These have been found to be applicable in many other areas of research as well.

One of the challenges in computer vision that have fueled innovation is that the inputs can be really big. A 1000 by 1000 pixel image results in a $3 \times 1000 \times 1000 = 3,000,000$ dimensional input. The dimension is multiplied by 3 because each pixel must be represented in 3 *channels* to incorporate the RGB values of each pixel. A fully connected layer with 1000 hidden units yields a 1000 by 3.000.000 dimensional matrix, which is incredibly large. With that many parameters, it is difficult to collect enough data to avoid overfitting. Moreover, the computational requirements of such a network are immense. A fully connected layer is a layer in which every neuron is connected to every neuron in the preceding layer. An alternative to this, which has emerged from computer vision research, is the *convolutional layer*.

What a convolutional layer does is that it takes the input tensor[5] and convolves it with a *filter* to produce a *feature map*, which makes up the next layer. The filter, also sometimes referred to as the *kernel*, is a tensor of lower dimension than the input, constructed in a way that extracts the features of interest. Traditionally, they are constructed by hand, but they can be learned as well. The convolution operation demands far fewer connections between layers and is thus less computationally expensive, and it allows for the extraction of general features without overfitting. Figure 2.7 illustrates how input is convolved with a filter to produce a feature map. Asterisk, $*$ is used to denote convolution. All elements in the filter are element-wise multiplied with the elements in an overlapping region in the input. These products are then summed up and stored in the corresponding region in the output/result. The region is then shifted one place to the right, and the process is repeated. After the rightmost region, the process starts again from the left, shifted one place down. This is repeated until the region in the bottom right corner is covered.

### Edge detection using convolution

Edges in an image are features that are commonly detected using convolution. For this purpose, the filter design is, in fact, rather intuitive. Imagine a matrix with pixel values for a grey-scale image, where the lower the value, the darker the pixel. fig. 2.8 illustrates how a vertical edge detection-filter convoluted with the original image produces a feature map where an edge is detected.

### Padding

A downside with convolution is that every time it is applied, the resulting feature map has lower dimensions than the input, i.e. the image shrinks. The dimension of the output,

[5] A tensor is an umbrella term of which specific instances include scalars, vectors, and matrices.

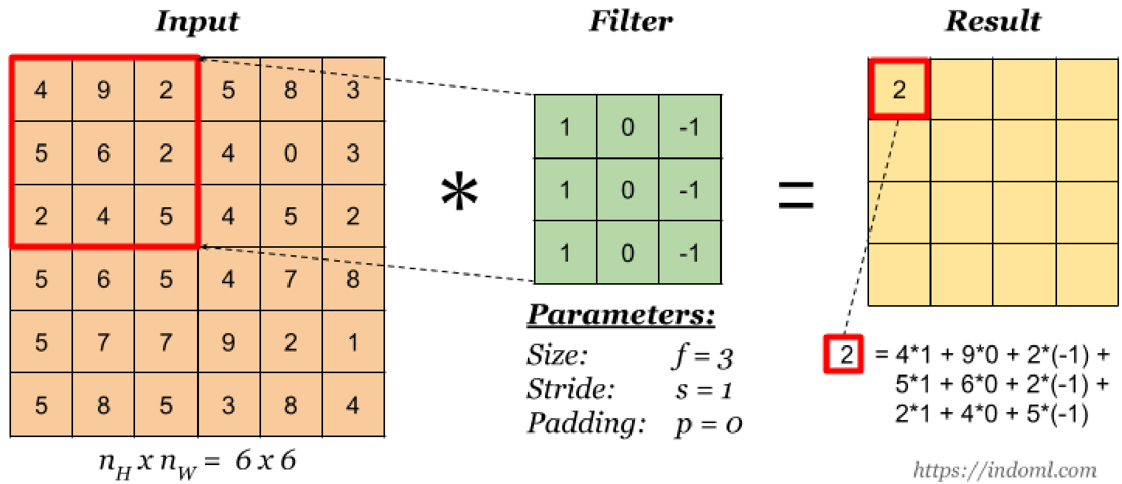



Figure 2.7: The convolution operation, as illustrated by Prijono [33].

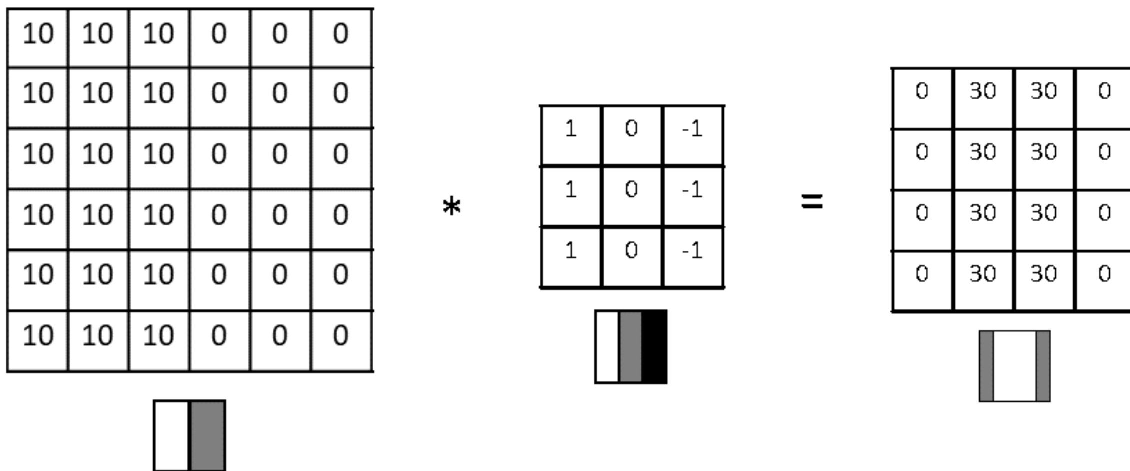



Figure 2.8: Vertical edge detection using convolution, as originally illustrated by Ng et al. [4], and revised by datahacker.rs [34].

where the input is an $n \times n$ matrix, and the filter a $f \times f$ matrix, is $(n-f+1) \times (n-f+1)$, given by eq. (2.32)

$$Dim(X * f) = (n - f + 1) \times (n - f + 1) \\ \text{, where } X \in R^{n \times n}, f \in R^{f \times f} \tag{2.32}$$

Another caveat is that the edge pixels in an image have much less impact on the output than what the middle pixels, which are iterated over multiple times, have. *Padding* is a concept that addresses both these problems. It involves adding a border - conventionally consisting of 0s, also called *zero-padding* - around the input. Figure 2.9 illustrates a case where padding with a border of size $p = 1$ is applied. As seen, the output has the same dimensions as the original input. This is referred to as *same* convolution. Convolution with no padding is called *valid* convolution.

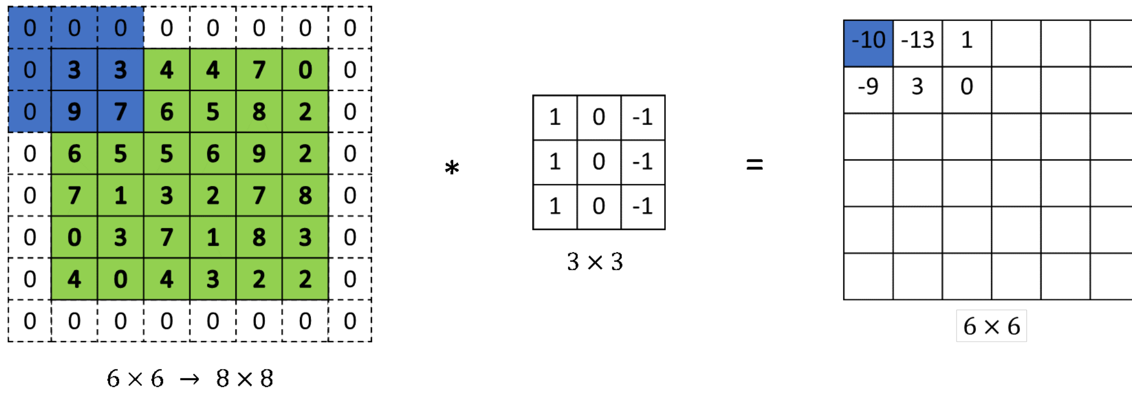



Figure 2.9: Padding of size $p = 1$ applied to the input of a convolution operation, originally illustrated by Ng et al. [4], and revised by datahacker.rs [34].

### Strided convolutions

It is possible to adjust the step size, or the *stride* in a convolution operation. The stride, $s$ denotes how many places the region is shifted after each multiplication-summation procedure. A higher stride results in a lower-dimensional output. Equation (2.33) shows the dimension of the output, taking both padding and stride into consideration.

$$Dim(X * f) = \lfloor \frac{n + 2p - f}{s} + 1 \rfloor \times \lfloor \frac{n + 2p - f}{s} + 1 \rfloor \\ \text{, where } X \in R^{n \times n}, f \in R^{f \times f} \tag{2.33}$$

### Pooling layers

An alternative, but somewhat similar operation to convolution is the *pooling* operation. There are two types; *max pooling* and *average pooling*. Max pooling is used the most - average pooling is quite rarely applied. In pooling, the filter contains no elements but applies a max-operation or an average-operation to the region. Thus, a pooling operation contains no learnable parameters, just the hyperparameters $f$ and $s$ for filter size and stride, respectively. It is possible to apply padding also in pooling layers, but this is rarely done. Figure 2.10 illustrates the two pooling operations. Equation (2.33) also applies to the output dimensions of a pooling operation.

The intuition behind max pooling is that it detects features in different locations of the input (image). A high number in a region signifies that there is a feature there, and this information is conveyed deeper into the network. If there is no particular feature, then the highest number will be relatively low. There is discussion whether this intuition really aligns with reality. Nonetheless, max pooling has proven to work well in a lot of experiments.

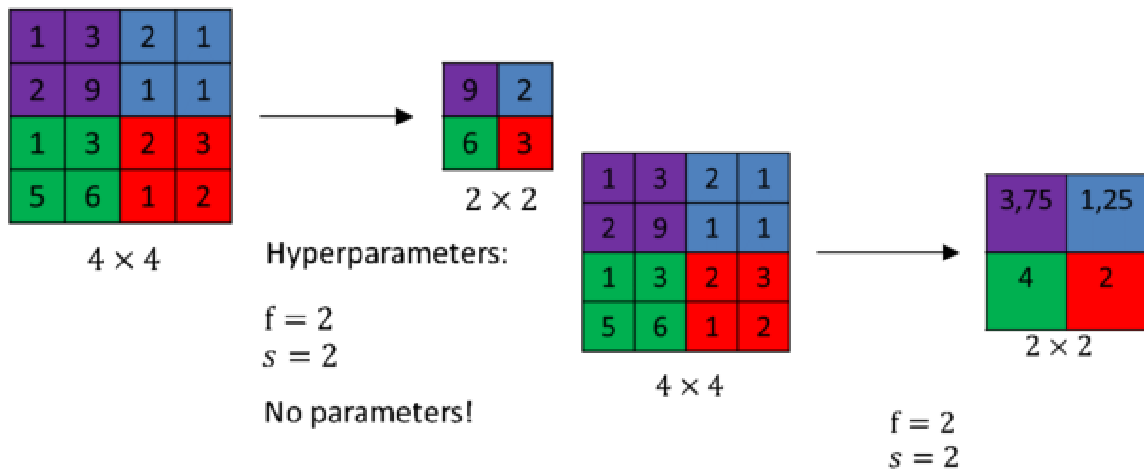



Figure 2.10: To the left: max pooling, to the right: average pooling. Originally illustrated by Ng et al. [4], and revised by datahacker.rs [34].

### 2.4.1 Examples of efficient convolutional network architectures

#### ResNets

Deep residual nets, or ResNets, are the result of an architecture presented by He et al. [35]. As networks grow deeper, they become harder to train, due to challenges such as vanishing and exploding gradients. The motivation behind ResNets is to facilitate the training of even deeper neural networks without degrading training performance. They are based on the notion of *skip connections* and *residual blocks*, fig. 2.11. A skip connection is an activation that is passed on to a layer that is even deeper than the layer directly following the activation. The block into which this activation is passed is a residual block.

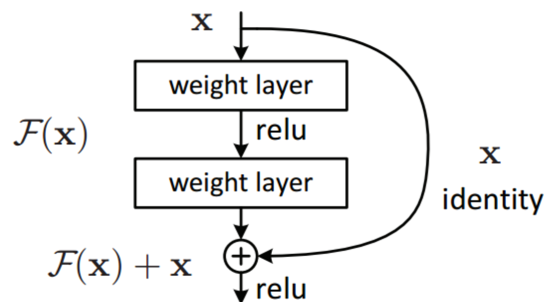



Figure 2.11: A skip connection into a residual block, as illustrated by He et al. [35].

ResNets have been proven successful in improving the training of very deep neural networks. Figure 2.12 shows as ResNet and a plain neural net.

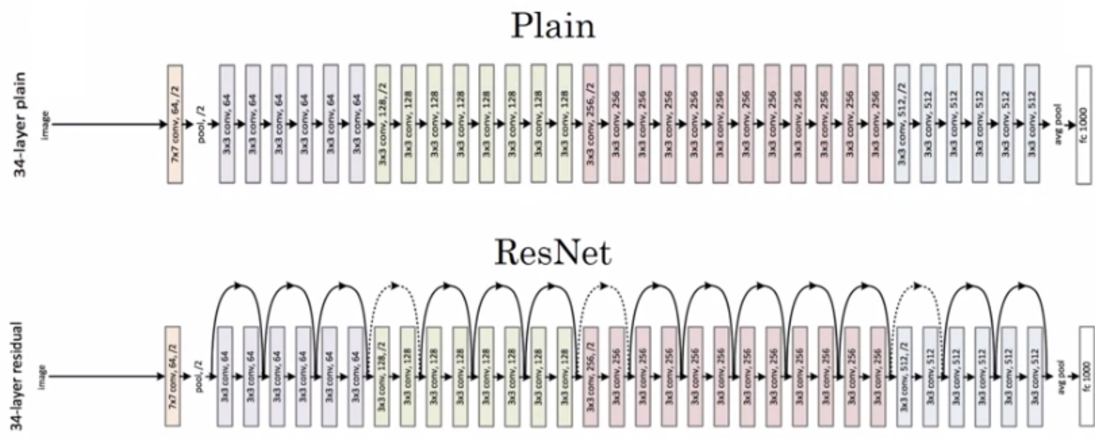



Figure 2.12: A plain neural net and a ResNet. Illustrated by He et al. [35].

### Inception Network

Inception is a deep convolutional neural network architecture proposed by Szegedy et al. [36]. The inspiration behind the name is actually the famous "we need to go deeper" internet meme [37], which is listed as the first reference in [36].

Raj [38] mentions some problems that the ResNet architecture addresses. Images in training sets can vary with respect to the position of, and the area covered by the object of interest. This makes the choice of filter size for the convolution operation hard, as larger filter sizes are in general preferred for information that is distributed globally over the image, while smaller filter sizes are preferred for more locally concentrated information. Stacking large convolutional operations is computationally expensive, and results in very deep networks which are prone to overfitting. While improved utilization of the computing resources inside the network is the main hallmark of the Inception architecture, it impacts the other problems as well.

The Inception architecture involves performing multiple operations in parallel on the same layer. This produces a network that is, in a sense, wider, as opposed to deeper. Several iterations of the architecture are presented in the paper, and more in later papers, but to convey the general structure fig. 2.13 illustrates the naïve version of the Inception module.

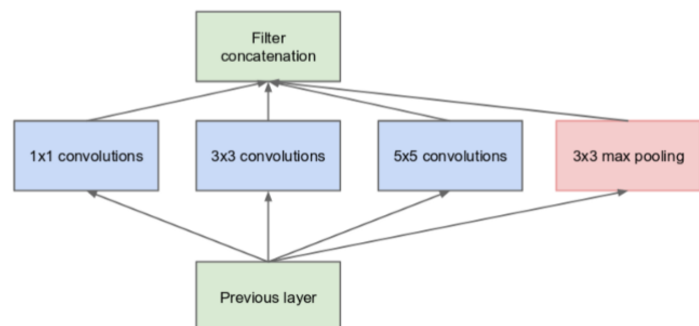



Figure 2.13: The naïve version of the Inception module, as illustrated by Szegedy et al. [36].

The actual network developed in Szegedy et al. [36] is called GoogLeNet, and is shown in fig. 1 in appendix B.

## 2.5 Course 5: Sequence Models

Sequence models are models that deal with sequence data, e.g. speech recognition, where an input audio clip is mapped to a text transcript, or sentiment classification, where the input is a phrase, and the output classifies the sentiment in the phrase. In the former, both the input and the output are sequences, while in the latter only the input is a sequence. Other examples are music generation; DNA sequence analysis; video activity recognition; and more.

*Note: This section does not deal with convolution, and* $*$ *now denotes element-wise multiplication.*

### Recurrent Neural Networks (RNNs)

Recurrent neural networks have had a great impact on areas such as speech recognition and natural language processing.

A problem with using standard neural networks for sequence models is that not all data examples of sequences have the same length, and the output sequences can also vary in length. To cope with this in a standard neural network, a maximum size must be set, and unused space filled with zeros or some sort of null value. This is not a good solution. Another problem with a naïve neural network architecture is the inability to share features learned across different positions in a sequence. If a model for name detection detects a name early in a sentence, this should weigh in on its ability to detect the same name later in the same sentence. Recurrent neural networks solve these problems. An RNN is illustrated in fig. 2.14. $T_x$ and $T_y$ are used to denote the input and output sequence length, respectively. The $< t >$-superscript is used to denote the $t$-th element in a sequence, so $x^{<t>}, t \in T_x$ are all elements in the same example.

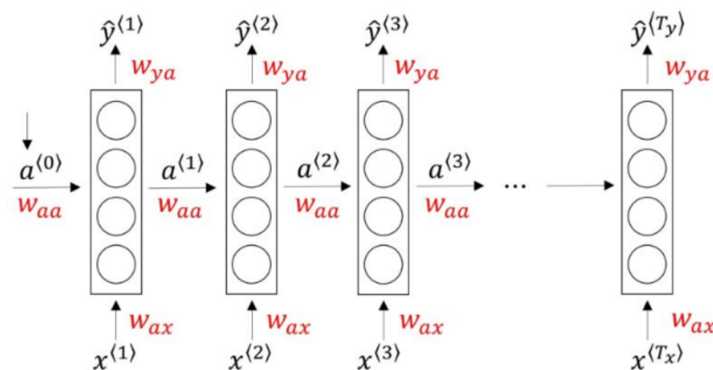



Figure 2.14: Simple representation of RNN architecture. Illustrated by Ng et al. [4], and revised by [34].

As shown in fig. 2.14, the activation for each element is passed as an input to the prediction for the next element. This addresses the second problem with standard neural networks, enabling the RNN to use features learned from previous elements. The initial activation

function $a^{<0>}$ is normally simply a vector of zeros. The architecture in fig. 2.14 also allows for varying sizes of inputs and outputs between examples but requires each input-output pair to be of equal length.

Forward propagation ensues with eqs. (2.34) and (2.35): one activation is passed on to the next element, and a second internal activation function computes the output for each element. The activation that is passed on need to be accounted for in backpropagation, and that is why backpropagation for RNNs is sometimes referred to as *backpropagation through time*.

$$a^{<t>} = g_a^{<t>}(w_{aa} * a^{<t-1>} + w_{ax} * x^{<t>} + b_a) \tag{2.34}$$

$$\hat{y}^{<t>} = g_y^{<t>}(w_{ya} * a^{<t>} + b_y) \tag{2.35}$$

This architecture only enables later elements to use features learned from previous elements, and not vice versa. *Bidirectional recurrent networks* (BRNNs) address this later in this section.

### Different RNN architectures

There are four possible input-output size combinations:

- Many-to-many
- Many-to-one
- One-to-many
- One-to-one

Figure 2.14 is an example of a many-to-many RNN, but with the specific case where $T_x = T_y$. In a machine translation algorithm, where a sentence in one language is translated into a different language, for example, usually $T_x \neq T_y$. In such a case the neural network is separated into two parts; an encoder that "reads and understands" the sentence, and a decoder that translates the sentence into another language. This is illustrated in fig. 2.15.

A one-to-one RNN is just a standard neural network. Many-to-one and one-to-many, as are the cases for sentiment analysis and music generation, respectively, are illustrated in fig. 2.16.

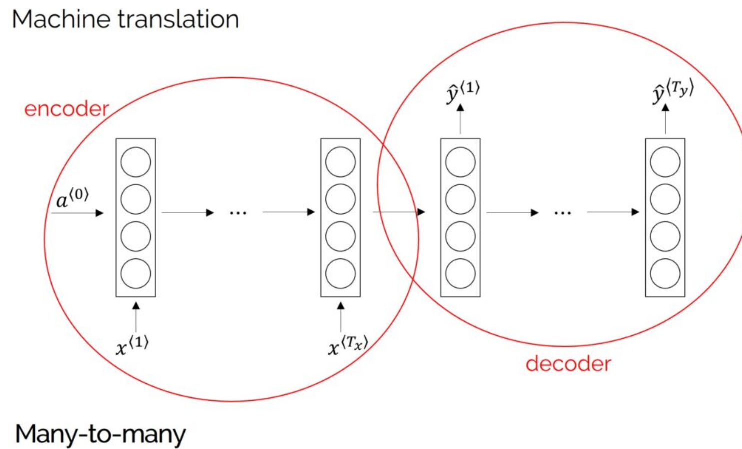



Figure 2.15: A many-to-many RNN separated into an encoding part and a decoding part. Illustrated by datahacker.rs [34]

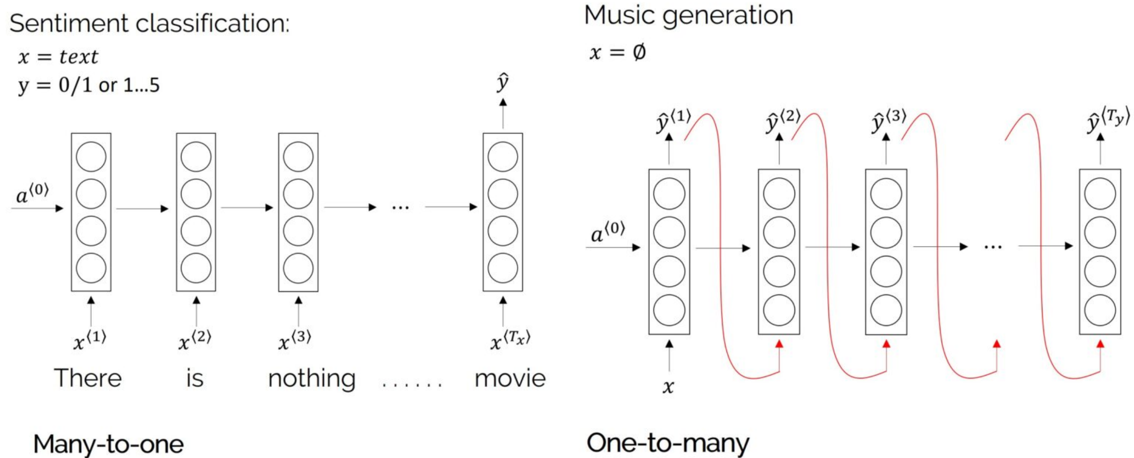



Figure 2.16: To the left: a many-to-one RNN architecture. To the right: a one-to-many RNN architecture. Illustrated by datahacker.rs [34].

### Language models

For natural language processing systems, a language model is usually applied to improve performance. A language model estimates the probability of a word appearing in a sentence, depending on how common it is and what word is preceding it. It is built by analyzing large amounts of text in the language, and optionally in the domain, in which the natural language processing system operates.

### Vanishing gradients with RNNs

With sequence data, elements in the same sequence, but far away from each other can have relations that are important for the output. This is common in natural language processing. Whether a noun is singular or plural affects the verb form to be used in the sentence. However, there can be many elements between the appearances of the noun and the verb. As a result, RNNs are particularly susceptible to vanishing gradients during the backpropagation of a sequence, because this decreases the effect elements have on other

elements that are not in close vicinity. Following are brief explanations for two solutions to this, *Gated Recurrent Unit* and *Long Short Term Memory*.

### Gated Recurrent Unit (GRU)

GRUs are modified hidden units in an RNN, intended to help the model in capturing long-term dependencies, such as mentioned above. The idea behind GRU is to store the activation of an element and pass it on for later use. this is done by adding a memory cell $c^{<t>}$, a candidate value $\tilde{c}^{<t>}$, and an update gate $\Gamma_u$ to each unit in the layer in the RNN. $c^{<t>}$ stores the memory of whether an element is single or plural, and passes it on to the next unit. In a GRU, the memory cell and the activation is the same signal; $c^{<t>} = a^{<t>}$. $\Gamma_u$ decides whether the stored memory, received from an earlier unit, should be applied to this element, i.e. whether the verb should be in singular or plural form. The purpose of $\tilde{c}^{<t>}$ is to evaluate whether the memory received from the previous unit should be replaced.

$\Gamma_u$ is computed by a sigmoid function. In the simplified GRU; if $\Gamma_u \approx 1$, the memory cell value $c^{<t>}$ is replaced by the candidate value $\tilde{c}^{<t>}$. If $\Gamma_u \approx 0$, the cell value $c^{<t>}$ is passed on to the next unit.

In a full GRU, another gate is added; the relevance gate, $\Gamma_r$. This modifies the computation of $\tilde{c}^{<t>}$, by adding a factor of how relevant the potential new memory is.

A GRU is implemented with eqs. (2.36) to (2.39).

$$\tilde{c}^{<t>} = tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \tag{2.36}$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \tag{2.37}$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \tag{2.38}$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \tag{2.39}$$

### Long Short Term Memory (LSTM)

LSTM is another unit for learning long-range connections in a sequence and is even more powerful than the GRU. LSTM was developed by Hochreiter and Schmidhuber [39]. In LSTMs, the memory cell and the activation are no longer the same signal, so $c^{<t>} \neq a^{<t>}$. It is possible to use a relevance gate, $\Gamma_r$ for LSTM, but it is rarely done. Instead, two new gates are added: the forget gate, $\Gamma_f$, and the output gate $\Gamma_o$. An LSTM is governed by eqs. (2.40) to (2.45).

$$\tilde{c}^{<t>} = tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \tag{2.40}$$

$$\Gamma_f = \sigma(W_f[c^{<t-1>}, x^{<t>}] + b_f) \tag{2.41}$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \quad (2.42)$$

$$\Gamma_o = \sigma(W_o[c^{<t-1>}, x^{<t>}] + b_o) \quad (2.43)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \quad (2.44)$$

$$a^{<t>} = \Gamma_o * tanh(c^{<t>}) \quad (2.45)$$

As seen in both GRUs and LSTMs, the gates are implemented quite similarly, mainly differing in their respective weights and biases. This is because they are intended to represent a binary output, thus the name "gate", of some property posed by the combination of the input $x^{<t>}$, and the memory cell $c^{<t-1>}$ or activation $a^{<t-1>}$ received from the previous layer. Because the memory cell and the activation is no longer the same signal, two separate gates are needed for computation; $\Gamma_u$ and $\Gamma_o$. Another thing to notice is the introduction of a separate forget gate $\Gamma_f$, replacing the $(1 - \Gamma_u)$ term from the GRU. Figures 2.17 to 2.19 illustrates a normal RNN unit, a GRU, and an LSTM. The relevance gate $\Gamma_r$ is omitted for simplicity.

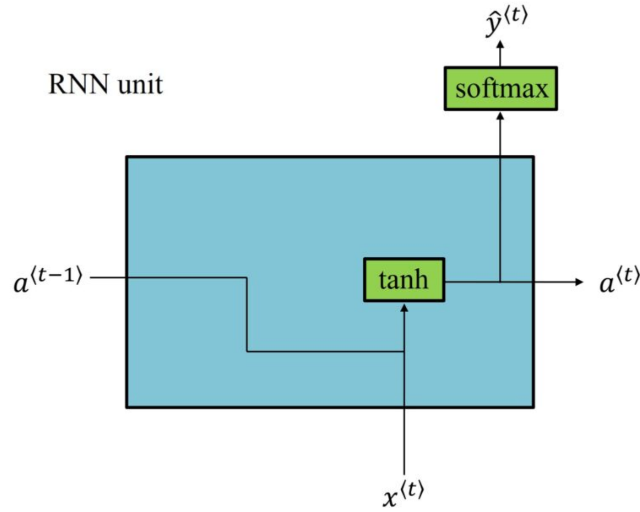



Figure 2.17: Normal RNN unit. Illustrated by datahacker.rs [34].

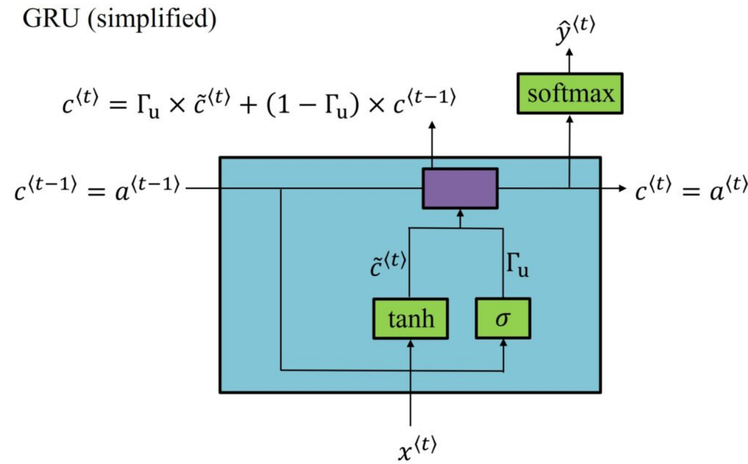



Figure 2.18: GRU without $\Gamma_r$. Illustrated by datahacker.rs [34].

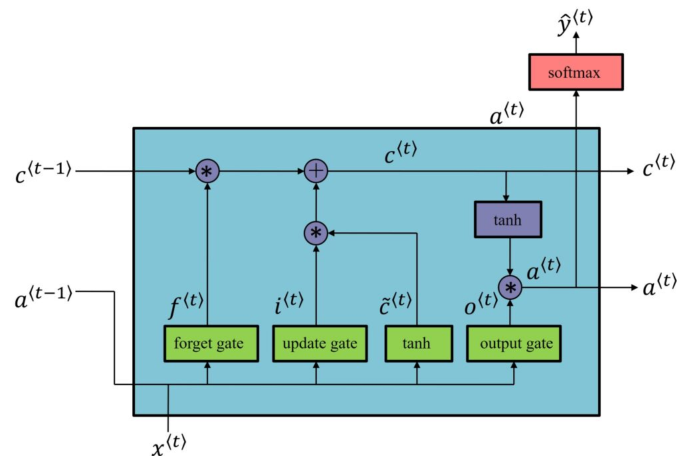



Figure 2.19: LSTM without $\Gamma_r$. Illustrated by datahacker.rs [34].

Both GRU and LSTM aim to achieve a form of memorization to apply learned features to other elements regardless of distance. LSTM is more comprehensive than GRU, and in general more powerful. It is often the default choice for the first iteration of a development cycle. However, the GRU is a simpler model, making it easier to build a big network, and it runs a bit faster than LSTM computationally. There is no widespread consensus on which is the best choice, as they each win out on different problems.

### Bidirectional RNNs

A common architecture for sequence models, especially for natural language processing, is the bidirectional recurrent neural network (BRNN). As mentioned previously, RNNs enables the usage of learned features from previous elements, but not from future elements - they are unidirectional. A BRNN adds connections from the later units to the earlier units in a "backward layer", as shown in fig. 2.20. This connection enables units to apply information from later units, before asserting their prediction. How this is done is not investigated in further detail in this thesis, but is covered by Schuster and Paliwal [40].

BRNNs are commonly combined with LSTM, but can also be implemented with GRU or a regular RNN unit. An example of where BRNN can be useful is in name entity recognition.

Given the sentence “Bear Grylls enjoys nature”, upon analysing the first word “Bear”, the algorithm has no way of telling whether it is a name or not. The sentence could turn out to be "Bear with me for a second", in which case “Bear” would not be a name. Without knowledge about the rest of the sentence, the algorithm lacks the preconditions for making the correct prediction.

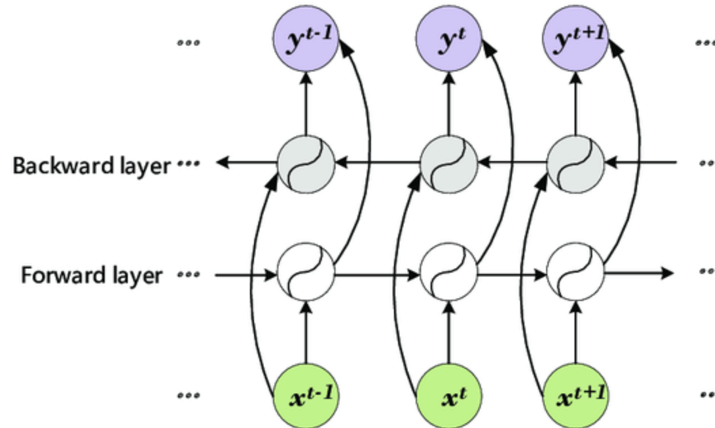



Figure 2.20: A BRNN with separate backward and forward layer. Illustrated by Mei et al. [41].

### Deep RNNs

The illustrations and explanations of RNNs so far have all considered a one-layer network for simplicity. Naturally, RNNs are usually applied in a deep sense, like in fig. 2.21.

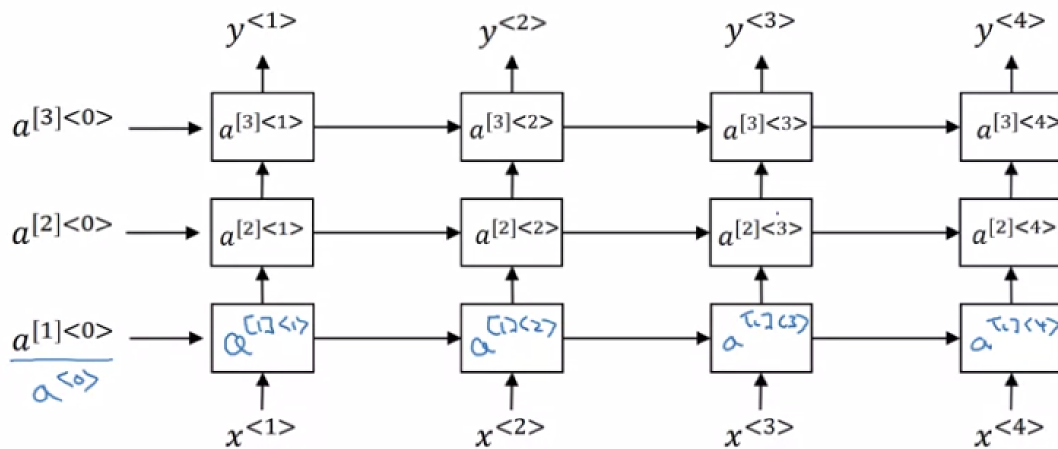


Figure 2.21: High-level overview of a deep RNN, as illustrated by [4]

# Chapter 3

# Techniques for Testing Machine Learning Systems

Reliable methods for testing are important to ensure a correctly functioning system and to sustain rapid development and, by extension, automated processes. This is necessary to maintain an operationalized machine learning system. Testing machine learning systems presents challenges because conventional software testing techniques do not always apply. The stochasticity of machine learning systems poses difficulties in terms of defining a reliable test oracle[1]. Their dynamic nature requires frequent testing, making effective testing methods desirable. Noisy data, non-convex objectives, model miss-specification, and numerical instability can all cause undesired behaviours in machine learning systems [43], making it difficult to pinpoint the origin of a bug. There is an important distinction between evaluating and testing models; evaluating involves measuring the predictive performance of a model with respect to relevant metrics. Testing involves revealing bugs in the system and ensuring that the learned logic is as intended.

This chapter presents what value traditional software testing techniques and data checks offer machine learning systems. Further, a proven approach for behavioural testing of NLP models is presented and, to highlight the extensivity of the challenge, a selection of methods based on different approaches with various prevalence are reviewed.

## 3.1 Conventional Software Testing

Machine learning systems introduce new and more complex requirements in terms of testing, but that doesn't mean that the need for traditional software testing disappears. Machine learning systems consist of several components that deal only with traditional software and/or data, and are not affected by the machine learning algorithm itself. Most components require standard software testing.

Mohandas [44] and Jordan [45] combine to form a typical software testing suite:

- **Unit tests**: tests on individual components that each have a single responsibility. Atomicity is important in this context; units must have a single responsible so that

[1] An oracle is a mechanism for determining whether a program has passed or failed a test [42].

they are easily testable.

- **Integration tests**: tests the combined functionality of individual components, i.e. how components integrate with each other. These tests typically take longer to run as they observe higher-level behaviour.
- **Regression tests**: testing previously encountered bugs to ensure that new changes do not reintroduce them.

From [45], conventions associated with the testing suite include:

- Avoid merging code unless all tests are passing.
- Always write tests for newly introduced logic.
- When fixing a bug, write a test to capture the bug and prevent future regressions.

### 3.1.1 Pre-train testing

Standard testing techniques and data checks can also be performed before the model is created. This is referred to as pre-train testing [45]. The main purpose of pre-train testing is to detect errors and faults that are susceptible to occur during, or as a result of, training, prior to initiating actual training. Identifying errors and faults early on prevents wasting an entire training job.

From [45], some tests that can be performed without trained parameters include:

- Verifying the validity of training and validation sets, e.g. checking for missing or faulty values.
- Verifying the shape of the model output.
- Checking that the output values are valid (e.g. that all outputs of a classification algorithm are between 0 and 1 and that the sum of all outputs do not exceed 1).
- Verifying decreasing loss after one batch of training.
- Training on different devices.

## 3.2 CheckList: Three Types of Behavioural Testing

This section reviews the paper "Beyond Accuracy: Behavioral testing of NLP models with CheckList" by Ribeiro et al. [46]. The paper is spoken highly of and referenced in several settings [44, 45, 47], including by Victor Sanh (Hugging Face [48]), Ambarish Jash (Google AI [49]), and Piero Molino (Stanford University [50], Ludwig [51], and former Uber AI [52]) in a webinar offered by Comet ML [53].

Behavioural testing involves treating the model as a black box and test input data against expected outputs. Ribeiro et al. [46] introduce a task agnostic methodology for testing NLP models based on the principles of behavioural testing, with a tool called CheckList. Ribeiro et al. [46] define three types of tests:

- Minimum functionality tests (MFT)
- Invariance tests (INV)
- Directional expectation tests (DIR)

CheckList is designed to test NLP models, but the concept behind it might be adopted and applied to other models. This section focuses on the concept rather than the tool itself. Ribeiro et al. [46] recommend structuring tests around the *capabilities* the model is expected to acquire from training. For an NLP model, these capabilities might include: *vocabulary and part of speech* - whether a model has the necessary vocabulary, and whether it can appropriately handle the impact of words with different parts of speech on the task; *sentiment* - ability to identify words that carry positive, negative, or neutral sentiment; *name entity recognition* (NER) - detecting names; *fairness* - outputs that should be independent of certain variables; *robustness* - to typos or irrelevant changes; *negation* - changes that negate the output; *temporal relationships* - understanding the order of events [46]. Capabilities in an image recognition model might include: *object rotation*; *perspective shift*; *lighting conditions*; *weather conditions* [45]. The examples are intentionally many since this is deemed to highlight the motivation behind these tests; the training data might include coincidental correlations which cause the model to learn unintended patterns.

#### Minimum functionality tests

Minimum functionality tests (MFT), inspired by unit tests in traditional software testing, are collections of simple labelled examples to check the behaviour within a capability. MFTs are particularly useful for detecting when a model uses "shortcuts" to handle complex inputs without actually mastering the relevant capability [46]. MFTs quantify model performance for specific cases, similarly to how unit tests isolate and test atomic components [45].

#### Invariance tests

Invariance tests (INV) involve applying label-preserving perturbations to inputs with the expectation that the model prediction remains unchanged. Different capabilities need different perturbation functions, e.g. altering an independent variable to test fairness, or introducing a typo to test robustness.

#### Directional expectation tests

Directional expectation tests (DIR) are similar to invariance tests, with the distinction that the prediction is expected to change in a certain way after applying a perturbation to the input.

Invariance tests and directional expectation tests are inspired by *metamorphic testing*, which is discussed in section 3.3. They allow testing on unlabelled data since they test behaviour with respect to relationships between predictions before and after perturbations are applied [46].

CheckList revealed critical bugs in commercial systems developed by large software companies, indicating that it complements current practices well [46]. CheckList is applied

| Capability | Min Func Test | INVariance | DIRectional |
|---|---|---|---|
| Vocabulary | Fail. rate=15.0% | 16.2% | C 34.6% |
| NER | 0.0% | B 20.8% | N/A |
| Negation | A 76.4% | N/A | N/A |
| ... | | | |

| Test case | Expected | Predicted | Pass? |
|---|---|---|---|
| A Testing **Negation** with ***MFT*** Labels: negative, positive, neutral | | | |
| Template: I {NEGATION} {POS_VERB} the {THING}. | | | |
| I can't say I recommend the food. | neg | pos | X |
| I didn't love the flight. | neg | neutral | X |
| ... | | | |
| Failure rate = 76.4% | | | |
| B Testing **NER** with ***INV*** Same pred. (inv) after removals / additions | | | |
| @AmericanAir thank you we got on a different flight to [ Chicago → Dallas ]. | inv | pos → neutral | X |
| @VirginAmerica I can't lose my luggage, moving to [ Brazil → Turkey ] soon, ugh. | inv | neutral → neg | X |
| ... | | | |
| Failure rate = 20.8% | | | |
| C Testing **Vocabulary** with ***DIR*** Sentiment monotonic decreasing (↓) | | | |
| @AmericanAir service wasn't great. You are lame. | ↓ | neg → neutral | X |
| @JetBlue why won't YOU help them?! Ugh. I dread you. | ↓ | neg → neutral | X |
| ... | | | |
| Failure rate = 34.6% | | | |

Figure 3.1: CheckListing a commercial sentiment analysis model developed by Google. Illustrated by Ribeiro et al. [46].

to a commercial sentiment analysis model developed by Google in fig. 3.1, which shows examples of test cases and how well the model performed on them.

## 3.3 Improving Dependability of Machine Learning Applications

The section reviews the paper "Improving dependability of Machine Learning Applications" by Murphy and Kaiser [54].

Murphy and Kaiser [54] presents a methodology for improving the dependability of machine learning systems, consisting of three approaches: *niche oracle-based testing*, *parameterized random testing*, and *metamorphic testing*. These approaches are not intended as alternatives but in combination. The paper places machine learning applications, being software systems with no reliable test oracle available, in the category of "non-testable programs" from [55]. It is naturally possible to pass a data point with a known label into a machine learning algorithm, and see whether it produces the correct result or not, but this is not a sufficient representation for the general case. In machine learning, as in any software testing, it is possible only to show the presence of bugs, but not their absence [54]. As they put it, the research in Murphy and Kaiser [54] "seeks to address the issue of how to devise test

cases that are likely to reveal bugs, and how one can indeed know whether a test actually is revealing a bug, given that we do not know what the output should be in the general case.".

### Niche oracle-based testing

The first approach uses a *"niche oracle"*, a test oracle that only applies to a very small subset of the input domain, for which the expected output, in fact, can be known in advance. In this approach, small predictable data sets are hand-crafted, such that a particular model will be obtained if the algorithm is implemented correctly. This particular model thus functions as a test oracle for a small data set, a niche oracle. These data sets are typically quite trivial, but the tests function as preliminaries before other testing can proceed. This approach bears similarities to MFTs discussed in section 3.2.

### Parameterized random testing

This approach is based on the notion of random testing [56, 57]. Random testing is a convenient way to produce large input data sets, in the absence of sufficient real-world data sets. This addresses the limitation of niche oracle-based testing, where the data sets are small. A challenge is that, due to the randomness, there is no test oracle specifying the expected output. In the *parameterized random testing*-approach, the randomness that is used to generate large data sets is parameterized based on the different equivalence classes that are intended for testing.[2] Although this approach does not result in any oracle, it helps reveal defects and inconsistencies in some cases [54].

### Metamorphic testing

Metamorphic testing, also discussed in [58, 59], is designed as a general technique where existing test cases, particularly those that have not revealed any failure, are used as bases for creating follow-up test cases intended to find uncovered flaws. The methodology revolves around reusing input test data to create additional test cases whose outputs can be predicted, allowing the application to act as a "pseudo-oracle" for itself, by specifying the behaviour that is expected upon changes to the input. Even though the output of the follow-up test case is as expected, it is based on the output of the initial test case, meaning that both outputs could be wrong. Still, metamorphic testing forms a powerful technique for revealing defects [54].

## 3.4 Testing Deep Neural Networks

The section reviews the paper "Testing Deep Neural Networks" by Sun et al. [60].

[2] The equivalence classes devised, and used to guide the generation of appropriate input data sets for the three approaches, in Murphy and Kaiser [54] are: small vs. large data sets; repeating vs. non-repeating attribute values; missing vs. non-missing attribute values; repeating vs. non-repeating labels; negative labels vs. non-negative-only labels; predictable vs. non-predictable data sets; and combinations thereof. These were devised after analysis of the MartiRank and SVM algorithms, the algorithms which, in the paper, are subjected to the three test approaches.

Sun et al. [60] propose a family of four novel test criteria that are tailored to the structural features of deep neural networks and their semantics. The criteria are inspired by the MC/DC (modified condition/decision coverage) criterion but are designed for the specific attributes of deep neural networks. MC/DC, developed by NASA [61], is a method of measuring the extent to which safety-critical software has been adequately tested. At its core is the idea that if a decision can be made, all possible factors that contribute to that decision must be tested [60].

The paper is quite extensive, and the reader is referred to [60] for deeper immersion in the criteria and their derivations than what is provided in this section.

Sun et al. [60] state that the core idea of their criteria is "to ensure that not only the presence of a feature needs to be tested but also the effects of less complex features on a more complex feature must be tested.". This formulation is influenced by the claim that nodes in a deeper layer in a deep neural network represent more complex attributes of the input [62]. Further, Sun et al. [60] consider every *feature*, i.e. a subset of nodes in a layer, as a *decision*, and say that its *conditions* are those features connected to it in the preceding layer, to integrate the context of MC/DC.

### Feature changes

The criteria are defined by capturing different ways of instantiating the changes of the conditions and the decision. Sun et al. [60] consider the observed on a feature to be either a sign change or a value change. A sign change occurs if the sign of all nodes in a feature differs on two separate test cases; e.g.

$$sign(n_{k,j}, x_1) \neq sign(n_{k,j}, x_2) \forall n_{k,j} \in \psi_{k,l}$$

, where $n_{k,j}$ is a node $j$ in layer $k$ in $\psi_{k,l}$. And $\psi_{k,l}$ is feature $l$ in layer $k$, and $x_1$ and $x_2$ are test cases. A sign change is denoted by $sc(\psi_{k,l}, x_1, x_2)$. A non-sign change occurs if the sign of no nodes in a feature differs on two separate test cases. A non-sign change is denoted by $nsc(\psi_{k,l}, x_1, x_2)$. Value change is dependent on the *value function*[60], $g \rightarrow \{true, false\}$, which expresses the deep neural network developer's intuition or knowledge about what constitutes a significant change on a feature $\psi_{k,l}$, by specifying the difference between two vectors $\psi_{k,l}[x_1]$ and $\psi_{k,l}[x_2]$. There are no formal restrictions on the form of the value function other than that it needs to be evaluated efficiently, for practical reasons. Thus value change signifies that a test case $x_1$ constitutes a significant change on feature $\psi_{k,l}$, compared to test case $x_2$:

$$g(\psi_{k,l}, x_1, x_2) = true$$

, and is denoted by $vc(g, \psi_{k,l}, x_1, x_2)$.

### The four test criteria

A test criterion is defined on an instance of a deep neural network, with a set of test cases, a set of feature pairs, and a coverage method [60]. The essence of the four test criteria are

the four coverage methods, which Sun et al. [60] define as follows:

**Sign-Sign Coverage, or SS Coverage**
A feature pair $\alpha = (\psi_{k,i}, \psi_{k+1,j})$ is SS-covered by two test cases $x_1$, $x_2$, denoted by $SS(\alpha, x_1, x_2)$, if the following conditions are satisfied by the deep neural network instances $\mathcal{N}(x_1)$ and $\mathcal{N}(x_2)$:

- $sc(\psi_{k,i}, x_1, x_2)$ and $nsc(P_k \setminus \psi_{k,i}, x_1, x_2)$
- $sc(\psi_{k+1,j}, x_1, x_2)$

where $P_k$ is the set of nodes in layer $k$.

In worded form from [60]: SS coverage provides evidence that the sign change of a condition feature $\psi_{k,i}$ independently affects the sign of the decision feature $\psi_{k+1,j}$, of the next layer .

**Value-Sign Coverage, or VS Coverage**
Given a value function $g$, a feature pair $\alpha = (\psi_{k,i}, \psi_{k+1,j})$ is VS-covered by two test cases $x_1$, $x_2$, denoted by $VS^g(\alpha, x1, x2)$, if the following conditions are satisfied by the deep neural network instances $\mathcal{N}(x_1)$ and $\mathcal{N}(x_2)$:

- $vc(g, \psi_{k,i}, x_1, x_2)$ and $nsc(P_k, x_1, x_2)$
- $sc(\psi_{k+1,j}, x_1, x_2)$

In worded form from[60]: Intuitively, the first condition describes the value change of nodes in layer k and the second requests the sign change of the feature $\psi_{k+1,j}$. In addition to $vc(g, \psi_{k,i}, x1, x1)$, $nsc(P_k, x_1, x_2)$, which denotes no sign changes for any node at layer $k$ is needed. This is to ensure that the overall change to the activations in layer $k$ is relatively small.

**Sign-Value Coverage, or SV Coverage**
Given a value function $g$, a feature pair $\alpha = (\psi_{k,i}, \psi_{k+1,j})$ is SV-covered by two test cases $x_1$, $x_2$, denoted by $SV^g(\alpha, x1, x2)$, if the following conditions are satisfied by the deep neural network instances $\mathcal{N}(x_1)$ and $\mathcal{N}(x_2)$:

- $sc(\psi_{k,i}, x_1, x_2)$ and $nsc(P_k \setminus \psi_{k,i}, x_1, x_2)$
- $vc(g, \psi_{k+1,j}, x1, x1)$ and $nsc(\psi_{k,i}, x1, x1)$

In worded form from [60]: The first condition is identical to the first condition for SS Coverage. The second condition considers the feature value change $vc(g, \psi_{k+1,j}, x1, x1)$ with respect to a value function $g$ by independently modifying one of its condition features' sign. Intuitively, SV Coverage captures the significant change of a decision feature's value that complements the sign change case.

**Value-Value Coverage, or VV Coverage**
Given two value functions $g_1$ and $g_2$, a feature pair $\alpha = (\psi_{k,i}, \psi_{k+1,j})$ is VV-covered by two test cases $x_1$, $x_2$, denoted by $VV^{g_1,g_2}(\alpha, x1, x2)$, if the following conditions are satisfied by the deep neural network instances $\mathcal{N}(x_1)$ and $\mathcal{N}(x_2)$:

- $vc(g_1, \psi_{k,i}, x_1, x_2)$ and $nsc(P_k, x_1, x_2)$

- $vc(g_2, \psi_{k+1,j}, x_1, x_2)$ and $nsc(\psi_{k+1,j}, x_1, x_2)$

In worded form from [60]: Intuitively, VV coverage targets scenarios in which there is no sign change for a condition feature, but the decision feature's value is changed significantly.

The test conditions required by these criteria exhibit particular conditions between the condition feature and the decision feature, and generating test cases for them are not trivial [60]. As discussed in [63], random test case generation is inefficient for coverage testing of deep neural networks, since every new input generates new coverage. Sun et al. [60] consider the symbolic encoding in the concolic testing[3] method in [64] expressive enough to encode test conditions required by their criteria. Sun et al. [60] also present a new test case generation algorithm based on gradient descent search, which scales better to large deep neural networks.

Other proposed structural test coverage criteria for deep neural networks include neuron coverage [65] and some extensions of it [66]: neuron boundary coverage, multisection neuron coverage, and top neuron coverage, and safety coverage [67].

## 3.5 Developing Bug-free Machine Learning Systems with Formal Mathematics

The section reviews the paper "Developing bug-free machine learning systems with formal mathematics" by Selsam et al. [43].

Due to the many potential causes for undesired behaviour in a machine learning system, implementation errors can be extremely difficult to detect. Selsam et al. [43] demonstrate a methodology in which developers use an *interactive proof assistant* [68, 69, 70, 71, 72, 73, 74] (via [43]) to both implement their system and to state a formal theorem that defines what it means for their system to be correct. This methodology enables developers to find and eliminate implementation errors systematically without recourse to empirical testing. Its structure is abstractly illustrated in fig. 3.2.

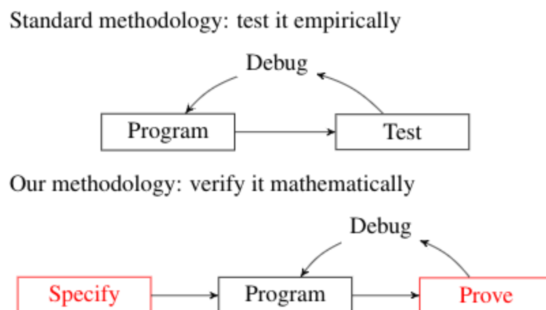



Figure 3.2: High-level comparison between standard methodology for testing machine learning systems and the methodology presented by Selsam et al. [43], as illustrated by Selsam et al. [43].

The interactive proof assistant consists of: a programming language; a language to state mathematical theorems; and a set of tools for constructing formal proofs[4] of such theorems.

[3]Concolic testing combines program execution and symbolic analysis to explore the execution paths of a software program [64].

[4]In Selsam et al. [43] the term *formal proof* means a proof that is in a formal system, and so can be checked by a machine.

In the approach, developers first use the theorem language to state a formal mathematical theorem, defining what it means for their implementation to be free of errors in terms of the underlying mathematics. Implementing the system with the programming language ensues, and developers use the set of tools to construct a formal proof of the previously stated theorem, which states that their implementation is correct. The interactive proof assistant will then expose any implementation errors systematically, by yielding impossible proof obligations.

The ultimate goal is to abstract the mathematics and logic to be checked by a machine so that cognitive demand during development is reduced.

Selsam et al. [43] report that their initial application of the methodology imposed many new requirements that increased the overall workload, but the development process as a whole was experienced as less cognitively demanding. They also suggest that their methodology can be adopted incrementally, by having the specifications not cover functional correctness, or not all theorems proved. This is subject to the assurance level of the applications. Selsam et al. [43] thus expect that pragmatic use of their methodology could be useful for developing a wide range of machine learning systems to varying standards of correctness.

Chapter 4

# Employing MLOps

A good machine model yields little reward if it is not available for practical use or applicable to real-world data. To fully utilize a machine learning system, it must be operationalized. Operationalizing is a demanding task that poses several challenges. As such, teams struggle in how to approach operationalization of machine learning systems. This chapter investigate methods for organizing this process, both in terms of useful software tools, and on a more organizational level, exploring the discipline of MLOps.

## 4.1 Machine Learning Pipeline

The section introduces the machine learning pipeline, a valuable technique for structuring and defining modules in a machine learning application. Also presented are other useful pipelines.

*Note: pipeline/pipelining in this context does not refer to the technique for implementing instruction-level parallelism within a CPU, also known as "instruction pipelining".*

The overarching purpose of a pipeline is to streamline all processes needed for an application to fulfil its purpose.

Pipelines exist in various forms. A data pipeline is a series of transformations that are applied to data between its source and a destination and is one of the core concepts in data engineering [21]. A deployment pipeline makes up the automatic process of taking code from version control to production deployment, through acceptance tests and development environments [75], and is instrumental to DevOps. These successful applications have inspired the machine learning pipeline. This section elaborates on why a pipeline is useful and explains how it can be used in machine learning.

One use case for a pipeline is to obtain an automated workflow [76], so that little human interaction is required for an iteration of the application it is applied to. This is a highly desired property in a deployment pipeline, e.g. a *CI/CD pipeline* (abbreviated from Continuous Integration & Continuous Delivery/Deployment). **Continuous integration** signifies the practice of developers merging their changes into the main branch (of version control) frequently, and is sustained by automatically building a new build that is run on automated tests [77]. Upon integrating more frequently, it is easier to detect bugs, and potential bug fixes requires less work. It also saves developers from spending a lot of time

implementing large changes that turn out not to be feasible. The backbone of this practice is the automation of building and testing, which, if done manually, requires so much time that continuous integration is not feasible. **Continuous Delivery/Deployment** is the extension of continuous integration, where integrated code is automatically deployed into a testing and/or production environment [77]. This allows for reliable and quick delivery of bug fixes, features and configuration changes into production [75]. "Continuous delivery" and "continuous deployment" are used interchangeably, but [77] defines the difference as "continuous delivery" requiring human intervention, typically in the form of a button click, before actually deploying, while "continuous deployment" implies that new releases are deployed automatically, without human intervention, and are only prevented in the case of failed tests.

Another effect of pipelines is modularity in a workflow. A machine learning pipeline can be created by splitting machine learning workflows into independent, reusable modular parts that can be pipelined together to create models [76]. Smaller parts with well-defined purposes improve readability and run time visibility, make maintenance and debugging easier, and provide a better foundation in terms of scalability. Modularity also allows for a higher degree of independence for each module, e.g. concerning the programming language or framework to be used in its development.

Algorithmia [76] emphasizes three problems that arise when scaling a monolithic (non-modular) architecture:

- **Volume**: when deploying multiple versions of the same model, the whole workflow is run every time, even though the first steps are identical. This results in unnecessary use of computation power and time.
- **Variety**: upon expanding the model portfolio code from the beginning of the workflow must be copied and pasted, which is inefficient.
- **Versioning**: some parts of the workflow are more prone to frequent changes, and when these changes are due, all script must be updated manually, which is time-consuming and creates room for errors.

Algorithmia [76] then suggests creating a pipeline from a more modular architecture to address these problems:

- **Volume**: only call the separate parts of the workflow when needed and cache or store results for later use, if necessary.
- **Variety**: parts from the beginning of the workflow can simply be pipelined into new models without being replicated.
- **Versioning**: only one copy of each part to update. The remaining modules all depend on the one copy where changes are implemented.

The steps in creating a machine learning model include, among other things, data preprocessing and training, which are essentially data transformations. These transformations are often achieved through the use of scripts or cells in a notebook and are thus hard to manage and run reliably [21]. One can thus integrate machine learning-related transformations into a data pipeline to create a machine learning pipeline. For a machine learning system, one would typically create two such pipelines; one for training and one for serving/making

predictions. This is due to the difference in how each process formats and accesses data. This difference is particularly prominent in models that serve real-time requests/streamed data, as opposed to batches [21].

The machine learning pipeline is created as a pure code artefact, which is independent of specific data instances, meaning it can be versioned with source control, and its deployment can be automated with a regular CI/CD pipeline [21]. This is illustrated in fig. 4.1.

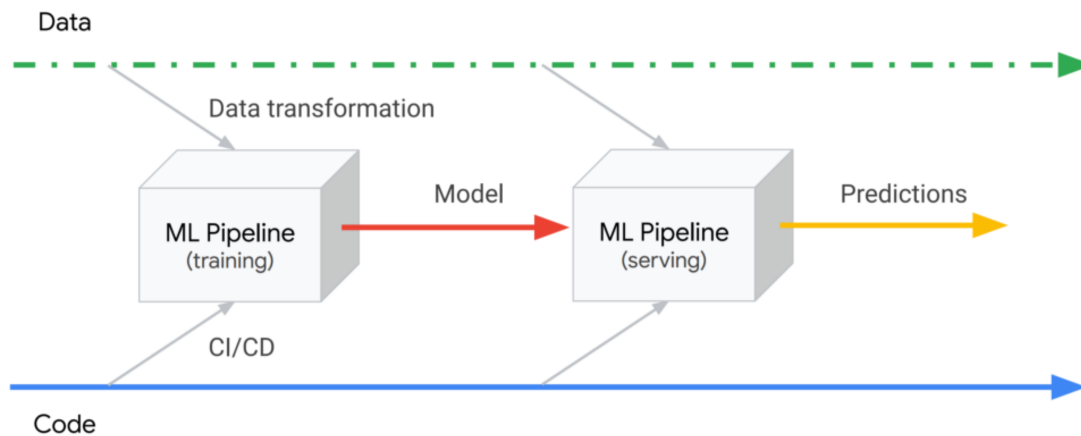



Figure 4.1: ML pipeline in CI/CD pipeline as illustrated by Breuel [21]

## 4.2 Machine Learning Operations (MLOps)

*"Would you spend many years and big money training athletes and then send them to the Olympic Games, only to make them stay in their hotel instead of competing?... Models that do nothing more than provide static insights in a slideshow are not truly "operational", and they don't drive real business change."* - Sweenor et al. [78]

The machine learning era has made way for a new discipline named *MLOps*. This section explains its emergence, describes what it is, and discusses its importance and status as a research topic.

#### What is MLOps, and why is it needed?

Defined by Sweenor et al. [78]; "ML Ops is a cross-functional, collaborative, continuous process that focuses on operationalizing data science by managing statistical, data science, and machine learning models as reusable, highly available software artefacts, via a repeatable deployment process."

Simply put; the purpose of MLOps is to get machine learning systems into production so that they can fully serve their purpose, instead of being reduced to a statistical tool that hardly ever sees the light of day. The term "MLOps" is derived from "machine learning" and "operations" in the same way that "DevOps" originates from "development" and "operations", and can be simplified as "DevOps for ML". In reality, it is a mix of many disciplines; machine learning, data science, data engineering, software development, DevOps, etc. There exists no official consensus on the disciplines involved, and thus "MLOps" is deemed a practical and concise term. Its diverse composition might be a reason why

MLOps has not been given the attention it deserves; no discipline considers MLOps to be a part of it.

MLOps is a continuous process in the way that an important part of it is to continuously monitor the system in production, and in the event of a machine learning model needing to be updated, other parts are set in motion as well. However, “process” is a slightly inaccurate term, so perhaps it is more correct to say that MLOps is a set of practices and that continuous monitoring is one of these practices.

Although there are preliminary preparation steps, the step when a system actually enters an operationalized state is deployment. Limited ability to deploy machine learning systems is a widespread problem [79, 80, 81]. In Algorithmia’s report from 2019 [80, 81] the majority of companies report on spending between 8 and 90 days to deploy a single model, and 18% even longer than that. According to [17] (2019), only 22% of companies using machine learning have successfully deployed a model.

Due to the current infancy of MLOps, it cannot simply be employed and, as a result, solve these problems. Hopkins [20] writes about MLOps in collaboration with HPE Ezmeral [19], an MLOps platform. According to Hopkins [20] Matt Maccaux[1] states that between 80-85% of companies employing MLOps are unable to put models into practice and that some 60% of all machine learning models across the enterprise have been built but not operationalized, due to a lack of implementation tools. Further from [20], Matheen Raza[2] claims that at its most efficient and effective “MLOps must enable enterprises to standardize the machine learning lifecycle while providing users with the flexibility to deploy their machine learning applications across their choice of infrastructure - either on-premises, in multiple clouds, or at the edge - while also maintaining enterprise-grade security and governance.”.

### Why are machine learning systems so difficult as compared to traditional software?

DevOps is extremely useful in traditional software development, enabling frequent releases of traditional software by abstracting away and automating much of the complexity involved in deploying new versions. This begs the question of why companies struggle so much with operationalizing their machine learning systems.

One of the root causes is that, as opposed to traditional software, where changes are only performed in the code before a new release, machine learning systems experience alterations along three axes: the code, the model and the data [22]. Code is predictable, being crafted in a controlled development environment. Data comes from the continuously changing “unending source of entropy that is the real world” [21], which renders it unpredictable. The model changes in a controllable manner, on the volition of the developers. However, its contents depend on the unpredictable data, yielding unpredictable characteristics. This unpredictability presents difficulties in automation and abstraction, and testing as discussed in chapter 3.

In terms of data, challenges occur - not solely because of its unpredictable nature - but also due to its sheer volume. Code can be stored and versioned with classic version control tools, such as Git, but this is highly impractical with large amounts of data that change frequently. The essence of a machine learning algorithm is that it builds itself, using data. Thus, versioning a machine learning model by tracking the changes in its contents; weights

[1]Matt Maccaux is the Global Field CTO for HPE Ezmeral.

[2]Matheen Raza is Senior Manager, GTM Strategy for HPE Ezmeral.

and other parameters, provides little value. Developers and data scientists have no way of making sense of that information. To version a model, it needs to be associated with metadata; the version of the code at the time of its creation, the hyperparameters used for training, and, perhaps most importantly, the data that was used for training [21].

### Incorporating MLOps

To successfully incorporate MLOps in your organization, it is paramount that it is given sufficient attention. One thing that more or less all sources on MLOps agree on is the importance of a close-knitted, cross-functional team [78, 21, 82, 83, 22]. Using a software project as a metaphor for the team, with each team member/faction making up each of the modules in the software project: it is of the essence that each of the modules receives the input it expects and that they produce the output expected from other modules. Naturally, the concept of cross-functional teams is not particularly revolutionizing; being considered important in practically any organization. It is considered particularly important in this context, however, because to be able to produce the output expected, one must first be familiar with the other modules. This may involve attaining new skills within other disciplines. If this groundwork fails to be performed properly, a team risks wasting time, having to redo work that was based on presumptions about the requirements.

In fact, the task of knitting all the involved disciplines together is so comprehensive, and the presence of an actor with a certain amount of knowledge within several disciplines, who can function as a *"gap-bridger"*, is so valuable, that Sweenor et al. [78] propose the role of *MLOps engineer*. The MLOps engineer is essentially in charge of managing the model life cycle, including provenance, version control, approval, testing, deployment, and replacement. The models must also be managed in a central repository to enable approvals, electronic signatures, version control, tracking and reuse. Thus, an MLOps engineer is ideally someone with enough knowledge about machine learning models to understand how to deploy them, and enough knowledge about operational systems to understand how to integrate, scale, and monitor models. The MLOps engineer is depicted among other relevant roles in fig. 4.2.

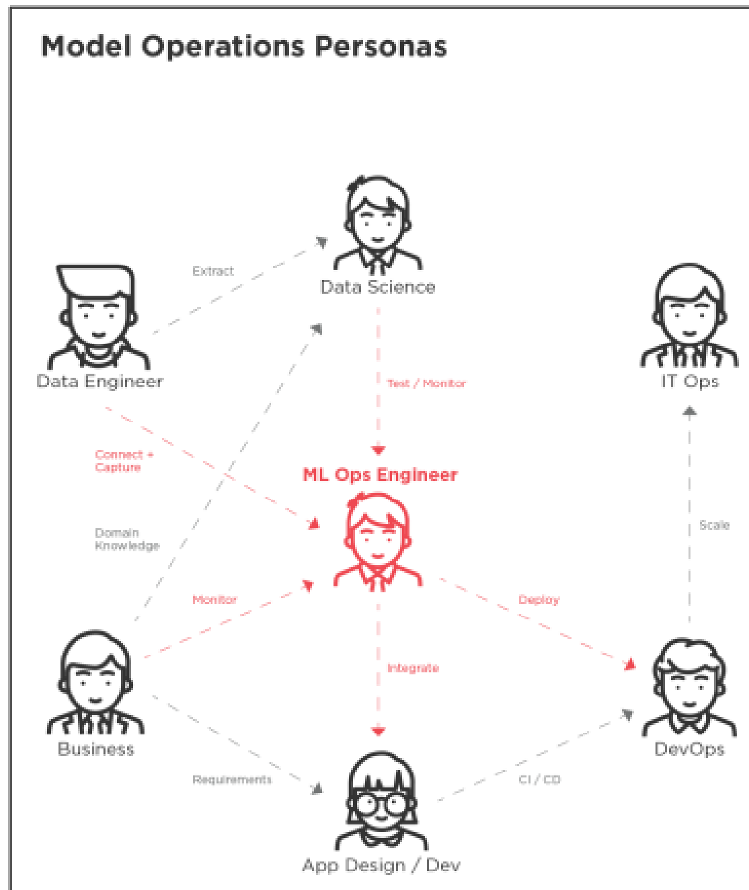



Figure 4.2: The role of the MLOps engineer throughout the entire pipeline, as illustrated by Sweenor et al. [78]

Sweenor et al. [78] elaborate on the collaborative team in fig. 4.2:
The *data engineer* ensures that all data sources are available and provides curated data sets to the other analysts. The *data scientist* explores the data, and trains and builds the predictive machine learning model. After this, they collaborate with the *MLOps engineer* to test the trained model on production data in a development/*"sandbox"*[3] environment. If the model produces satisfactory test results, the MLOps engineer deploys it into final production environments. Data scientists and MLOps engineers both communicate with the *business user* to understand what the model should predict and what regulatory constraints are applicable. The *application designer* collaborates with the data scientist, deciding which models to use and how best to integrate them, and the *application developer* writes the actual code for integrating the model into the application. The *DevOps engineer* leads the system from development to *IT Operations*, which maintains the infrastructure and ensures that the application and its models run, perform, and scale optimally.

The importance of a well-defined team and, situationally, a specified MLOps engineer is further manifested by [86], which reports that 40% of respondents say that they work with both models and infrastructure, in a survey with responses from 331 professionals in the machine learning domain from 63 different countries.

[3] A sandbox is a testing environment that isolates untested code changes and outright experimentation from the production environment or repository. [84] via [85].

### 4.2.1 A four-step model of MLOps

There is little consensus as to exactly what MLOps includes. The scope and purpose of MLOps, and thus how it is divided into separate parts, is influenced by the role of the author of the literature addressing it. Sweenor et al. [78] define four main steps of MLOps: **Build, Manage, Deploy and Integrate,** and **Monitor**. Agrawal and Mittal [87] prefer to divide MLOps into five steps: **Business Understanding, Data Acquisition, Model Development, Model Deployment**, and **Model Monitoring**.

The scope of this thesis best coincides with the four-step model. The five-step division emphasizes the process of deciding whether or not machine learning should be applied to a problem, considering use case, data availability, etc. While this is important, this thesis focuses on the case where a machine learning model is ready, but the operationalization remains. The four-step model does not make up a complete guide of MLOps, but it does offer insight into the diversity of its contents and addresses several important aspects. Following is a systematic review of the four-step model presented by Sweenor et al. [78]. Contextual elaborations and reflections by the author of this thesis and other sources are added occasionally, for completeness and for emphasizing relevance.

#### Build

The building step described by Sweenor et al. [78] revolves around machine learning in its data scientific term. Data scientists orient themselves in the data landscape to make sure that training data provides characteristics that are representable for data that will be encountered in the real world, once the model is deployed. They must then set up machine learning and data pipelines that perform necessary operations on the data and the model. Feature engineering is performed to optimize the potential of the machine learning models. After this, the models are trained and tested until satisfactory behaviour is achieved.

Feature engineering involves analyzing variables in data and determining what transformations to apply to these variables with the purpose of increasing the predictive power of a machine learning model. It is also possible to create new variables to achieve this purpose. It is critical that these engineered features can be reproduced in the operational system [78]. If not, the model will be unable to perform predictions due to lack of necessary features.

Testing the model includes examining the model's performance and accuracy through executing it on data that is representative of the production environment. Often, in practice, there are several environments; development environment, testing environment, as well as other pre-production environments, including environments where new models are run in parallel with the current one, to best evaluate any potentially increased performance. Such agility can often be provided by modern cloud-based technologies, without adding much overhead in cost and effort [78].

#### Manage

What Sweenor et al. [78] describes as the managing step includes the MLOps engineer, described previously in this section, and how that role integrates with other roles in a collaborative, cross-functional team. Also included in the managing step is attempting to control model proliferation. Multiple people working along the machine learning pipelines

causes frequent creation of models for testing purposes. Over time it becomes clear that several of these models address the same problems, but this is not sufficiently communicated between team members. Automation techniques, e.g. AutoML [88, 89], make it easy to generate new models. This is a good thing in terms of streamlining testing and development, but it also poses problems in how it results in a greater need to track, govern, and manage models across the organization [78].

In terms of managing model proliferation, Sweenor et al. [78] claims that the best way to centralize, and thus avoid multiple model generation sources, is to reuse and repurpose models from a centrally managed repository, which tracks who is building which model and keeps track of existing versions. Multiple people can then use a single, approved model for multiple applications. Such an approach provides insight into the progress of the data science team and the number of models it is building. It tracks connections between data, models, and projects, and is searchable [78].

Tracking and managing models lead to important insights and overviews. It enables quick transitions between models when deemed necessary. It also provides the means necessary for analysis; determining which alterations to the model resulted in certain effects. Consistent version tracking is essential for reproducibility [21], and for compliance - compliance is important in many organizations, and thus being able to identify which version of a model performed a specific prediction can be vital [78].

Due to the dynamic and data-dependent nature of machine learning models, they are more difficult to track than traditional software. Version control for models should track the data and parameters that were used to build each version, as well as the model outputs; accuracy, coefficients, etc. This to achieve a clear provenance and a way to track how the model was produced [78].

Both models and metadata can be tracked using standard version control, such as Git. It is important not to tie the model life cycle to the code life cycle, as training and code releases should happen on different schedules [21]. Data is usually too large and mutable for standard version control to be a practical solution. The ideal solution would be a purpose-built tool, but as of now, there is no clear consensus in the market [21].

#### Deploy and integrate

*"In the Deploy and Integrate step, the model stops looking in the rearview mirror at historical data and looks through the windshield at live, real-world data"* - Sweenor et al. [78].

Deployment and integration are closely intertwined. Sweenor et al. [78] define "deployment" as the process of taking a model from the environment in which it is developed and turning it into an executable form. "Integration" is followingly defined as the process of taking that deployed model and embedding it within an external system. Deployment options usually depend on integration endpoints and vice versa. Thus, Sweenor et al. [78] conclude that the best way to approach deployment is as a function of how the model will be integrated, and the best way to regard integration is as an extension of deployment.

The goal of deploying and integrating is to make an actual impact on the business application. This impact naturally depends on the application in question. E.g., in the case of embedding product recommendations from a machine learning model into a mobile application, the model does not represent the main purpose of the application and must be integrated in a way that aligns well with the application development. Contrastingly, in

the case of using time series data from an industrial process to train a machine learning algorithm to model the process, the machine learning model is the very essence of the application, and integration can be done in a way that better accommodates the model.

Nonetheless, a machine learning model must be integrated into an application that facilitates it. This can largely alter the software development life cycle [78], and thus MLOps comes into play also on parts that are not directly related to the machine learning model. The degree to which the software development life cycle is changed naturally depends on the prominence the machine learning model exhibits in the application.

Sweenor et al. [78] recommend entirely decoupling the machine learning models from the application, by providing them as a set of APIs that developers can test and incorporate into their code. This requires thorough documentation from the data scientists, describing how the models are to be used, what are their required inputs and expected outputs, their limitations and domains of applicability, levels of accuracy and confidence, and data dependencies [78]. This process is subject to close collaboration between data scientists and developers upon iterations of model development and refinement - it cannot simply be "thrown over the wall" [78].

Due to the unpredictable nature of data, a predictive model may quickly become highly inaccurate and needs to be swapped or disabled. The model must be integrated into the application in a way that allows for this to happen without requiring a new version of the application [78].

To detect if a model is deteriorating in performance, it needs to be continuously monitored.

**Monitor**

Monitoring starts once a model has been deployed. The monitor step described by Sweenor et al. [78] covers three types of metrics: statistical, performance, and business/ROI (Return of Investment). Also included in the monitor step are the tasks of automatic retraining and remodelling, thus monitoring extends to continuously reviewing the model, and retraining or replacing it when deemed necessary.

Sweenor et al. [78] divide statistical metric tracking into three practices:

- *Accuracy tracking* includes misclassification rate, confidence rate, or other error rates decided by the data scientists. When these rates fall below a given threshold the model should be retrained or remodelled.

- *Champion-challenger* is a practice of continually comparing the model in production (*champion*) with a potentially better model (*challenger*). A challenger model – either an updated version of the champion model, or a different model altogether – is run periodically, and its results are compared to those of the champion model. The most accurate model is chosen to run in production.

- *Population stability* involves constantly checking the distribution of data sets over time for reasonable consistency, to ensure that the current models are still relevant.

An obvious challenge with accuracy tracking is that there are no verified labels to which

the predictions can be compared. Else the predictions would have no purpose. In some cases, there might be an indirect way of assessing the quality of the predictions, such as measuring the amounts of clicks of an advert recommendation. In other cases, one has to rely on statistical measurements, by comparing predictions made in separate time periods, and check if they deviate more than any acceptable reason indicates that they should [21].

Performance metrics are less subjected to interpretation. Their purpose is to evaluate how well the infrastructure supports the models. Performance metrics include input/output, execution time, number of records scored per second, as well as hardware metrics, such as memory and CPU usage [78].

Business metrics and ROI evaluate the impact of the model in a business sense and thus serve as an important check against statistical metrics [78]. The statistical metrics determine how well a model performs in terms of the specified desired behaviour. The business metrics determine how much this desired behaviour actually impacts the business that the model is intended to improve. It is possible that the model is performing optimally, but that its purpose has no significant impact on the business. When the business metrics and statistical metrics do align, it indicates that the model is beneficial for the business, and yields return on investment of operationalizing machine learning [78].

It is important to monitor metrics across slices, to detect problems affecting specific segments. Different segments could be erroneous, but tested globally these errors might null each other out, and pass by undetected [21].

*Model drift*, also known as *concept drift*, is a phenomenon that occurs when real-world variables change over time, resulting in deteriorating accuracy of the model predictions. Such changes may occur due to real-world variables changing in their distribution, or their relationship to each other, or new variables may be introduced, that were not present at the time of creation of the current model [78]. When this happens, the model is no longer a good representation of the real-world process. To counteract model drift, models require constant monitoring and updating [78].

Updating the model involves either retraining the model; keep the current variables and model structure, but use newer data - or complete remodelling; add, remove, or alter variables, and restructure the model. In a scenario where real-world variables have changed in their relationship to each other, the model is likely to be improved through retraining. The structure is the same, but how the variables combine to influence the output is altered. Retraining requires the least effort and is in most cases sufficient to counteract model drift [78]. Remodelling might be required in a scenario where one or several real-world variables that affect the process have been newly introduced or neglected during initial creation. Figure 4.3 illustrates model decay over time, and how retraining improves performance.

Sweenor et al. [78] present a series of steps that ensue when a model is retrained and promoted to production, and include proposed ways of conducting them:

- *Model accuracy assessment* is based on a set of Gini coefficients[4]. If outside allowable limits, models are automatically reevaluated using a genetic algorithm for searching the parameter space. Candidate models are regularized with Elastic Net [5], decreasing variance at the cost of introducing some bias.

[4]Gini coefficient measures the inequality among values of a variable. It is a metric commonly used in econometric studies. [90].

[5]Regularization technique.

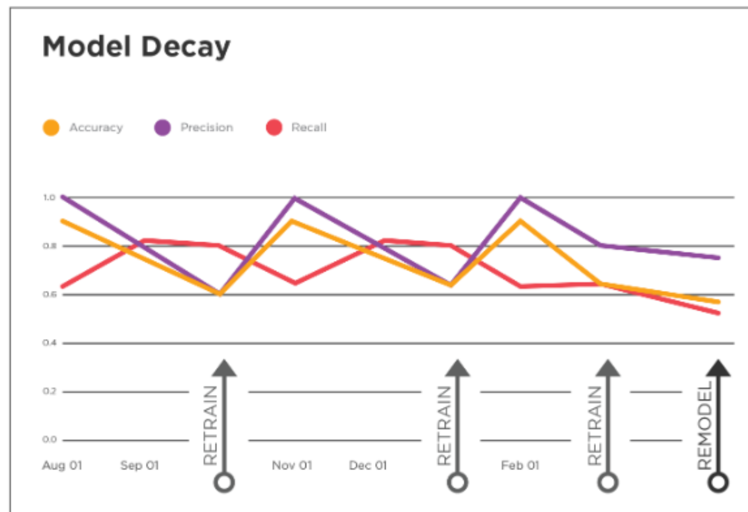



Figure 4.3: Model decay and the effect of retraining as illustrated by Sweenor et al. [78].

- *Model updates and explainability* includes assessment of business goals, variable importance, overfit, bias, outliers, and regulatory perspectives. The current and prior models are evaluated in a champion-challenger setting.
- *Model diagnostics* involves assessing local predictive power and accuracy across the predictor space. Areas of concern can be identified by assessing particular regions of the predictor space per recent trends. Statistical hypothesis test results and model diagnostic metrics provide more global insight. Model diagnostic metrics may include Akaike information criterion (AIC), Bayesian information criterion (BIC), the area under the ROC curve (AUROC), and visualizations such as ROC curves and lift charts. These metrics are discussed in chapter 5.
- *Model versioning, approval and audit* relates to keeping track of new model versions. In most cases of retraining, models are retrained on new data to account for variance in market conditions that have occurred since previous model training. Thus, several versions of the models must be saved and governed and should be available for auditing and compliance assessments.

Whether retraining or remodelling is the appropriate action and what the respective triggers should be are decisions to be made by the team behind the model.

### Automating monitoring

Several steps of the retraining phase are time-consuming and inefficient when performed through human interaction. It is desirable to automate model drift detection and retraining initiation, both for efficiency and accuracy. Several metrics can be subjected to automatic computation and evaluation, and frequent, automated, champion-challenger procedures are likely to provide better models in production. A human can still be in charge of the ultimate approval of redeployment, as a safety measurement, but automating several tasks can significantly reduce the workload, allowing for more frequent iterations, resulting in all over better performance.

Sweenor et al. [78] conclude that the sequence of build, manage, deploy and integrate, and monitor introduces structure and logical flow around analytics pipelines and allows the organization to realize the value of data science through MLOps. The sequence ensures that the best model gets embedded into a business system and that the deployed models are consistent with business requirements. They also claim that companies that build this kind of methodical thinking into their data science have a big competitive advantage over those that consistently fail to operationalize models by failing to prioritize action over mere insight.

# Chapter 5

# Components in the Machine Learning Life Cycle

This section presents an architecture consisting of typical components in a machine learning life cycle. The goal is to paint a clear picture of the entire operationalization process with descriptions that are interpretable by both software engineers and data scientists. The architecture provides clarity around which modules interact with each other, which domain each module mainly operates in, and how the modules can be separated. Such an architecture is hoped to function as a base for planning the structure and operationalizing of a machine learning system, with concrete modules to associate with needs, challenges, and solutions.

The architecture is illustrated in fig. 5.1, with component descriptions in tables 5.1, 5.2 and 5.3.

Upon considering what techniques, tools or technologies to use, fig. 5.1 and tables 5.1, 5.2 and 5.3 can be convenient, low-effort resources for evaluating how well a potential asset aligns with the machine learning system in question.

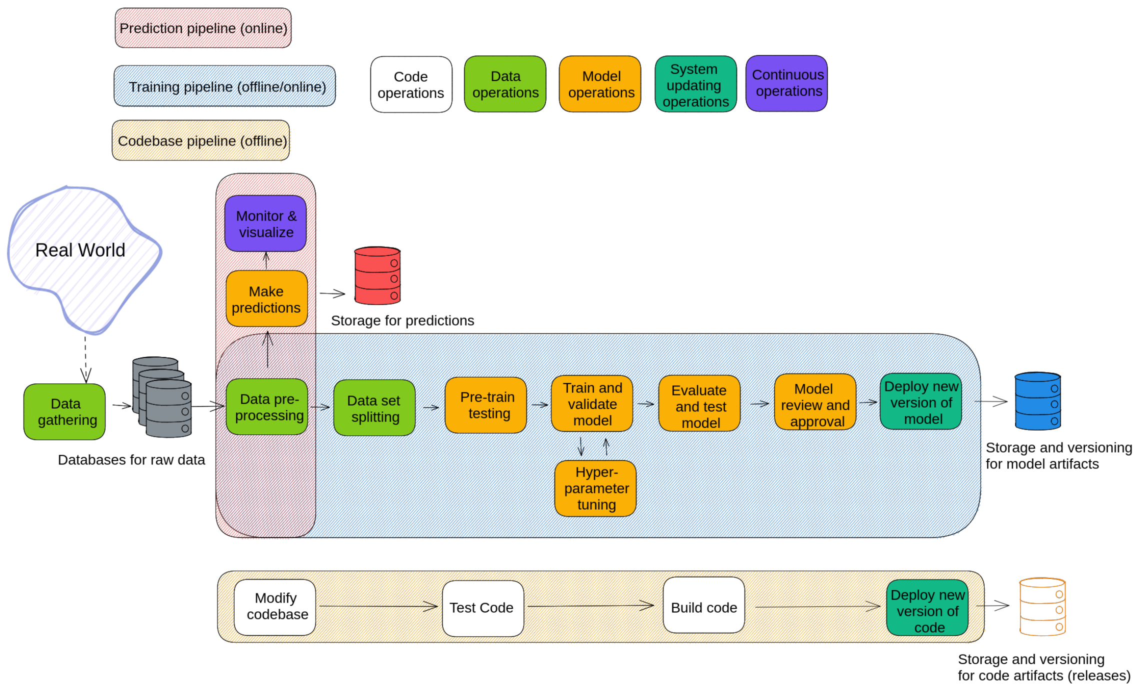



Figure 5.1: Architecture of a machine learning life cycle. Illustrated by the author.

Figure 5.1 consists of three pipelines and aims to illustrate how the components integrate. The codebase pipeline is only included for completeness, and not addressed further. The prediction pipeline represents the functionality of the system when online and in production. The training pipeline is marked as online/offline. In a highly automated well-designed system this pipeline can be online and operating with little human interaction. The is however a demanding feat, and many systems will employ an offline training pipeline - perhaps with some parts being online and automated. Coloured boxes represent a rough grouping of what type of operations the different components perform. For the sake of simplicity, some details and connections are left out; these connections along with a selection of components considered particularly interesting are further discussed later in this section. Tables 5.1, 5.2 and 5.3 provide brief descriptions of each component and their roles in the life cycle.

| Component Descriptions | | | | |
|---|---|---|---|---|
| **Component** | Inputs | Outputs | Purpose | Why is it important? |
| **Data gathering** | Measurements from the real world. | Sensor data organized in data storage facilities (e.g. databases). | Gather data for ML model to train and test on. | More diverse, and more data and results in better model performance. |
| **Data pre-processing** | Raw data from several sources. | Data from different sources, cleaned and verified, with the possible addition of engineered data, aggregated into a single data set. | Making data usable for the ML model. | Eliminate errors that might cause the ML model to misbehave. |
| **Data set splitting** | A single data set. | Three data sets: a training set, a validation set, and a test set. | Separate data into sets to be used for training, validating, and testing. | Clearly organizes the separate data sets. |
| **Pre-train testing** | Training set, hyperparameters, and empty, initialized ML model (e.g. neural net). | Test results and detailed info if available, and baseline models. | Detecting errors and bugs that would render a training job useless before said training job is initiated. Also, provide baseline models. | Reducing the amount of unnecessary training jobs saves time and power. Baseline models indicate expected performance level. |

Table 5.1: Description of ML components in fig. 5.1

| Component Descriptions | | | | |
|---|---|---|---|---|
| **Component** | Inputs | Outputs | Purpose | Why is it important? |
| **Training & validation** | Training set, validation set, hyperparameters, and the same empty, initialized ML model used in pre-train testing. | A trained model. | Building the actual ML model. | The component that produces an ML model. The other components ultimately facilitates this component. |
| **Hyper-parameter tuning** | None, or possibly validation results | Hyper-parameters | Provide different combinations of hyperparameter values | Hyper-parameters affect the performance of the model. Several values should be tested to ensure optimal or near optimal model performance. |
| **Model evaluation and testing** | Trained model and test set. | Evaluation metric scores (e.g. accuracy) and test results. | Evaluating how well the trained model performs and ensuring correctly learned logic. | Exposing the trained model to unseen data is necessary in order to ensure that the model behaves as intended. |
| **Review and approval** | Evaluation metric scores and test results. | Approval or disapproval of model. | Investigate evaluation metric scores and test results and consider whether they are satisfactory. | Ensure correct logic and certain performance level before predictions can be trusted. |

Table 5.2: Description of ML components in fig. 5.1

| Component Descriptions | | | | |
|---|---|---|---|---|
| **Component** | Inputs | Outputs | Purpose | Why is it important? |
| **Deployment** | The approved ML model | The ML model implemented in a form where it can be used and supported (e.g. an API endpoint) | Making the ML model operative, so that it is available for use by the system as a whole. | Puts the model into production where it is integrated into the entire online system. |
| **Prediction** | Pre-processed unseen, real-world data. | Model predictions. | Use the trained model to make predictions on real-world data. | This component performs the predictions that make up the ultimate goal of the ML system. |
| **Monitoring** | Model predictions and metrics (e.g. statistical, performance, business). | Notifications/ alerts if metric values exceed pre-defined thresholds. | Detecting faults or errors that occur in production or deteriorating model performance. | Provides insight to help determine when the model should be retrained/ remodelled. |
| **Visualization** | Model predictions and metric values. | Illustrative/ visual representations of model predictions and metrics. | Offering insight about how the model behaves to the team behind it. | The team behind the model must understand how it behaves, e.g. for analytical, compliance and business purposes. |

Table 5.3: Description of ML components in fig. 5.1

## 5.1 Data Operations

Data gathering, data pre-processing and data set building are components that deal with data management. These are important as they lay the foundation for the machine learning model and impact its potential. The boundaries between these components are not universally defined, e.g. run:ai [91] includes aggregating data from several sources into a single data set as part of the data gathering component, while Mohandas [44] considers it part of data pre-processing. In this thesis, aggregating data is part of data pre-processing, as described in table 5.1.

The most pivotal part of data gathering is making sure that the data collected is comprehensive enough to train a good model and representative enough of the data that will be encountered in production.

The purpose of data pre-processing is to prepare the data for the succeeding steps in the pipeline. At its core, it is aggregating data from several sources into one single data set, cleaning data to ensure that all feature values are in the same format and of the same data type, and filling in or omitting missing values. Further operations may include extracting data, using only what is necessary, scaling data, and engineering extra features that may be beneficial for the training process. Feature engineering is also mentioned in section 4.2.1.

Data set building, also referred to as splitting, splits the pre-processed data set into a training set, a validation set, and a test set. The training set is used to train the machine learning model. The validation set is used to estimate the accuracy of the machine learning model, and to tune hyperparameter values. The test set is used for final evaluation of the performance of the model. Mohandas [44] specifies a set of criteria for building these data sets:

- Each data set should be representative of data that the machine learning model will encounter in production.
- Output values should be equally distributed across all data sets.
- Data should be shuffled in a way that prevents input variance.
- Random shuffles should be avoided if this can lead to data leaks, e.g. in time series.

All of these operations have further research potential [92, 44].

## 5.2 Training, Validating and Refinement

Training involves feeding data into the algorithm, which adjusts itself to best fit the data by altering its internal learnable parameters. Training proceeds quite straightforwardly, assuming all pre-training tests are passed. The aspects subject to pre-train testing can be monitored during the actual training as well, to ensure that everything runs smoothly.

To ensure that the trained model is satisfactory, it must be validated. This is done using a validation set.

Since it can be difficult to form a notion about how well a model should perform, comparing multiple models is a favourable approach. Several models, with different settings or different training data, can be trained and compared, and the most promising one selected for use.

One technique for doing this is cross-validation, mentioned in chapter 2, where the training set is divided into several parts, and each of these parts are in turn used as the validation set, while the remaining parts are used for training.

Another comparison, which is natural to conduct, is with previous models. If training is initiated as part of retraining, the new model should be compared with the old one to ensure that it is an improvement. In addition to previous models, comparisons can be made with *baseline models*. Baseline models are simple models that provide reasonable results and does not require much time to build [93]. Linear regression and logistic regression are some examples of baseline models. Baseline models give a lower boundary for how well the trained machine learning model should perform. If the trained model does not provide significantly improved performance to that of the baseline models, one should reconsider the structure of the model, or whether the problem at hand requires machine learning to be solved at all. Baseline models can be produced early on, e.g. in the pre-train testing component, and used for comparisons during validation.

As discussed in chapter 2, bias and variance are essential metrics for validation. However, several metrics should be employed to obtain more comprehensive insights. Metrics used for final model evaluation, discussed in section 5.3, are also applicable for validation.

In section 4.2.1, AIC and BIC are mentioned. These criteria were developed to assess the relative quality of statistical models, and have been adopted for machine learning model selection [94, 95, 96, 97]. AIC (Akaike Information Criterion) is a metric developed by Akaike [98]. The core concept is to penalize the inclusion of additional variables to a model, countering overfitting. A lower AIC implies a better model. AIC is most widely used for comparing models, rather than evaluating them in isolation [99, 100]. BIC (Bayesian Information Criterion) is a variant of AIC, developed by Schwarz [101], where the penalty for additional variables is increased. AIC and BIC are computed by eqs. (5.1) and (5.2), respectively, where $k$ denotes the number of free parameters in the model, and $\hat{L}$ is the maximum value of the likelihood function for the model. $n$ is the number of observations, i.e. the number of examples in the validation set.

$$AIC = 2k - 2\ln(\hat{L}) \tag{5.1}$$

$$BIC = k\ln(n) - 2\ln(\hat{L}) \tag{5.2}$$

More detailed information about applications of and intuition behind AIC and BIC are found in [94, 95, 99, 100, 96].

Although a model has been singled out based on these comparisons, it is still room for tuning it at a lower level.

#### Hyperparameter tuning

In addition to the internal parameters that are learned throughout the training process, the model is defined by a set of hyperparameters; learning rate, regularization parameters, etc. Hyperparameter optimization (HPO) is the process of choosing the optimal set of these hyperparameters [22].

There are several HPO techniques; some are mentioned in chapter 2, and a selection is

briefly explained/repeated here:

- *Grid search* is the most basic one [102]. In grid search, a finite set of values is specified for each hyperparameter, and the grid search algorithm evaluates the Cartesian product of these sets to find the most optimal combination of hyperparameters.
- *Random search* samples combinations of hyperparameter values at random a specified amount of times. This is especially effective compared to grid search when some hyperparameters have much more impact than others [102, 103], which is often the case [103, 104].
- *Bayesian optimization*, as explained by Feurer and Hutter [102], is an iterative algorithm with two key characteristics: a probabilistic surrogate model and an acquisition function to decide which point to evaluate next. In each iteration, the surrogate model is fitted to all observations of the target function made so far. The acquisition method, using the predictive distribution of the probabilistic model, then determines the utility of different candidate points. This includes a trade-off between exploration and exploitation.

Since tuning, i.e. parameter fitting, neural networks are blackbox[1] functions [105, 106], these, and most HPO techniques involve multiple training cycles of the machine learning model [22]. The HPO also grows exponentially with the number of hyperparameters, as each new hyperparameter adds a new dimension to the search space. This results in HPO techniques being very expensive and resource-heavy in practice, especially for deep learning applications, as discussed by Yang and Shami [107].

In addition to providing methods and heuristics for obtaining good hyperparameter settings, HPO techniques lay the foundation for automating hyperparameter tuning, reducing the need for human interaction [102]. The computations are equally resource-heavy when automated, but resources are often cheaper than data scientists and engineers, and automated computations can execute 24/7, resulting in a more effective refinement process. Another benefit of automated HPO is that it is more reproducible than manual search [102].

Hyperparameter optimization and selection are active and broad areas of research with significant importance for machine learning [102, 107, 103, 108].

## 5.3 Model Evaluation

The component constitutes both testing and evaluation. However, since testing is discussed in chapter 3 this section covers model evaluation.

Evaluating the model constitutes measuring how well it performs with respect to a selection of metrics designed to provide insight into the model.

[1] When there is a function that we cannot access but we can only observe its outputs based on some given inputs, it is called a black-box function. On the other hand, black-box optimization (BBO) deals with optimizing these functions. Tuning of large neural networks is considered as an example of these functions. [105]

#### Evaluation metrics

Ng et al. [4] mentions in chapter 2 that scoring models with a single metric make it easier to compare them with each other. However, that single metric is usually a combination of multiple metrics, weighted in the way that best represents the model and the requirements. Which metrics are most important also depend on the application of the machine learning model.

What metrics to use depend on what type of model is being evaluated. In a classification model, there are four types of outcomes [109, 110]:

- *True positives* involve correctly predicting that an observation belongs to a class.
- *True negatives* involve correctly predicting that an observation does not belong to a class.
- *False positives* involve wrongly predicting that an observation belongs to a class.
- *False negatives* involve wrongly predicting that an observation does not belong to a class.

From [109, 110], common metrics for classification models include:

- *Accuracy* is the percentage of correct predictions for the test data.

$$accuracy = \frac{correct\ predictions}{all\ predictions}$$

- *Precision* is the fraction of true positives among all predicted positives, in a specific class.

$$precision = \frac{true\ positives}{true\ positives\ +\ false\ positives}$$

- *Recall* is the fraction of true positives among all actual positives, in a specific class.

$$recall = \frac{true\ positives}{true\ positives\ +\ false\ negatives}$$

- *Specificity* is the fraction of true negatives among all actual negatives.

$$specificity = \frac{true\ negatives}{false\ positives\ +\ true\ negatives}$$

- *F1 score* is the harmonic mean of precision and recall. A drawback of this is the equal importance given to precision and recall. This can be adjusted with a weighted form of F1 score.

$$F1\ score = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}$$

- *PR curve* is the curve between precision and recall for various threshold values. AUC (Area Under Curve) is often appended to this - the higher the AUC, the better. Figure 5.2 illustrates a PR curve on the right-hand side.

- *ROC curve* (Receiver Operating Characteristics) is the curve plotted against *recall* (true positive rate) and $1 - specificity$ (false positive rate). AUC is used often also in this context, and intuitively, the higher the AUC, the better. Figure 5.2 illustrates a ROC curve on the left-hand side.

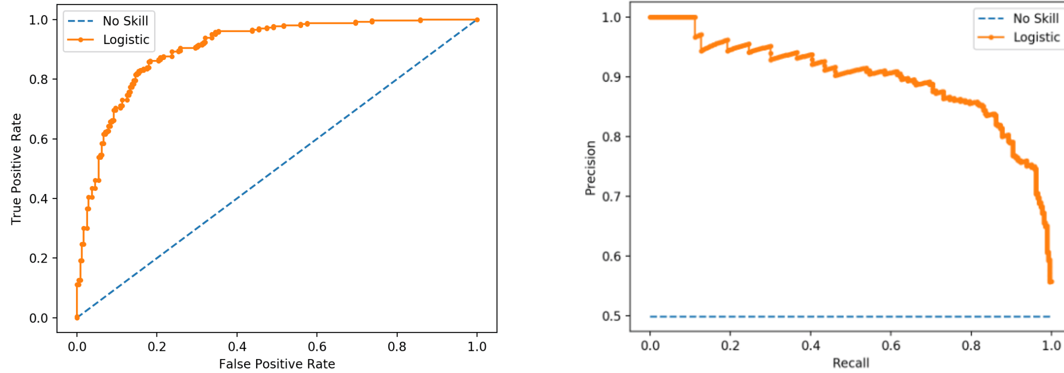



Figure 5.2: To the left: a ROC curve, to the right: a PR curve. Both comparing a logistic regression model with a no skill classifier. Illustrated by Brownlee [111].

Regression models operate in the continuous domain and are not eligible for the metrics applied to classification models. Many metrics for regression models include the *error* of a prediction, which is the difference between the predicted value and the actual value: $e = y_{pred} - y$, where $e$ is error, $y_{pred}$ is the predicted value, and $y$ is the actual value. $n$ denotes the number of data points, and $i$ the $i$-th data point. From [109, 112], common metrics for regression models include:

- *Bias* and *variance* can be evaluated on the test set as well.
- *MAPE* - Mean Absolute Percentage Error.

$$MAPE = \frac{1}{n} \sum \frac{|e_i|}{y_i}$$

- *MAE* - Mean Absolute Error.

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |e_i|$$

- *MSE* - Mean Squared Error.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} |e_i|$$

- *MSE* - Root Mean Squared Error.

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} |e_i|}$$

- *Explained variance* compares the variance within the expected outcomes to the variance in the error of the model.

$$EV = 1 - \frac{Var(y - y_{pred})}{y}$$

- $R^2$ *coefficient* represents the proportion of variance in the outcome that the model is capable of predicting based on its features. $\overline{y}$ denotes the mean of $y$.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - y_{pred_i})}{\sum_{i=1}^{n}(y_i - \overline{y})}$$

The aforementioned metrics are purely statistical, and even though they can be applied in different variations, they provide only a mathematical insight into the model. Paleyes et al. [22] claims that simply measuring the accuracy of the model is not sufficient to understand its performance, and that performance metrics should reflect audience priorities. More specific metrics, such as KPIs [2] and other business-driven measures, should be defined and measured [22].

## 5.4 Deployment

The deployment and integration step in the four-step model from Sweenor et al. [78], discussed in section 4.2 is also relevant for this component. This section elaborates further on deployment.

For an application to be able to use the machine learning model, it needs to be deployed. Deploying a model involves taking it from development and turning it into an executable form, in which it can be integrated into an application or another external system. Sweenor et al. [78] and Breuel [21] recommend decoupling the model and the application that it will integrate to. Updates to the application and the model are invoked on different schedules, and updating one should not consequently require an update of the other. Decoupling the two also results in increased modularity, of which advantages are discussed in section 4.1.

DevOps focuses on techniques and tools for deploying and maintaining traditional software systems, and though some principles can be applied directly to operationalizing machine learning, there are several unique challenges [22].

A widespread topic in traditional software, that machine learning can benefit from is code reuse. Reusing data and models can result in savings in terms of time, effort or infrastructure [22]. Paleyes et al. [22] presents Pinterest as an illustrative case in their approach towards learning image embeddings [113]. Pinterest uses three models internally, which use similar embeddings. To facilitate individual iterations on the models, they were initially maintained completely separately. This naturally posed problem as the effort put into working on these embeddings was tripled. To cope with this, the teams investigated the possibility of learning a universal set of embeddings. This they achieved, and reuse in this manner resulted in simplifying their deployment pipelines and improving performance on individual tasks.

Paleyes et al. [22] points out how there might appear to exist a clear separation of responsibilities between researchers/data scientists and software engineers; the former produce

[2] A Key Performance Indicator (KPI) is a type of performance measurement that demonstrates the success of an organization's activity or objective.

the model while the latter build infrastructure for it to run on, while in reality their areas of concern often overlap in deployment. Considering model inputs and outputs and performance metrics, contributors from both disciplines work on much of the same code. It is thus beneficial to integrate the data scientist team into the development process so that they acquire ownership of the product code at the same level as the software engineers. This approach has been proven to produce long term benefits regarding speed and quality of product delivery, despite posing onboarding challenges [114].

For further and more detailed information about deployment, the reader is referred to [83, 115, 114, 82].

## 5.5 Monitoring

The monitoring step in the four-step model from Sweenor et al. [78], discussed in section 4.2 is also relevant for this component. This section elaborates further on monitoring.

When deployed and operative, the model must be kept under watch continuously to ensure that it is performing at a satisfactory level. Monitoring involves keeping track of several metrics and, based on their values, decide upon when the model needs an update. The monitoring process should also provide insight that facilitates the detection of faults or errors that may occur unpredictably in the production environment.

The process of defining metrics is overlapping in the monitoring and the testing phase. It stands to reason that most of the metrics specified to evaluate the model in terms of deployment adequacy are also subject to monitoring when the model is in production. Threshold values are defined for these metrics, and procedures are implemented to trigger alerts when metric values are observed below their respective thresholds, which indicates that it is time to update the model.

Klaise et al. [116] directs attention to the importance of outlier detection to flag predictions that cannot be used in production. The paper mentions two reasons behind the occurrence of such predictions; the models being unable to generalize outside of the training set, and overconfident predictions on out-of-distribution instances due to poor calibration. Another word for the former is *extrapolation* - obtaining higher-dimensional insights from lower-dimensional training - which is infamously difficult in machine learning [117, 118, 119, 120]. Figure 5.3 illustrates a simple example of extrapolation.

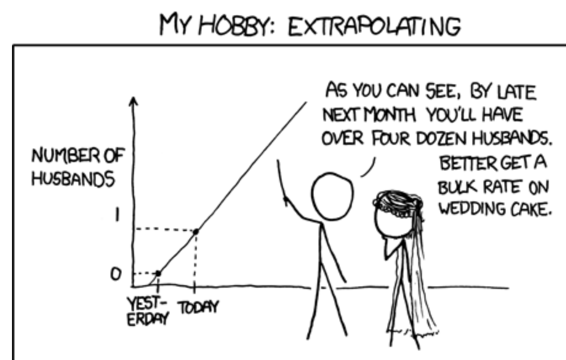



Figure 5.3: The challenge of extrapolating, as illustrated by Munroe [121].

Regular model updates are often required to ensure that it reflects the most recent data trends and the environment [22]. Multiple techniques exist for adapting models to new

data, e.g. regular training and continual learning [122].

The decision to update models in production is also affected by practical considerations, particularly concept drift, which directly impacts the quality and frequency of the model update procedure [22]. The phenomenon can have significant adverse effects on model performance, e.g. in classification problems [123], or in AutoML context [88].

Concept drift can arise in and affect a variety of industries, e.g. in the finance industry [124], marine images [125], and predictive maintenance for wear and tear of industrial machinery [126].

Ackermann et al. [127] highlight a problem with currently available end-to-end machine learning platforms; the final machine learning solutions are usually so sensitive to problems specifics that their needs are not fulfilled by out-of-the-box tools.

For further and more detailed information about monitoring, the reader is referred to [128, 127, 22].

Chapter 6

# Modern Technologies for Machine Learning Operationalization

This chapter presents a selection of modern technologies that offer solutions intended to simplify aspects of operationalizing machine learning systems. There exist a plethora of technologies, and it is difficult to evaluate at first glance which ones are worthwhile delving deeper into. Many technologies apply a buzz-word oriented terminology, and occasionally invent their own words to describe what they offer, which may cause some confusion or information overload. Dissecting formulations and extracting information for comparisons on general grounds is difficult and extremely time-consuming, and by intuition; the ones that convey themselves in an unintelligible manner are perhaps the ones who have the least to offer. The technology reviews are mainly based on the respective technologies' official websites, as well as various articles and blog posts weighing in on them. Personal perceptions of the technologies are further expressed in chapter 8.

## 6.1 Docker and Kubernetes

Kubernetes and Docker are two well-established technologies in software system operationalization. They are not tailored for machine learning systems but often used in combination with or as a basis for other technologies. Due to their renown and them not being machine learning-specific technologies, their reviews are limited to contain only basic functionality.

### 6.1.1 Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables the separation of applications from infrastructure while managing them in a similar fashion, so software can be delivered quickly [129]. It is basically a container management engine, creating containers with automated application deployment on top of operating systems. A container is a software unit that packages up, i.e. containerizes, code and all its dependencies, enabling the application to run quickly and reliably on different computing environments [130].

### 6.1.2 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [131]. It is often used in combination with Docker, where Docker containerizes applications, and Kubernetes manages them. The applications are deployed across a cluster of machines. The cluster is controlled by a *master* server, and *nodes* denote the other servers in the cluster. The master compares the desired state (set by the user) to the state of the cluster and decides which nodes to run to obtain the desired state.

## 6.2 Run:AI

The Run:AI software platform aims to enable data science teams to fully utilize all available resources, speeding up machine learning workload execution, both on-premise and in the cloud [132].

The technology involves a software layer that sits on top of the hardware, decoupling data science workloads from the underlying hardware, as in fig. 6.1.

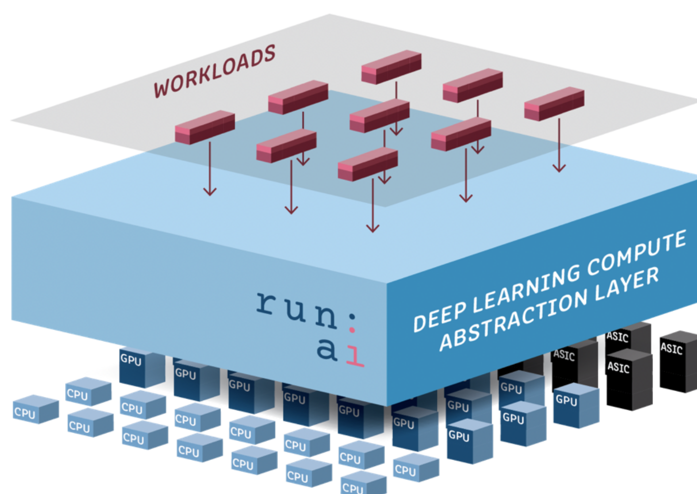



Figure 6.1: Run:AI as an abstraction layer between hardware and AI workloads. Illustrated by [133].

This is reminiscent of virtualization, which initially does not seem effective when the goal is to exploit as much performance as possible from the underlying hardware/AI, intuitively by being close to the metal [133]. In [133], Omri Geller, CEO and co-founder of Run:AI, states: “Traditional computing uses virtualization to help many users or processes share one physical resource efficiently; virtualization tries to be generous. But a deep learning workload is essentially selfish since it requires the opposite; it needs the full computing power of multiple physical resources for a single workload, without holding anything back. Traditional computing software just can’t satisfy the resource requirements for deep learning workloads.”. Run:AI claims to offer a different kind of virtualization, with a rebuilt software stack for deep learning that exceeds the limits of traditional computing, making training faster and cheaper [133].

Run:AI aims to address the diversity of hardware AI by adding support for several deep

learning dedicated chips so that there is no degraded performance when using Run:AI as a third party software layer, instead of directly using what is offered by the chip manufacturers [133].

In [133] Geller also points out that Run:AI addresses the diversity in AI workloads as well. Different types of AI algorithms need to run with different optimizations: different in terms of distribution strategy, hardware chips, etc. Run:AI facilitates this.

Run:AI's virtualization software is based on powerful distributed computing and scheduling concepts from High-Performance Computing (HPC) but is implemented as a simple Kubernetes plugin. The virtualization software speeds up data science workflows and offers visibility. It also enables IT teams to manage expensive resources more efficiently, thus reducing idle GPU time [132].

**Key features**, as presented by Run:AI themselves [132]:

- Pool GPU Compute - pool resources to ensure visibility and control over prioritization and allocation of resources. Heterogeneous resources are pooled so they can be used within two logical environments; *build* and *train*. These environments are tailored for the different characteristics of building and training jobs to increase utilization.
- Guaranteed Quotas - automatic and dynamic provisioning of GPUs to break the limitations of static limitations. Projects are allowed to use more GPUs than their quota allows, to reduce idle resource time.
- Elasticity - dynamically change the number of resources allocated to a job to accelerate data science delivery and increase GPU utilization.
- Kubernetes-based Scheduler - easily orchestrate distributed training with batch scheduling, gang scheduling and topology awareness. Run:AI is easily integrated with Kubernetes as a plug-in, requiring no advanced setup.
- Gradient Accumulation - run training jobs even when there are not enough available resources. Instead of suspending a job, it is shrunk using the elasticity feature. In this state, gradients are accumulated during a training job, enabling the job to run with limited resources.

## 6.3 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform offering high-performance data pipelines, streaming analytics, data integration, and mission-critical applications [134].

Event streaming is analogous to the human body's central nervous system, where applications are always ready to handle incoming events instead of polling at regular intervals [134]. Event streaming ensures a continuous flow and interpretation of data to facilitate the right information at the right place, at the right time [134].

Event streaming is applicable to several use cases, such as payment processing, capturing and analyzing IoT data, and serving as the foundation for data platforms, event-driven architectures, and microservices [134].

Kafka can be deployed in various ways; on bare-metal hardware, virtual machines and containers, and in the cloud or on-premises. Kafka's event streaming consists of three key capabilities:

- To **publish** (write) and **subscribe to** (read from) streams of events. This includes continuous export/import of data between systems.
- To **store** streams of events for as long as desired.
- To **process** streams of events, either as they occur or retrospectively.

Kafka consists of servers and clients that communicate via a high-performance TCP network protocol. Kafka is run as a cluster of one or more servers, where some servers form the storage layer, called the *brokers*. The clients allow for writing distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner. Clients are divided into *producers*, which publish events, and *consumers*, which subscribe to and processes events. A key element for Kafka's scalability is that the producers and consumers are decoupled and agnostic of each other. Events are organized and stored in *topics*.

The **core cabailities** of Kafka, as they present them [134], are:

- High throughput - using a cluster of machines with latencies as low as 2ms to deliver messages at network limited throughput.
- Scalable - production clusters that can scale up to a thousand brokers, trillions of messages per day, petabytes of data, and hundreds of thousands of partitions. Storage and processing can be elastically expanded and contracted.
- Permanent storage - storing streams of data safely in a distributed, durable, and fault-tolerant cluster.
- High availability - allows for stretching clusters efficiently over availability zones or connecting separate clusters across geographic regions.

Kafka [134] defines their **ecosystem** to consist of:

- Built-in stream processing - using event-time and exactly-once processing to process streams of events with joins, aggregations, filters, transformations, and more.
- Connect interface - out-of-the-box Connect interface that integrates with hundreds of event sources and sinks.
- Client libraries - accepts a variety of languages for reading, writing, and processing streams of events.
- Open-source tools - a large collection of open-source tools.

## 6.4 Dataflow

*"Unified stream and batch data processing that's serverless, fast, and cost-efficient."* [135].

Dataflow focuses on addressing the following use cases:

- Stream analytics - making data more organized and accessible from the instant it is generated. Dataflow provides a streaming solution with the resources needed to

ingest, process, and analyze data in real-time streams with fluctuating volume. The solution is abstracted to reduce complexity and make analytics accessible to both data analysts and engineers.

- Real-time AI - offering analytics through AI solutions for anomaly detection, pattern recognition, and predictive forecasting.
- Sensor and log data processing - an intelligent IoT platform provides business insights from the users' global device network.

Dataflow presents their **key features** as:

- Autoscaling of resources and dynamic work rebalancing - data inputs are partitioned automatically and constantly rebalanced to even out worker resource utilization and reduce the effect of "hot keys"[1] on pipeline performance. This data-aware resource scaling contributes to minimizing pipeline latency, maximising resource utilization, and reducing cost per data record.
- Flexible scheduling and pricing for batch processing - get a lower price for batch processing on jobs with flexibility in job scheduling time. Jobs are guaranteed to be retrieved for execution within a six-hour window.
- Ready-to-use real-time AI patterns - enabling customers to build intelligent solutions ranging from predictive analysis and anomaly detection to real-time personalization and other advanced analytics use cases. Real-time reactions with near-human intelligence to facilitate large torrents of events are enabled with Dataflow's real-time AI capabilities. With ready-to-use patterns, real-time AI capabilities allow for real-time reactions with near-human intelligence to large torrents of events.

The business-level benefits that Dataflow [135] claims to provide are:

- Streaming data analytics with speed - enable fast, simplified streaming data pipeline development with lower data latency.
- Simplify operations and management - allowing teams to focus on programming instead of managing server clusters as Dataflow's serverless approach removes operational overhead from data engineering workloads.
- Reduce total cost of ownership - by pairing resource autoscaling with cost-optimized batch processing capabilities, Dataflow can offer "virtually limitless" capacity to manage seasonal and spiky workloads without overspending.

## 6.5 Apache Spark

The descriptions from [136, 137, 138] combined yields the following summary. Apache Spark is an open-source unified analytics engine for large-scale data processing. It handles both batches and real-time streams. Apache Spark can distribute data processing tasks across multiple computers, also known as *clustering*, either on its own or in tandem with other

[1] A hot key is a key with enough elements to negatively impact pipeline performance. These keys limit Dataflow's ability to process elements in parallel, which increases execution time [135].

distributed computing tools, such as Kubernetes. These two qualities are paramount within the world of big data and machine learning. Spark provides easy-to-use APIs that abstract away much of the work associated with distributed computing and big data processing, enabling developers to spend less time and energy on this.

The Apache Spark cluster architecture fundamentally consists of two main components: a *driver* and *executors*. The driver separates the application code into multiple tasks that are distributed across worker nodes. On these worker nodes are executors, which run the tasks assigned to them. To mediate between the components, a cluster manager is required - either Spark's own, or a third party application, e.g. Kubernetes. The architecture is illustrated in fig. 6.2

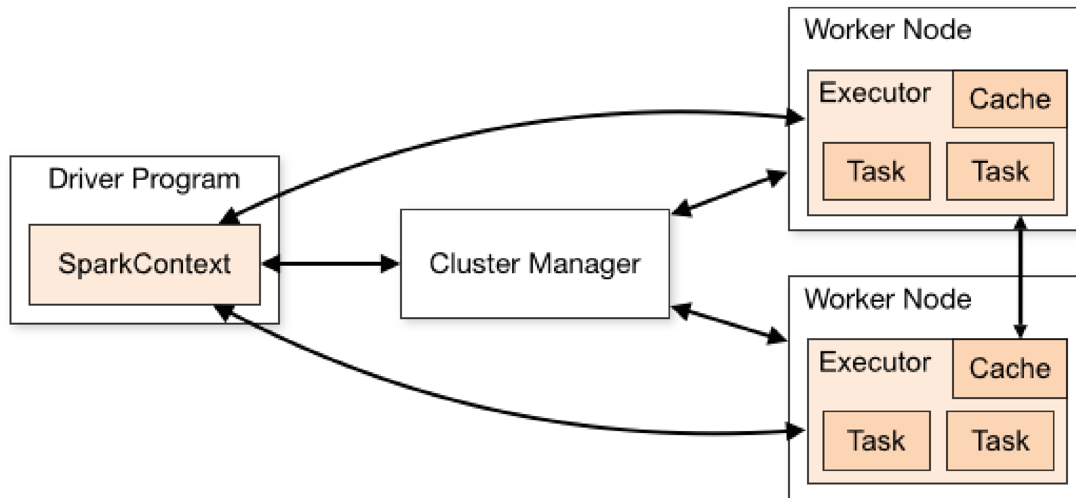



Figure 6.2: Spark cluster architecture. Illustarted by [138].

Apache Spark focuses on providing [136, 138]:

- Speed - Spark provides an in-memory engine that significantly increases the processing speed. This is done primarily by reducing the number of operations consisting of reading or writing to disk.
- Real-time stream processing - Spark can handle real-time streaming combined with the integration of other frameworks.
- Multiple workload support - Spark can run multiple workloads, including interactive queries, real-time analytics, machine learning, and graph processing. Multiple workloads can be combined seamlessly in one application.
- Increased usability - Spark supports applications written in Java, Scala, Python, and R.
- Advanced analytics - Spark supports stream processing, graph processing, machine learning, and SQL queries.

Apache Spark is an extensive technology, with several sub-technologies developed through several iterations with contributions from over 1200 developers [138]. The **key components** of Apache Spark are [136, 137, 138]:

- Spark Core - the heart of Apache Spark, responsible for providing distributed task transmission, scheduling, and I/O functionality.
- Spark MLlib - Spark's machine learning library aims to make practical machine learning scalable and easy. It provides several machine learning algorithms, functionality for featurization, pipelines, persistence - saving and loading algorithms, models, pipelines, etc., and utilities such as linear algebra, statistics, data handling, etc.
- Deep Learning Pipelines - is an extension of the pipelines from Spark MLib. The extended functionality includes: image loading, applying pre-trained models as transformers in a pipeline, transfer learning, distributed hyperparameter tuning, and deploying models in DataFrames and SQL
- Structured Streaming - provides a higher-level API and easier abstraction for writing applications. Streaming computations can be expressed the same way as batch computations on static data. The Spark SQL engine takes care of running it incrementally and continuously and updating the results as data arrives.
- Spark SQL - a module for structures data processing. Includes information about the structure of the data and the computation being performed, which is used to perform extra optimizations. This with the help of a query optimizer called Catalyst that produces an efficient query plan for performing the required calculations across the cluster.
- Spark Graph X - module with a selection of distributed algorithms for processing graph structures. Allows for performing graph operations on dataframes, including taking advantage of Catalyst for graph queries.

## 6.6 MLflow

MLflow is an open-source platform for managing the end-to-end machine learning life cycle. This includes experimentation, reproducibility, deployment, and a central model registry [139].

MLflow is very versatile and integrates with a large number of other technologies. It works with any machine learning library and language since all functions are accessible through a REST API and CLI. MLflow runs the same way in any cloud. Through integration with Apache Spark, it scales to big data [139].

According to their website [139], MLflow's **key features** are:

- MLflow Tracking - track experiments to record and compare parameters and results. The tracking component enables extensive tracking of parameters, code versions, metrics, and output files, which can be visualized at will. Available through UI as well as API.
- MLflow Projects - package machine learning code in a reusable, reproducible form to share with other data scientists or transfer to production. Accessible through an API and CLI, the Project format includes tools for running projects, enabling chaining together projects to form workflows. MLflow Projects are, at the core, just a convention for organizing and describing code.

- MLflow Models - manage and deploy models from a variety of ML libraries to a variety of model serving and inference platforms. MLflow Model packages the machine learning models with an `MLmodel` file that defines the *flavour* the model should be viewed in. This allows the models to be easily shipped to various downstream tools. MLflow models can be hosted as REST endpoints.
- MLflow Registry - a centralized model store for managing models' full life cycle stage transitions; from staging to production. It offers model versioning and annotations and can assign different stages to distinct model versions. It is available both through a set of APIs and through UI.

## 6.7 Databricks

In their own words [136]; "Databricks is an environment that makes it easy to build, train, manage, and deploy machine learning and deep learning models at scale. Databricks integrates tightly with popular open-source libraries and with the MLflow machine learning platform API to support end-to-end machine learning lifecycle from data preparation to deployment."

The creators of Databricks are behind several other technologies, such as Apache Spark and MLFlow, which are discussed in sections 6.5 and 6.6, respectively. Databricks mainly focuses on integrating with these and other technologies, and offer extensions that are built on top of these. These extensions are not open source and must be purchased.

Databricks extends Apache Spark with:

- Databricks Runtime ML - built on Databricks Runtime and is a ready-to-go environment optimized for machine learning and data science. It automates the creation of a cluster optimized for machine learning. Databricks Runtime is a highly optimized Apache Spark engine, running on auto-scaling infrastructure. It has added some components and updates to Apache Spark to improve the usability, performance and security of big data analytics.
- Managed Delta Lake - aiming to simplify data architectures by unifying data, analytics and AI workloads on one platform.
- Integrated workspace - for developing models, and offering real-time collaboration. Allows for publishing notebooks as interactive dashboards, and provides one-click visualizations.
- Workflow automation - containing APIs to build workflows in notebooks. Also offers production streaming with monitoring.
- Enterprise security - end-to-end data security and compliance. Provides access control for notebooks, clusters, jobs, and structured data. Offers audit logs and data encryption.
- More integrations - via authenticated ODBC/JDBC, REST APIs, and data source connectors.
- Expert support

Managed MLflow is the extended version of MLflow which offers:

- Notebooks and workspace integration
- Scalable cloud/clusters for project runs
- ACL-based stage transition in model management.
- Built-in batch interference and stream analytics
- Security and management

Chapter 7

# Miniature Machine Learning System

This chapter explains the development of a miniature machine learning system. The system is designed to run locally, and besides its purpose as a learning resource, it can function as a low-effort testing tool for evaluating techniques and tools for operationalizing and automating machine learning systems, with a short feedback loop. The deep neural network in the system models an industrial process, the van de Vusse reactor, by training on data generated from a developed simulator. Takeaways from the development of this system is discussed in chapter 8.

## 7.1 System Specification

The machine learning system should fulfil the following criteria:

- Include a deep neural network and procedures to train and test it.
- Means for producing large amounts of data, and processing it.
- The neural network should model an industrial process, and make predictions based on time series data.
- The system as a whole should be configurable, and a user who is unfamiliar with the code should be able to make desired changes.
- Visualization should be available for relevant components.
- Have a modular architecture, i.e. consists of several single-purposed modules that can be combined in pipelines.

## 7.2 Design Choices

Justifications and choices are discussed in the succeeding sections, where relevant. This section provides a summarized overview of the design choices made for the system.

Python is chosen as the programming language, due to familiarity and to it being a highly popular language for machine learning development [140]. Together with Python, Conda

[141] is used as a package management system. This is a measure taken to increase the portability of the system. Pytorch [142] is a machine learning framework that accompanies Python and simplifies many machine learning-related operations. Further mathematical functionality is provided by Numpy [143], a Python library.

Configuration is made possible through `config.json`. `json` is intuitive, easily configurable, and easy to both integrate and manage. The simulator is developed using the CasADi framework [144], which is discussed in section 7.4.2.

Visualization is an important tool for obtaining insight into the different components/modules of the system, and to ensure that they function as expected. An effort has thus been put into developing generalized plotting functionality that can support multiple use cases. The visualization functionality largely relies on matplotlib [145], a comprehensive library for creating visualizations in Python.

A `README.md`, explaining how to use the system is found in appendix A

## 7.3 Configurability

It is desirable that the machine learning system is configurable so that functionality can be altered, different testing scenarios can be performed, and the performance of the model can be attempted enhanced, easily, and without requiring familiarity with the code.

The configurable aspects are mentioned when relevant in the succeeding sections. The means for configuration, a file named `config.json` is found in appendix A.

## 7.4 Process Simulator for Data Generation

To create a system that can be tested quickly and frequently, it is important to have a vast, or preferably infinite, source of data. The simulator does not represent the perfect real-world dynamics of a CSTR with a van de Vusse reaction, due to simplifications, but the dynamics are reasonably realistic. This does not matter in this context, as the purpose is to investigate the machine learning system itself, and not to provide the best possible model for the CSTR. The implementation of the simulator is shown in appendix A, in `simulate_vandevusse.py`.

### 7.4.1 CSTR with van de Vusse reaction

A continuous stirred-tank reactor (CSTR) is a common model for a chemical reactor in chemical engineering and environmental engineering. A CSTR often refers to a model used to estimate the key unit operation variables when using a continuous agitated-tank reactor to reach a specified output. The mathematical model works for all fluids: liquids, gases and slurries.

A van de Vusse reaction is highly nonlinear and shows exotic behaviour, such as the inverse response [146]. For this reason it is an interesting topic to investigate in terms of control and stabilization [147, 148, 149, 150]. Hajizadeh and Hosseini [151] presents an artificial neural networks based control of CSTRs with the van de Vusse reaction, and Luz and

Santos [152] presents an approach for predicting the dynamics of this process with neural networks. This motivated the use of the van de Vusse reaction in this thesis.

In the van de Vusse reaction, cyclopentenol ($B$) is produced from cyclopentadiene ($A$), with the formation of cyclopentanadiol ($C$) and dicyclopentadiene ($D$) as byproducts, according to the reactions in eq. (7.1) [146]. As described by Engell and Klatt [153], the reaction occurs in a jacketed CSTR reactor, due to the exothermic nature of the reaction.

$$\begin{aligned} A &\xrightarrow{k_1} B \xrightarrow{k_2} C \\ 2A &\xrightarrow{k_3} D \end{aligned} \tag{7.1}$$

$A$ is fed into the reaction. $B$ is desired species. $C$ and $D$ are undesired species, byproducts. $k_1$, $k_2$ and $k_3$ are reaction rates.

As per Luz and Santos [152], considering the constant density throughout the reactor and ideal level control, for simplicity, the dynamics of the system is described by the differential equations in eqs. (7.2) to (7.7), resulting from the mass and energy balance of the reactor and the cooling jacket. The reaction rates are shown in eq. (7.8), parameters for the process are listed in table 7.2, and state and input variables are listed in table 7.1.

$$\frac{dC_A}{dt} = \frac{F_{in}}{V_R}[C_{A_{in}} - C_A] - k_1(T)C_A - 2k_3(T)C_A^2 \tag{7.2}$$

$$\frac{dC_B}{dt} = -\frac{F_{in}}{V_R}C_B + k_1(T)C_A - k_2(T)C_B \tag{7.3}$$

$$\frac{dC_C}{dt} = -\frac{F_{in}}{V_R}C_C + k_2(T)C_B \tag{7.4}$$

$$\frac{dC_D}{dt} = -\frac{F_{in}}{V_R}C_D + k_3(T)C_A^2 \tag{7.5}$$

$$\begin{aligned} \frac{dT}{dt} &= \frac{F_{in}}{V_R}[T_{in} - T] + \frac{k_W A_R}{\rho Cp V_R}[T_k - T] \\ -\frac{1}{\rho Cp}&[k_1(T)C_A\Delta H_1 + k_2(T)C_B\Delta H_2 + k_3(T)C_A^2\Delta H_3] \end{aligned} \tag{7.6}$$

$$\frac{dT_k}{dt} = \frac{Q_k}{m_k Cp_k} + \frac{k_W A_R}{m_k Cp_k}[T - T_k] \tag{7.7}$$

$$\begin{aligned} k_1 &= k_{10}10^{-\frac{E_1}{T+273.15}} \\ k_2 &= k_{20}10^{-\frac{E_2}{T+273.15}} \\ k_3 &= k_{30}10^{-\frac{E_3}{T+273.15}} \end{aligned} \tag{7.8}$$

| | Symbol | Variable |
|---|---|---|
| State variables | $C_i$ | Concentration of species $i$ in CSTR $[mol/l]$ |
| | $T$ | Temperature in CSTR $[C°]$ |
| | $T_k$ | Temperature of cooling jacket $[C°]$ |
| Input variables | $F_{in}$ | Inlet feed rate $[l/hr]$ |
| | $Q_k$ | Jacket cooling rate $[kJ/hr]$ |

Table 7.1: State and input variables in a CSTR with van de Vusse reaction

| Parameter | Value | Denotes |
|---|---|---|
| $C_{A_{in}}$ | 5.1 | Inlet feed concentration $[mol/l]$ |
| $T_{in}$ | 104.9 | Inlet feed temperature $[C°]$ |
| $k_{10}$ | $1.287 * 10^{12}$ | A->B Pre-exponential factor $[1/hr]$ |
| $k_{20}$ | $1.287 * 10^{12}$ | B->C Pre-exponential factor $[1/hr]$ |
| $k_{30}$ | $9.043 * 10^{9}$ | 2A->D Pre-exponential factor $[1/hr]$ |
| $E_1$ | 9758.3 | A->B Activation Energy $[K]$ |
| $E_2$ | 9758.3 | B->C Activation Energy $[K]$ |
| $E_3$ | 8560 | 2A->D Activation Energy $[K]$ |
| $\Delta H_1$ | 4.2 | A->B Heat of Reaction $[kJ/molA]$ |
| $\Delta H_2$ | $-11$ | B->C Heat of Reaction $[kJ/molB]$ |
| $\Delta H_3$ | $-41.85$ | 2A->D Heat of Reaction $[kJ/molA]$ |
| $\rho$ | 0.9342 | Density $[kg/l]$ |
| $Cp$ | 3.01 | Heat capacity of reactants $[kJ/(kg * K)]$ |
| $k_W$ | 4032 | Thermal conductivity $[kJ/(h * K * m^2)]$ |
| $A_R$ | 0.215 | Area of jacket cooling $[m^2]$ |
| $V_R$ | 10 | Reactor volume $[l]$ |
| $m_K$ | 5 | Mass of cooling $[kg]$ |
| $Cp_K$ | 2 | Heat capacity of cooling $[kJ/(kg * K)]$ |

Table 7.2: Parameters in a CSTR with van de Vusse reaction

### 7.4.2 Modelling framework

CasADi [144] and Python are the framework and programming language used to develop the simulator. CasADi is an open-source software tool for nonlinear optimization and algorithmic differentiation, with functionality that exceeds the requirements for the simulator. The complete simulator implementation can be found in appendix A.

One of the core aspects of CasADi is the symbolic framework. With this, the user can construct symbolic expressions using a MATLAB inspired everything-is-a-matrix syntax, i.e. vectors are $n \times 1$ matrices, and scalars $1 \times 1$ matrices.

The simulator applies the `sx` symbolics, which is a part of the symbolic framework. The `sx` data type represents matrices whose elements consist of symbolic expressions, and is used for the state and input variables in the plant, as shown in an excerpt from `simulate_vandevusse.py`:

```
x = SX.sym('x', nx)  # Concentration of A, B, C, D, temperature in CSTR, T
↪  and temp of cooling jacket T_k
p = SX.sym('p', nu)  # Feed rate F_in and Jacket cooling rate, Qk
```

u is usually used to denote the input in a control system. When creating integrators in

CasADi, the convention is to use p, representing a set of parameters that affect the ODEs (Ordinary Differential Equations). These parameters include, but are not limited to, input variables. p is therefore used to represent the symbolic matrix, while the actual input values are stored in u, and makes up the values of p at the time of the simulation.

CasADi uses the freely available SUNDIALS suite [154], which contains the two popular integrators CVodes and IDAS for ODEs and DAEs (Differential Algebraic Equations) respectively. This simulator uses CVodes. The integrator is defined simply as shown in an excerpt from simulate_vandevusse.py:

```
xdot = vertcat(dCA, dCB, dCC, dCD, dT, dTk)
ode = {'x':x, 'ode': xdot, 'p': p}
opts = {'tf': dt} # Sets correct step size in integrator
ode_solver = integrator('F', 'cvodes', ode, opts)
```

, where dCA, dCB, dCC, dCD, dT and dTk are the ODEs in eqs. (7.2), (7.3), (7.4), (7.5), (7.6) and (7.7), respectively. vertcat is a function to perform vertical concatenation. 'F' has no functionality other than being a name for the integrator, ode is the set of ODEs for the integrator to solve and opts include specifying the time step of each integration.

The simulation starts by invoking simulate_vdv, defined as shown in an excerpt from simulate_vandevusse.py:

```
def simulate_vdv(x0, u):
    states = [x0]
    for k in range(u.shape[0]):
        res = ode_solver(x0=x0, p = u[k])
        x0 = res["xf"]
        states.append(x0)
    return np.concatenate(states, axis = 1)
```

, with x0 and u as initial state values and input sequence, respectively.

## 7.5 Neural Network

The neural net implementation in this system is based on an example provided by Solution Seeker AS in conjunction with an exercise from the course TTK28 [155] at NTNU in autumn 2020. Modifications have been performed to generalize the net, and the training and testing procedures, to facilitate their usage in this ML system.

The neural net is implemented with the PyTorch framework [142], a popular framework for machine learning development with Python.

The neural net consists of two hidden fully connected layers with 50 neurons each. The training procedure uses Adam optimization, an MSE loss function, and L2 regularization. Hyperparameters, such as the number of training epochs, learning rate and L2 regularization factor are subject to configuration through the config.json file. The input and output columns can also be configured, to alter which features the neural net learns from and which states it should predict.

It is also possible to specify whether or not training should be conducted on a previously trained net, so to train it further.

The complete implementation of the neural net class and the training and testing procedures can be found in appendix A.

## 7.6 Setting up Modular Machine Learning Pipelines

The system consists of several modules which each serve one purpose. Due to the size of the project, and the controlled area it is developed in, not all modules are as comprehensive as described in chapter 4, nor are all the modules included. A training pipeline and a prediction pipeline are defined, but neither is automated; separate modules are invoked manually, sequentially to produce the workflow of a pipeline. The pipelines are illustrated in fig. 7.1.

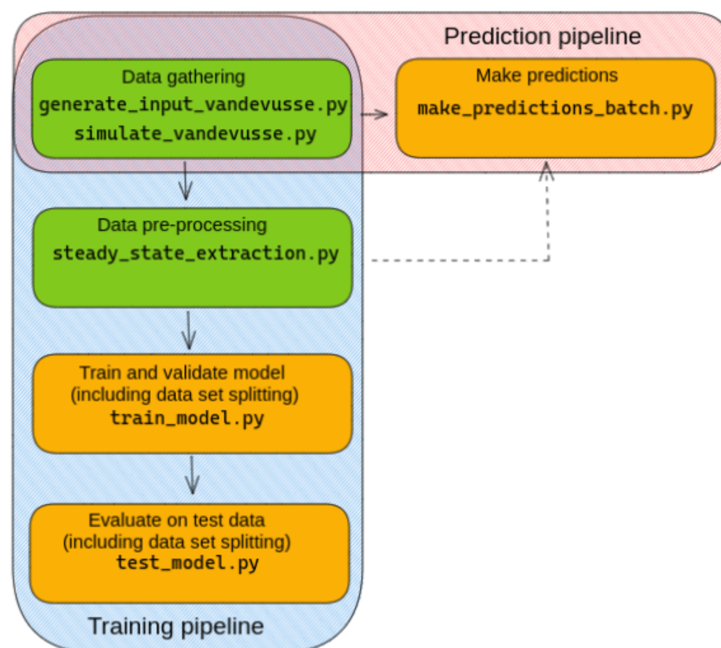



Figure 7.1: Flow of the training and prediction pipeline in the developed miniature machine learning system

### Training pipeline

Data gathering is conducted through `generate_input_vandevusse.py` which generates an input sequence per configured specifications.
`simulate_vandevusse.py` uses the generated input sequence to simulate the CSTR for the specified amount of time. The simulation results and input sequence are stored to be used in data pre-processing.

Data pre-processing in this system does not involve operations to verify the data, as it is not considered necessary in a small, controlled development environment with only one data source. The data pre-processing component consists of
`steady_state_extraction.py` which extracts only data points from periods when the

specified variable is in a steady state. The duration and perturbation thresholds for the steady-state are also configurable. The steady-state data set is then stored for training.

Data set building demands little effort and is done with a few lines of code. As such, this process is performed at the start of the training and testing procedures, using a set of indices to extract the testing set.

After splitting the data set, the training set is divided into batches of specified size, and training ensues with a specified number of epochs. The net is evaluated on the validation set after each epoch, and once more with visual representation after training is completed. Refinement and hyperparameter tuning need to be done manually by the user, through `config.json`.

Upon testing, the data set is loaded, and the test set extracted and fed into the trained neural net, which performs predictions. Visual representation ensues, and metrics for MSE, MAE and MAPE are computed. Reviewing the model is a task subject to human assessment. No behavioural tests are performed on the model.

#### Prediction pipeline

A separate pipeline makes up the prediction workflow. For organizational purposes, the data storage files should be changed, so as not to confuse training data with "production" data. Just as for the training pipeline, the prediction pipeline starts with `generate_input_vandevusse.py` to generate an input sequence. Depending on which features the neural net is trained on, `simulate_vandevusse.py` is invoked as well. If none of the states in the CSTR are used as input features for the neural net, it is not necessary to perform the simulation prior to prediction.

If one only wishes to predict on steady-state data, and avoid transients, the next operation is `steady_state_extraction.py`. If not, no further data pre-processing is necessary. Given that the net is trained on steady-state data, extracting steady state is likely to yield better average accuracy, due to the net not performing predictions on data points that are difficult to predict. The predictions performed, however, will not be any more accurate, and ultimately one only ends up with fewer predictions.

Finally, `make_predictions_batch.py` uses the neural net to perform predictions on the new data and visualizes the results. There is no functionality for performing predictions on an input data stream.

## 7.7 Demonstration

The section demonstrates the training and prediction pipelines.

For this purpose, the configuration file is on the same format as the example in appendix A. The most important settings are:

- The `"input_cols"` which decides the input for the neural network - the features that are trained on. For the demonstration this includes the input variables and the concentration levels of the species in the CSTR.
- The output of the neural net, `"output_cols"`, is the temperature in the CSTR, `T`,

sometimes referred to as Tout. "all_output_cols" is not used, but simply present as an overview of the possible features.

- "further_train_existing_net": "false" decides that a new net is initialized, instead of further training an already trained net.
- "hidden_layers", "training_batch_size", "shuffle_training_data" , "n_epochs", "learning_rate" and "l2_regularization" are hyperparameters that decide the structure and specifics of the neural network.
- "n_iterations" decides the time horizon of the simulation, and "samples_per_hour" determines the sampling time.
- Under "input_generation"→"options" is decided how many iterations an input variable should stay constant for in "input_interval_size", and whether just one or both input variables are changed at each interval is decided by "perturbation_style", where the unused option is "double". The remaining values are self-explanatory.
- Under "data_extraction" is decided the threshold value perturbation for steady-state. the threshold duration for the steady-state, and the steady-state variable.

### Training pipeline

The data gathering step, which invokes generate_input_vandevusse.py and simulate_vandevusse.py, produces the initial data set. The data is plotted in figs. 7.2 and 7.3.

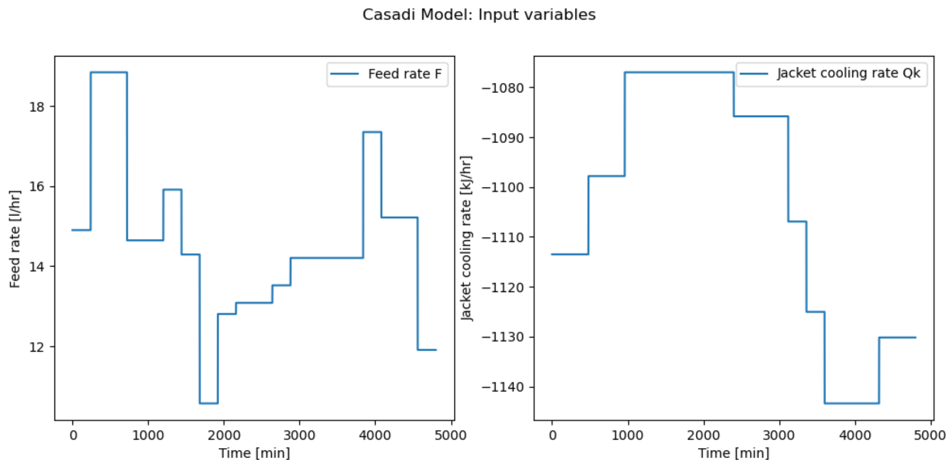



Figure 7.2: Input data for training pipeline.

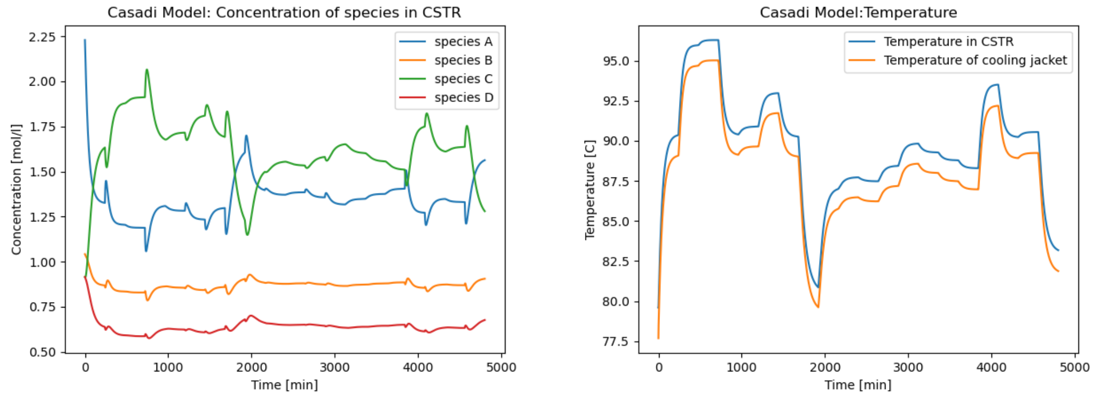



Figure 7.3: States in CSTR for training pipeline.

The next step is to invoke `steady_state_extraction.py`, performing the necessary pre-processing before training is initiated. The state on which steady state is considered is the temperature in the CSTR, `T`. Figure 7.4 shows the states post-extraction.

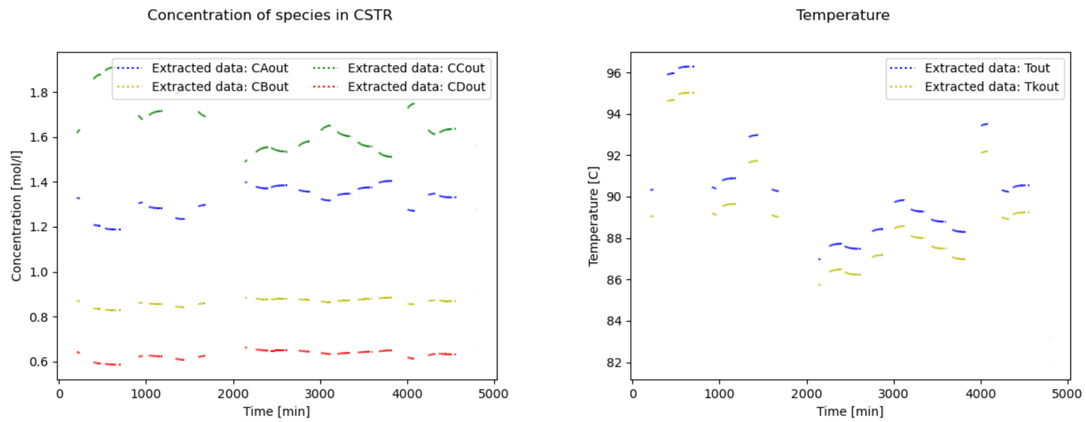



Figure 7.4: States in CSTR extracted at time of steady state for `T`

After pre-processing, training is initiated through `train_model.py`. After training, the trained neural net is subjected to the validation set, resulting in the metric scores shown in fig. 7.5. The validation set consists of random samples from the data set after the training set has been extracted. There is no continuity in the data points, and plotting the predictions is messy and provides little insight.

```
Error on validation data
MSE: 0.4395087957382202
MAE: 0.6225247979164124
MAPE: 0.686882495880127 %
```

Figure 7.5: Metric scores for trained neural net on validation set

The last step in this training pipeline is testing the model with `test_model.py`. Figure 7.6 shows how the trained net performs on the test data.

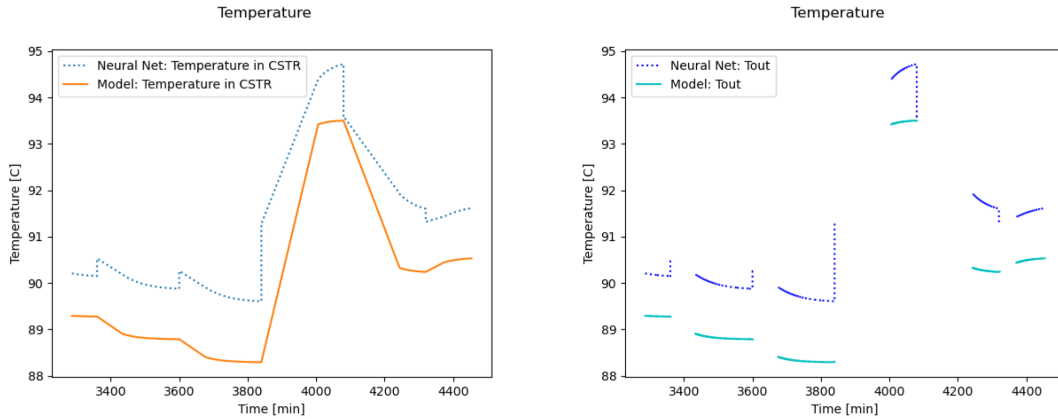



Figure 7.6: Model predictions on test set versus actual values. The left plot shows continuous graphs, while the right plot shows each individual data point, which are only those in steady state.

### Prediction pipeline

The configuration file is changed so that generated data is stored in the "production" data set. After this, the data gathering step is performed to generate new data, shown in fig. 7.7.

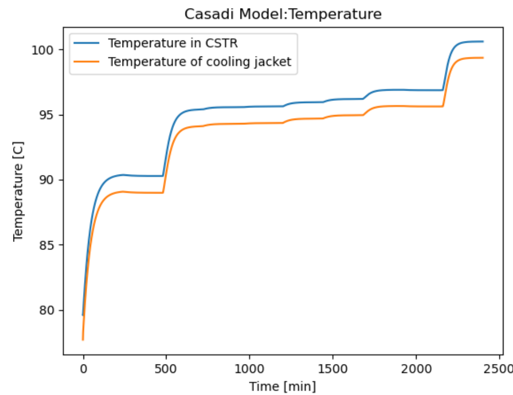



Figure 7.7: Temperature values for the "production" data.

No pre-processing is performed. In this context, the steady-state extraction does not make sense; the extraction is performed on the steady-state of `T`, which is the output of the neural network, and which the value is unknown for in the data subject to predictions. This drawback is discussed in chapter 8. The next step is then to invoke the neural net to perform predictions, through `make_predictions_batch.py`. The predictions performed by the neural net are plotted in fig. 7.8.

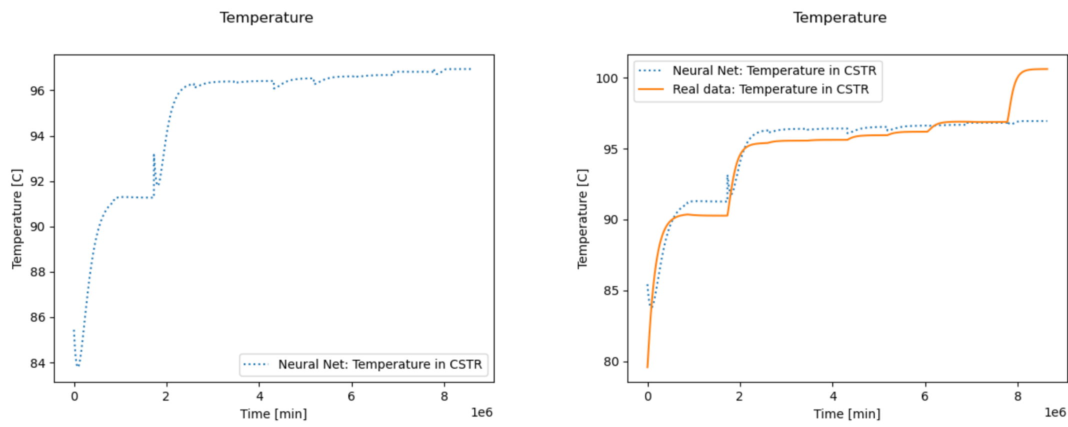



Figure 7.8: Predictions on "production" data performed by the neural network. The leftmost plot shows the isolated predictions. The right hand side shows the predictions compared with the "unknown" actual values.

# Chapter 8

# Discussion

## 8.1 Testing

As systems become operationalized and more processes are automated, testing grows increasingly important to sustain rapid development and reliable operation.

Machine learning models are stochastic and thus difficult to test using conventional methods. An machine learning system, however, consists of several modules that facilitate the machine learning algorithm, but whose behaviour does not rely on the algorithm. Thus, conventional methods for software testing and data validation can be applied to direct the resulting behaviour of the machine learning algorithm as much as possible in the intended direction.

Testing machine learning systems involves testing the behaviour, i.e. the learned logic of the machine learning algorithm. This is inherently difficult because for machine learning algorithms there exists no reliable test oracle to indicate what the correct output is for an arbitrary input. CheckList introduces three types of tests; one (MFT) involves testing against a small set of test cases with known correct output, and the two others (INV and DIR) involve testing how the output changes based on perturbations to the input. CheckList is proven to work well on NLP models and constitutes a very promising methodology. Although there is no documentation, to the best of my knowledge, of the test types being successfully applied to different types of models, it can be speculated that doing so would yield results. This belief is supported by another similar methodology consisting of *niche oracle-based testing* (similar to MFT), *metamorphic testing* (similar to INV and DIR), and *parameterized random testing*, which successfully detected new bugs in systems to which it was applied. The creators behind the methodology have however not been able to determine its adequacy, and the tested systems do not represent deep neural networks, which are proven to respond inefficiently to randomly generated test cases.

It would be interesting to investigate the applicability of these tests on regression models. It would be challenging due to the increased difficulty of producing test cases with known correct output, and to determine expected output change based on input perturbations. The actual outputs would likely also be increasingly difficult to evaluate since the domain is continuous and not binary, or even discrete.

For safety-critical deep neural network systems with a high level of quality assurance,

it is possible to adopt a set of coverage criteria based on the MC/DC criterion develop by NASA. These criteria are developed to verify the learned logic by investigating the behaviour of each individual neuron in a deep neural network using concolic test case generation. Although the experiments performed with the criteria show promising results, the testing procedure is comprehensive and demanding to implement, and likely not worthwhile applying to a general system with more relaxed safety standards.

An experimental methodology suggests using mathematics to interactively verify the machine learning program during development, effectively developing a bug-free system. It might work in specific cases, but is only available with a specifically developed programming language, and appears to slow down the development process significantly, as the creators themselves admit to experiencing a substantial increase in workload. In its current state, the methodology is not deemed practically applicable. Further research might pave the way for a new programming language paradigm that can be useful for machine learning development, but this is not likely to be the case in the foreseeable future.

Instead of extensive testing, one might decide to approach the assurance of desired behaviour by ensuring that only acceptable inputs occur. Good outlier detection can be used to disregard predictions outside the scope in which the system is tested, and thus function as a safety measure or compensation for lack of testing. This may degrade performance, but can in many cases be an acceptable sacrifice.

## 8.2 MLOps

Multiple disciplines are combined to form MLOps. MLOps is the synergy between these disciplines, as opposed to a patchwork. The disciplines in questions are roughly grouped under data science or software engineering. MLOps is prone to become a secondary consideration for all team members instead of an established area of concern on its own. Decisions concerning machine learning systems are thus not weighed into from an MLOps perspective but from the perspective of all its associated disciplines, which are dominated by their primary responsibilities.

A dedicated role in the form of an MLOps engineer addresses this problem by incorporating a persona with sufficient skill in all or multiple MLOps-related disciplines into the team. The MLOps engineer manages tasks that lie on the boundary between disciplines, oversees the processes and should approach decisions with a nuanced MLOps perspective.

An MLOps engineer is definitely valuable. It might, however, not bridge gaps between disciplines completely, but simply create smaller gaps. In addition to incorporating an MLOps engineer, one should strive to achieve a common understanding of the requirements, challenges, and possibilities in the system, allowing the team members to proceed with a unified mindset. The overview in chapter 5 provides the basis for such a unification, tying together software architecture and data science. It is believed that such an overview must be extended by more detailed descriptions and tailored to the system in question.

Employing MLOps with a four-step sequence of build, manage, deploy and integrate, and monitor is claimed to "ensure that the best model gets embedded into a business system and that the deployed models are consistent with business requirements". The claim is a bit flippant, since if it was that easy, then problem solved. Not taking too much away from the proposed sequence, however; it is a clever sequence with well-defined areas of responsibility, and addresses common challenges and advises on solutions. It incorporates

a way of thinking; it is important to approach MLOps systematically and integrate it fully instead of treating it as a separate process. I do not doubt that this sequence is helpful, and perhaps sufficient for many teams, but I would hesitate to give it the status of a generalized blueprint.

In the four-step sequence, deploy and integrate, and monitor are heavily emphasized, while the entire development process is compressed into the building step. One might argue that deployment, integration, and monitoring are of higher importance for operationalization since they represent putting the model into production and making sure it functions properly while there. This is, after all, the ultimate goal of operationalizing - having the model actually influence a system, as opposed to simply providing statistics. My criticism is that deploying, integrating, and monitoring an arbitrary machine learning system, does not yield an operationalized system. MLOps must be facilitated from the early development stage with the creation of well-defined single-responsibility modules that can be tested effectively and integrated to form pipelines. The system must be structured to retrain or remodel at volition, and to support continual champion-challenger processes. It is not that development is completely ignored in the four-step sequence, but it is given a disproportionate amount of attention. The overview from chapter 5 complements the four-step sequence by providing more details about the structure and requirements in the development phase. It is not comprehensive enough to form an out-of-the-box template but directs attention to an important area.

Facilitating MLOps involves designing a sustainable and robust system, which provides necessary operations through easy access and efficient execution. Pipelines provide an intermediate layer between modules and the entire system which results in propitious abstractions. This allows development to proceed by defining necessary pipelines for the entire system, and followingly designing modules to make up these pipelines. Modular pipelines motivate well-defined modules and are advantageous in terms of scalability, maintainability, etc. On a higher level, pipelines contribute to a modular system that allows for decoupling independent processes, such as application code and machine learning models. Having to redeploy the application due to an updated machine learning model, or vice versa, is unnecessary.

Towards the end of this thesis, Coursera [4] released a specialization on MLOps. There was unfortunately no time to complete it, but the fact that it was released emphasizes the relevance of MLOps.

## 8.3 Technologies

It is only so much insight to be gained about a technology through reading about it, as opposed to using it. In the early stages of this thesis, it was considered to employ some technologies with the developed miniature machine learning system, but it was deemed too time-consuming to be fruitful, and most technologies apply to a much larger scale than the developed system constitutes, rendering the idea infeasible. The solutions offered by the technologies do, however, provide some insight into practical challenges in MLOps. The most focused areas appear to be data streaming, computation resources, pipeline support, and analytics and insights. Following are some preliminary reflections on the technologies, based on my perceptions.

Docker and Kubernetes are prevalent in traditional software systems, and are useful also in machine learning systems, complemented by other technologies.

Apache Spark constitutes the most comprehensive technology, covering multiple areas. It has been developed through several iterations and shows characteristics of contributions from a large number of developers in how it has adopted and improved aspects from a variety of technologies. It appears to be at the very forefront with the quality of the solutions it offers, from real-time streaming, clustering, query optimization, deep learning pipelines, etc. It is not clear to what degree one can pick and choose from its solutions, although it appears to be reasonably flexible as it is open-source, and the creators explicitly mention the possibility of integrating Kubernetes as a cluster manager.

Run:AI focuses on efficient computations and auspicious resource allocation for the customer, through what appears to be a novel variation of virtualization. Its features do not extend notably into other areas, and thus Run:AI does not require that you commit to several solutions as a package deal. On the downside, it is not open source or freely available. Given that their solution actually works for maximizing the utilization of resources, and that this a desired feature for the system in question, it seems like a good choice of technology.

Apache Kafka provides the means for incorporating distributed event streaming into your system in a wide variety of ways. It is also open-source. Given that distributed event streaming is a desired feature, Apache Kafka seems to be the go-to technology.

Dataflow focuses on autoscaling of resources and analytics. It is a part of Google Cloud and is not free of charge. I suspect that it is favourable towards integration with other Google products. This might cause seamless integration and constitute a desirable trait for some companies. For other companies, it might constitute bothersome commitment requirements.

MLflow is open source and integrates with several other technologies. It aims to manage the end-to-end machine learning life cycle. To me, what it offers does not seem very comprehensive; just simple additions to the development process through some tracking and packaging tools. It might be that this is a good thing; that it just offers simple additions that make management a little simpler without having to commit to any significant restructuring.

Databricks offers paid extensions to open source solutions, specifically Apache Spark and MLflow. Their extensions do not strike me as particularly useful, including interactive notebooks, which have been pointed out to be difficult to manage and run reliably [21], and tools that do not seem appealing to the average software developer. The extensions do include support and security, which can be advantageous for teams that are unable or unwilling to handle these issues themselves. These “premium memberships” of Apache Spark and MLflow raise doubts about how complete the technologies are in their standard editions.

Upon choosing technologies I perceive it as highly important to consider the capabilities of the development team that will employ them, and which groups in the company they primarily support. The results of using a technology and how enjoyable it is to use are two different aspects. E.g., a technology might offer valuable insights and accessible user-friendly interfaces for analysts or managers, but be difficult to integrate and employ for the developers.

## 8.4 Miniature machine learning system

Developing the machine learning system has highlighted the utility of pipelines, especially in terms of development. Well-defined modules emphasize what a task actually involves, making it easier to develop with sustainability in mind. The modules are also much simpler to debug. Providing input to modules, through a configuration file, allows for the modules to be used in multiple pipelines, reducing the total workload. The challenge of data pre-processing is emphasized through the management of multiple data sets, and how good feature engineering, e.g. steady-state extraction, requires sound mathematical comprehension. Another challenge that was highlighted is creating pipelines to accommodate data streams, as opposed to batches. This might have been manageable through integration with Apache Kafka. Although not all components of a machine learning life cycle were extensively implemented, insight was gained concerning their functionality and complexity.

The machine learning system was developed with the primary intention of investigating the components, and how they integrate. Extensive functionality of each component and performance of the neural network was never the objective. Still, some possibilities for increased functionality have been thought of.

Hyperparameter tuning with some sort of HPO technique could have been implemented and automated in a fashion to provide a better performing neural net. In the current system, the user is required to manually alter the hyperparameter values in the configuration file before initiating the training procedure. This is slow and tedious, and a machine learning system should have a better approach for this.

The steady-state extraction involves extracting data points where a chosen state in the CSTR is in steady state. In the demonstration, section 7.7: the state making up the output of the neural net, $T$, is the selected state. By analyzing the time constant of the system, steady-state extraction could have been performed based on the input variables. This is intuitively not the most efficient use of steady-state extraction. As seen in fig. 7.6, the neural net does not noticeably consider steady-state in prediction.

By deriving an expected time interval between input perturbation and states reaching a steady state, it is possible to extract data points a certain period after the last input perturbation. This is likely to be more useful in an actual industrial process, where input values are subject to direct influence, and thus accurate and continuous observation. This is not the case for the states in the process - even though there are techniques for obtaining estimates of the states, combining measurements and mathematics, these estimates are not as reliable as the values for input variables. Moreover, extracting data based on the observed steady-state of the state subject to prediction, in production, makes no sense, since there is no point in making a prediction if the state is observable. During training, extracting the output variable offers some benefit; given that the time between input perturbations is sufficiently large, extracting steady state data points provides a correlation between input and output values that is easier to train.

It could also have been interesting to attempt to integrate certain technologies into the system, such as Apache Kafka and Docker, to provide an automated event-driven system in a containerized form, which would have provided a more complete product in terms of demonstration and deliverability.

# Chapter 9

# Conclusion

The challenges and difficulties faced when operationalizing machine learning systems are widespread problems. Due to the rapid evolution of machine learning technology and the emergence of big data leading to a significant increase in possible applications for machine learning, the focus has mainly been on increasing potential through better and more diverse machine learning models, rather than realising potential through sustainable development. Operationalizing machine learning systems requires both data scientists and software engineers. Unclear communication between these factions imposes an obstacle for operationalization. The differences between machine learning systems and traditional software systems poses challenges when applying conventional operationalization techniques to machine learning.

Testing the logic of a system requires new techniques when applied to stochastic systems, such as machine learning systems, where there is no reliable test oracle. An approach involving behavioural tests based on a small selection of test cases where an oracle can be derived and metamorphic testing seems promising. A proven example is CheckList which employs tests based on these techniques on NLP models. There are other approaches based on neuron coverage testing and interactive formal verification, where the former is mainly intended for safety-critical deep learning systems, and the latter is too underdeveloped to be considered practical.

## 9.1 Advice for best practices

Incorporating a dedicated MLOps engineer into a cross-disciplinary team helps to define responsibilities and helps bridge the gap between disciplines. The MLOps engineer is ideally proficient in multiple MLOps-related disciplines and is in charge of managing the model life cycle, including version control, approval, testing, deployment, and replacement. The MLOps engineer must make sure the processes proceed in alignment with MLOps principles.

To further bridge the gap between data scientists and software engineers a common understanding of the requirements, needs and challenges associated with different components from all perspectives should be established. A tool for this can be an illustrative overview of the architecture depicting how modules integrate to form pipelines and listing the inputs, outputs and purposes of each module systematically in a manner that is understandable

for all involved disciplines.

Modularity should be obtained with pipelines for defining, and possibly automating, workflows such as training, prediction and deployment. Modularity should be incorporated on a higher level as well, decoupling the machine learning model from the application, so that updates can be performed independently of each other. To achieve the letter, one can think of deployment as a function of how a model is integrated, and integration as an extension of deployment.

Techniques from related disciplines should be used for what they are worth, extensive testing with conventional techniques on components that do not rely on the machine learning algorithm results in higher quality assurance. Applicable deployment techniques from DevOps, e.g. the CI/CD pipeline, contribute to more automated and manageable processes.

Aim for automation. Automated processes make the life cycle easier to manage and, with well-defined testing, increases reliability. Automation also allows for processes to run continuously, enabling more frequent model updates, and more comprehensive hyperparameter optimization.

When choosing technologies, teams should keep their level of skills in mind. Some technologies attempt to tailor the complete machine learning life cycle and focus on features such as integrated notebooks, user interfaces, and visualizations of everything. This might be good for a business whose area of expertise is not cutting edge technology but rather applies some machine learning as a secondary feature of their business. Typically a model which requires little maintenance and is developed by a small team, but offers insights and predictions that are interesting for non-technical staff. Teams and companies that are more technologically advanced and continually look to improve their models and extend their applications should choose flexible technologies, which focus on modular parts and can be integrated into various environments without much setup. Some technologies deemed promising are Docker, Kubernetes, Apache Kafka, Apache Spark, and Run:AI.

## 9.2 Miniature machine learning system

The developed machine learning system constitutes a minimum viable product in the form of a testing tool but would benefit from further development. Developing it offered insight into machine learning pipelines; separating the workflow into multiple pipelines resulted in more structure and overview during development.

It also provided clarity concerning the area of responsibility for each component. Even though not all possible extensions and additions to functionality were fulfilled, insight was gained concerning the challenges and importance of each module, as well as the amount of work required to implement it. Some components were revealed to be relatively straightforward, while other components could have benefited from including more mathematical features.

This thesis attempts to shine a light on the field of MLOps. Gathering information from various literature and providing my interpretation will hopefully provide some clarity and make it easier to navigate through the field. Sculley et al. [128] addresses the technical debt of machine learning, and I also believe that actors in the machine learning field should lift their head above the water more often, and share their experiences and research so that MLOps will see a more prosperous development.

## 9.3 Future Work

Increasing the functionality of the machine learning system in terms of a testing suite:

- Integrate Docker and Apache Kafka and create an automated pipeline. This can enable support for data streams, and create a containerized system that can be deployed, or simply easily run on a new system.
- Implement automated hyperparameter tuning. Looking into HPO techniques and ways to implement and automate them will result in a more comprehensive system.
- Enhance steady-state extraction by extracting data based on the time constant of the system

To further investigate operationalization of machine learning systems:

- AutoML is a discipline that can be useful in terms of operationalization. Some sources are [88, 89].
- Holistic approaches is the name of a section in [22]. It involves thinking outside the box when approaching operationalization of machine learning systems, instead of adopting and modifying techniques from traditional software systems. Data Oriented Architectures (DOA)[156, 157] is an example of such an approach.

# Bibliography

[1] SAS Institute Inc. Machine Learning: What it is and why it matters | SAS, 2021. URL https://www.sas.com/en_us/insights/analytics/machine-learning.html.

[2] Dong Yu, Li Deng, Inseon Jang, Panos Kudumakis, Mark Sandler, and Kyeongok Kang. Deep learning and its applications to signal and information processing. *IEEE Signal Processing Magazine*, 28(1):145–154, 2011. ISSN 10535888. doi: 10.1109/MSP.2010.939038.

[3] Itamar Arel, Derek Rose, and Thomas Karnowski. Deep machine learning-A new frontier in artificial intelligence research. *IEEE Computational Intelligence Magazine*, 5(4):13–18, 10 2010. ISSN 1556603X. doi: 10.1109/MCI.2010.938364.

[4] Andrew Ng, Kian Katanforoosh, and Younes Bensouda Mourri. Deep Learning Specialization, 2021. URL https://www.coursera.org/specializations/deep-learning.

[5] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech and Language Processing*, 20(1):30–42, 1 2012. ISSN 15587916. doi: 10.1109/TASL.2011.2134090.

[6] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel Rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012. ISSN 10535888. doi: 10.1109/MSP.2012.2205597.

[7] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22(12):3207–3220, 12 2010. ISSN 08997667. doi: 10.1162/NECO{_}a{_}00052. URL http://direct.mit.edu/neco/article-pdf/22/12/3207/842857/neco_a_00052.pdf.

[8] Xue-Wen Chen and Xiaotong Lin. Big data deep learning: Challenges and perspectives. *IEEE Access*, 2:514–525, 2014. ISSN 21693536. doi: 10.1109/ACCESS.2014.2325029.

[9] Christian Johansson, Markus Bergkvist, Davy Geysen, Oscar De Somer, Niklas Lavesson, and Dirk Vanhoudt. Operational Demand Forecasting in District Heating

Systems Using Ensembles of Online Machine Learning Algorithms. In *Energy Procedia*, volume 116, pages 208–216. Elsevier Ltd, 6 2017. doi: 10.1016/j.egypro.2017.05.068.

[10] Ryan Lagerquist, Amy McGovern, Cameron R. Homeyer, David John Gagne, and Travis Smith. Deep learning on three-dimensional multiscale data for next-hour tornado prediction. *Monthly Weather Review*, 148(7):2837–2861, 7 2020. ISSN 15200493. doi: 10.1175/MWR-D-19-0372.1. URL https://journals.ametsoc.org/view/journals/mwre/148/7/mwrD190372.xml.

[11] Space Weather Prediction Center and National Oceanic And Atmospheric Administration. Forecast Verification Glossary. URL https://www.swpc.noaa.gov/sites/default/files/images/u30/Forecast%20Verification%20Glossary.pdf.

[12] Harold E. Brooks and James Correia. Long-term performance metrics for National Weather Service Tornado warnings. *Weather and Forecasting*, 33(6):1501–1511, 12 2018. ISSN 15200434. doi: 10.1175/WAF-D-18-0120.1. URL https://verification.nws.noaa.gov/services/public/.

[13] Dinithi Nallaperuma, Rashmika Nawaratne, Tharindu Bandaragoda, Achini Adikari, Su Nguyen, Thimal Kempitiya, Daswin De Silva, Damminda Alahakoon, and Dakshan Pothuhera. Online Incremental Machine Learning Platform for Big Data-Driven Smart Traffic Management. *IEEE Transactions on Intelligent Transportation Systems*, 20(12):4679–4690, 12 2019. ISSN 15580016. doi: 10.1109/TITS.2019.2924883. URL https://ieeexplore.ieee.org/abstract/document/8759919.

[14] Ibai Lana, Javier Del Ser, Manuel Velez, and Eleni I. Vlahogianni. Road Traffic Forecasting: Recent Advances and New Challenges. *IEEE Intelligent Transportation Systems Magazine*, 10(2):93–109, 6 2018. ISSN 19411197. doi: 10.1109/MITS.2018.2806634. URL https://ieeexplore.ieee.org/document/8344781.

[15] Eleni I. Vlahogianni, Matthew G. Karlaftis, and John C. Golias. Short-term traffic forecasting: Where we are and where we're going. *Transportation Research Part C: Emerging Technologies*, 43:3–19, 6 2014. ISSN 0968090X. doi: 10.1016/j.trc.2014.01.005. URL https://www.sciencedirect.com/science/article/pii/S0968090X14000096.

[16] João Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys*, 46(4), 3 2014. ISSN 15577341. doi: 10.1145/2523813. URL http://dx.doi.org/10.1145/2523813.

[17] Deeplearning.ai. Companies Slipping on AI Goals, Self Training for Better Vision, Muppets and Models, China Vs US?, Only the Best Examples, Proliferating Patents, 12 2019. URL https://info.deeplearning.ai/the-batch-companies-slipping-on-ai-goals-self-training-for-better-vision-muppets-and-models-china-vs-us-only-the-best-examples-proliferating-patents.

[18] Inc. Gartner. Gartner Identifies the Top Strategic Technology Trends for 2021. Technical report, Gartner, 2020. URL https://www.gartner.com/en/newsroom/press-releases/2020-10-19-gartner-identifies-the-top-strategic-technology-trends-for-2021.

[19] Hewlett Packard Enterprise. HPE Ezmeral Software Platform for Digital Transformation | HPE. URL https://www.hpe.com/us/en/ezmeral.html.

[20] Curt Hopkins. Operationalizing machine learning: The future of practical AI, 4 2020. URL https://www.hpe.com/us/en/insights/articles/operationalizing-machine-learning--the-future-of-practical-ai-2004.html.

[21] Cristiano Breuel. ML Ops: Machine Learning as an Engineering Discipline. 1 2020. URL https://towardsdatascience.com/ml-ops-machine-learning-as-an-engineering-discipline-b86ca4874a3f.

[22] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. Challenges in Deploying Machine Learning: a Survey of Case Studies. *The ML-Retrospectives, Surveys & Meta-Analyses Workshop, NeurIPS 2020*, 11 2020. URL http://arxiv.org/abs/2011.09926.

[23] Favio Vazquez. Deep Learning made easy with Deep Cognition. *Becoming Human: Artificial Intelligence Magazine*, 2017. URL https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351.

[24] Satya Ganesh. Weights and Bias in a Neural Network. *Towards Data Science*, 2020. URL https://towardsdatascience.com/whats-the-role-of-weights-and-bias-in-a-neural-network-4cf7e9888a0f.

[25] Assaad Moawad. Neural networks and back-propagation explained in a simple way. *Medium - DataThings*, 2018. URL https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e.

[26] Scott Fortmann-Roe. Understanding the Bias-Variance Tradeoff, 2012. URL http://scott.fortmann-roe.com/docs/BiasVariance.html.

[27] Seema Singh. Understanding the Bias-Variance Tradeoff, 2018. URL https://towardsdatascience.com/understanding-the-bias-variance-tradeoff-165e6942b229.

[28] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256, 2010. URL http://www.iro.umontreal.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[30] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 12 2015. URL https://arxiv.org/abs/1412.6980v9.

[31] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456. PMLR, 6 2015. URL http://proceedings.mlr.press/v37/ioffe15.html.

[32] James Somers. The Pastry A.I. That Learned to Fight Cancer, 2021. URL https://www.newyorker.com/tech/annals-of-technology/the-pastry-ai-that-learned-to-fight-cancer.

[33] Benny Prijono. Student Notes: Convolutional Neural Networks (CNN) Introduction – Belajar Pembelajaran Mesin Indonesia, 2018. URL `https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/`.

[34] datahacker.rs. Deep Learning | Master Data Science, 2021. URL `http://datahacker.rs/deep-learning/`.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-December, pages 770–778. IEEE Computer Society, 12 2016. ISBN 9781467388504. doi: 10.1109/CVPR.2016.90. URL `http://image-net.org/challenges/LSVRC/2015/`.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9. CVPR, 2015.

[37] We Need To Go Deeper | Know Your Meme, 2010. URL `https://knowyourmeme.com/memes/we-need-to-go-deeper`.

[38] Bharath Raj. A Simple Guide to the Versions of the Inception Network , 2018. URL `https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202`.

[39] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997. ISSN 08997667. doi: 10.1162/neco.1997.9.8.1735. URL `http://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf`.

[40] Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. ISSN 1053587X. doi: 10.1109/78.650093.

[41] Xiaoguang Mei, Erting Pan, Yong Ma, Xiaobing Dai, Jun Huang, Fan Fan, Qinglei Du, Hong Zheng, and Jiayi Ma. Spectral-Spatial Attention Networks for Hyperspectral Image Classification. *Remote Sensing*, 11:963, 5 2019. doi: 10.3390/rs11080963.

[42] Cem Kaner. A Course in Black Box Software Testing, 2004. URL `http://www.testingeducation.org/k04/OracleExamples.htm`.

[43] Daniel Selsam, Percy Liang, and David L Dill. Developing Bug-Free Machine Learning Systems With Formal Mathematics. In *Proceedings of the 34th International Conference on Machine Learning*, pages 3047–3056. PMLR, 7 2017. URL `https://github.com/Theano/Theano/issues/4770`.

[44] Goku Mohandas. MLOps - Made With ML, 2021. URL `https://madewithml.com/courses/mlops/`.

[45] Jeremy Jordan. Effective testing for machine learning systems., 8 2020. URL `https://www.jeremyjordan.me/testing-ml/`.

[46] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912. Association for Computational Linguistics, 2020. URL `https://github.com/marcotcr/checklist`.

[47] Yulia Gavrilova. Machine Learning Testing: A Step to Perfection, 2020. URL `https://serokell.io/blog/machine-learning-testing`.

[48] Clement Delangue and Julien Chamound. Hugging Face – The AI community building the future. URL `https://huggingface.co/`.

[49] Google LLC. Google AI. URL `https://ai.google/`.

[50] Stanford University. Stanford University. URL `https://www.stanford.edu/`.

[51] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox, 9 2019. URL `https://ludwig-ai.github.io/ludwig-docs/`.

[52] Uber Technologies Inc. Uber AI. URL `https://www.uber.com/us/en/uberai/`.

[53] Gideon Mendels. Live Panel: How do top researchers from Google, Stanford and Hugging Face approach new ML problems?, 2020. URL `https://vimeo.com/user124635767`.

[54] Christian Murphy and Gail Kaiser. Improving the Dependability of Machine Learning Applications. Technical report, Department of Computer Science, Columbia University, 2011. URL `https://academiccommons.columbia.edu/doi/10.7916/D8P2761H`.

[55] Elaine J Weyuker. On Testing Non-testable Programs. *The Computer Journal*, 25 (4):465–470, 1982. doi: https://doi.org/10.1093/comjnl/25.4.465. URL `https://academic.oup.com/comjnl/article/25/4/465/366384`.

[56] Joe W. Duran and Simeon C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, 1984. ISSN 00985589. doi: 10.1109/TSE.1984.5010257.

[57] Richard Hamlet. Random Testing. In *Encyclopedia of Software Engineering*, pages 970–978. John Wiley & Sons, Inc., 1 2002. doi: 10.1002/0471028959.sof268. URL `https://onlinelibrary.wiley.com/doi/full/10.1002/0471028959.sof268https://onlinelibrary.wiley.com/doi/abs/10.1002/0471028959.sof268https://onlinelibrary.wiley.com/doi/10.1002/0471028959.sof268`.

[58] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical report, Department of Computer Science, The Hon Kong University of Science and Technology, 1998. URL `http://arxiv.org/abs/2002.12543`.

[59] Zhi Quan Zhou, D H Huang, T H Tse, Zongyuan Yang, Haitao Huang, and T Y Chen. Metamorphic Testing and Its Applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST)*. Software Engineers Association, Japan, 2004.

[60] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Testing Deep Neural Networks. *arXiv*, 3 2018. URL `http://arxiv.org/abs/1803.04792`.

[61] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A Practical Tutorial on Modified Condition/ Decision Coverage. Technical report, NASA Scientific and Technical Information Program Office, 2001. URL `http://www.sti.nasa.gov`.

[62] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding Neural Networks Through Deep Visualization. In *Deep Learning Workshop, 31st International Conference on Machine Learning*, 6 2015. URL http://arxiv.org/abs/1506.06579.

[63] Augustus Odena, Catherine Olsson, David G Andersen, and Ian Goodfellow. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019. URL http://proceedings.mlr.press/v97/odena19a.html.

[64] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 109–119, New York, NY, USA, 2018. ACM. ISBN 9781450359375. URL https://doi.org/10.1145/3238147.3238172.

[65] Pei Kexin, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems CCS CONCEPTS. In *Proceedings of the 26th Symposium on Operating Systems Principles*, volume 17, pages 1–18, New York, NY, USA, 2017. ACM. ISBN 9781450350853. URL https://doi.org/10.1145/3132747.3132785.

[66] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, Yadong Wang, and Chun-Yang Chen. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, volume 18, pages 120–131, New York, NY, USA, 2018. ACM. ISBN 9781450359375. URL https://doi.org/10.1145/3238147.3238202.

[67] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10805 LNCS, pages 408–426. Springer Verlag, 2018. ISBN 9783319899596. doi: 10.1007/978-3-319-89960-2{_}22. URL https://doi.org/10.1007/978-3-319-89960-2_22.

[68] The Coq Development Team. The Coq Proof Assistant Reference Manual: Version 8.5. Technical report, INRIA, 2016. URL http://www.opencontent.org/openpub.

[69] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9195, pages 378–388. Springer Verlag, 2015. doi: 10.1007/978-3-319-21401-6{_}26. URL http://leanprover.github.io.

[70] M.J. Gordon, A.J. Milner, and CP Wadsworth. Edinburgh lcf: a mechanised logic of computation. *Lecture notes computer science*, 78, 1979.

[71] Michael J. C. Gordon and T. F. Melham. Introduction to Hol a Theorem Proving Environment for Higher Order Logic, 1993. URL https://philpapers.org/rec/GORITH.

[72] John Harrison. HOL light: A tutorial introduction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1166, pages 265–269. Springer Verlag, 1996. ISBN

3540619372. doi: 10.1007/BFb0031814. URL https://link.springer.com/chapter/10.1007/BFb0031814.

[73] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer, 2002. URL https://books.google.no/books?hl=en&lr=&id=R6ul20M6nTIC&oi=fnd&pg=PR2&dq=Isabelle/HOL:++a++proof++assistant++for++higher-order+logic&ots=ex4SpHJNPg&sig=18lDo7zU4Ll64erSEH-4dEbINOw&redir_esc=y#v=onepage&q=Isabelle%2FHOL%3A%20%20a%20%20proof%20%20assistant%20%20for%20%20higher-order%20logic&f=false.

[74] S Owre, J M Rushby, and N Shankar. PVS: A Prototype Verification System. In *Automated Deduction CADE-11*, pages 748–752. Springer, 1992. doi: https://doi.org/10.1007/3-540-55602-8{_}217.

[75] Dan Merron. Deployment Pipelines (CI/CD) in Software Engineering – BMC Software | Blogs, 5 2020. URL https://www.bmc.com/blogs/deployment-pipeline/.

[76] Algorithmia. The ML pipeline and why it's important | Algorithmia Blog, 9 2020. URL https://algorithmia.com/blog/ml-pipeline.

[77] Sten Pittet. Continuous integration vs. continuous delivery vs. continuous deployment. URL https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

[78] David Sweenor, Steven Hillion, Dan Rope, Dev Kannabiran, and Thomas Hill. *ML Ops: Operationalizing Data Science*. O'Reilly Media Inc., Sebastapol, CA, first edition edition, 2020. ISBN 978-1-492-07463-2.

[79] Anthony Mullen, Saniye Aleybi, Van Baker, Arun Chandrasekaran, Alexander Linden, Magnus Revang, and Svetlana Sicular. Predicts 2020: Artificial Intelligence — the Road to Production, 12 2019. URL https://www.gartner.com/en/documents/3975770/predicts-2020-artificial-intelligence-the-road-to-produc.

[80] Kyle Wiggers. Algorithmia: 50% of companies spend between 8 and 90 days deploying a single AI model | VentureBeat. *VentureBeat: The Machine*, 12 2019. URL https://venturebeat.com/2019/12/11/algorithmia-50-of-companies-spend-upwards-of-three-months-deploying-a-single-ai-model/.

[81] Lawrence E Hecht. Add It Up: How Long Does a Machine Learning Deployment Take? – The New Stack. *TheNewStack*, 12 2019. URL https://thenewstack.io/add-it-up-how-long-does-a-machine-learning-deployment-take/.

[82] Yingnong Dang, Qingwei Lin, and Peng Huang. AIOps: Real-World Challenges and Research Innovations. In *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019*, pages 4–5. Institute of Electrical and Electronics Engineers Inc., 2019. ISBN 9781728117645. doi: 10.1109/ICSE-Companion.2019.00023. URL https://www.moogsoft.com/resources/aiops/guide/everything-aiops/.

[83] Danilo Sato, Arif Wider, and Christoph Windheuser. Continuous Delivery for Machine Learning. *Martin Fowler*, 9 2019. URL https://martinfowler.com/articles/cd4ml.html.

[84] Margaret Rouse. *Sandbox Definition*. TechTarget. URL http://searchsecurity.techtarget.com/definition/sandbox.

[85] Sandbox (software development) - Wikipedia. URL https://en.wikipedia.org/wiki/Sandbox_(software_development).

[86] Sasu Mäkinen, Henrik Skogström, Eero Laaksonen, and Tommi Mikkonen. Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help? *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN) of 43rd International Conference on Software Engineering (ICSE)*, 3 2021. URL http://arxiv.org/abs/2103.08942.

[87] Sumeet Agrawal and Anant Mittal. Automate and Productize Machine Learning Algorithms MLOps: 5 Steps to Operationalize Machine Learning Models White Paper. Technical report, Informatica, 2020.

[88] Bilge Celik and Joaquin Vanschoren. Adaptation Strategies for Automated Machine Learning on Evolving Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. ISSN 19393539. doi: 10.1109/TPAMI.2021.3062900. URL https://ieeexplore.ieee.org/abstract/document/9366792.

[89] Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212, 1 2021. ISSN 09507051. doi: 10.1016/j.knosys.2020.106622.

[90] Analyttica Datalab. Gini Coefficient or Gini Index in our Data Science & Analytics platform, 2018. URL https://medium.com/@analyttica/gini-coefficient-or-gini-index-in-our-data-science-analytics-platform-d0408fc83772.

[91] run:ai. Machine Learning Operations, 2021. URL https://www.run.ai/guides/machine-learning-operations/.

[92] Pranjal Pandey. Data Preprocessing : Concepts. Introduction to the concepts of Data... | by Pranjal Pandey | Towards Data Science. *Towards Data Science*, 2019. URL https://towardsdatascience.com/data-preprocessing-concepts-fa946d11c825.

[93] Dennis Li, Euxhen Hasanaj, and Shuo Li. Baselines. *Machine Learning Blog , MLD, Carnegie Mellon University*, 2020. URL https://blog.ml.cmu.edu/2020/08/31/3-baselines/.

[94] Amit Ganatra, Gaurang Panchal, Devyani Panchal, and Y.P. Kosta. Searching Most Efficient Neural Network Architecture Using Akaike's Information Criterion (AIC). *Article in International Journal of Computer Applications*, 1(5):975–8887, 2010. doi: 10.5120/126-242. URL https://www.researchgate.net/publication/43656073.

[95] Zhiye Zhao, Yun Zhang, and Hongjian Liao. Design of ensemble neural network using the Akaike information criterion. *Engineering Applications of Artificial Intelligence*, 21(8):1182–1188, 12 2008. ISSN 09521976. doi: 10.1016/j.engappai.2008.02.007.

[96] Min Qi and Guoqiang Peter Zhang. An investigation of model selection criteria for neural network time series forecasting. *European Journal of Operational Research*, 132(3):666–680, 8 2001. ISSN 03772217. doi: 10.1016/S0377-2217(00)00171-5.

[97] Alboukadel Kassambara. Regression Model Accuracy Metrics: R-square, AIC, BIC, Cp and more . In *Machine Learning Essential: Practical Guide in R*. STHDA, 1 edition, 2018. ISBN 978-1986406857. URL http://www.sthda.com/english/articles/38-regression-model-validation/158-regression-model-accuracy-metrics-r-square-aic-bic-cp-and-more/.

[98] Hirotugu Akaike. A New Look at the Statistical Model Identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974. ISSN 15582523. doi: 10.1109/TAC.1974.1100705.

[99] Sachin Date. The Akaike Information Criterion. Learn about the AIC and how to use it | by Sachin Date | Towards Data Science, 2019. URL `https://towardsdatascience.com/the-akaike-information-criterion-c20c8fd832f2`.

[100] Rebecca Bevans. Akaike Information Criterion | When & How to Use It, 2020. URL `https://www.scribbr.com/statistics/akaike-information-criterion/`.

[101] Gideon Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 6 (2):461 – 464, 1978. doi: 10.1214/aos/1176344136. URL `https://doi.org/10.1214/aos/1176344136`.

[102] Matthias Feurer and Frank Hutter. Hyperparameter Optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019. ISBN 3-030-05317-2. doi: 10.1007/978-3-030-05318-5{_}1. URL `https://doi.org/10.1007/978-3-030-05318-5_1`.

[103] James Bergstra and Joshua Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 13:281–305, 2012. URL `https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a`.

[104] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An Efficient Approach for Assessing Hyperparameter Importance. In *Proceedings of the 31st International Conference on Machine Learning, PMLR*, pages 754–762, 2014.

[105] Nidhal El-Omari. What is the meaning of "black box optimization"?, 5 2020. URL `https://www.researchgate.net/post/What_is_the_meaning_of_black_box_optimization/5fe512ed8a8a836a7c486ea0/citation/download`.

[106] Charles Audet and Warren Hare. Introduction: Tools and Challenges in Derivative-Free and Blackbox Optimization. In *Springer Series in Operations Research and Financial Engineering*, pages 3–14. Springer Nature, 2017. doi: 10.1007/978-3-319-68913-5{_}1. URL `https://doi.org/10.1007/978-3-319-68913-5`.

[107] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 11 2020. ISSN 18728286. doi: 10.1016/j.neucom.2020.07.061. URL `https://www.sciencedirect.com/science/article/pii/S0925231220311693`.

[108] Philipp Probst, Bernd Bischl, and Anne-Laure Boulesteix. Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *Journal of Machine Learning Research*, 20, 2 2018. URL `http://arxiv.org/abs/1802.09596`.

[109] Jeremy Jordan. Evaluating a machine learning model., 2017. URL `https://www.jeremyjordan.me/evaluating-a-machine-learning-model/`.

[110] Kartik Nighania. Various ways to evaluate a machine learning model's performance | by Kartik Nighania | Towards Data Science, 2018. URL `https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15`.

[111] Jason Brownlee. ROC Curves and Precision-Recall Curves for Imbalanced Classification, 2020. URL `https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-imbalanced-classification/`.

[112] Nicolas Vandeput. Forecast KPIs: RMSE, MAE, MAPE & Bias, 2019. URL https://towardsdatascience.com/forecast-kpi-rmse-mae-mape-bias-cdc5703d242d.

[113] Andrew Zhai, Hao-Yu Wu, Eric Tzeng, Dong Huk Park, and Charles Rosenberg. Learning a Unified Embedding for Visual Search at Pinterest. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, pages 2412–2420, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362016. doi: 10.1145/3292500.3330739. URL https://doi.org/10.1145/3292500.3330739.

[114] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software Engineering for Machine Learning: A Case Study. In *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019*, pages 291–300. Institute of Electrical and Electronics Engineers Inc., 5 2019. ISBN 9781728117607. doi: 10.1109/ICSE-SEIP.2019.00042.

[115] Arif Wider and Christian Deger. Getting Smart: Applying Continuous Delivery to Data Science to Drive Car Sales , 2017. URL https://www.thoughtworks.com/insights/blog/getting-smart-applying-continuous-delivery-data-science-drive-car-sales.

[116] Janis Klaise, Arnaud Van Looveren, Clive Cox, Giovanni Vacanti, and Alexandru Coca. Monitoring and explainability of models in production. *arXiv*, 7 2020. URL http://arxiv.org/abs/2007.06299.

[117] Andre Ye. Real Artificial Intelligence: Understanding Extrapolation vs Generalization. *Towards Data Science*, 2020. URL https://towardsdatascience.com/real-artificial-intelligence-understanding-extrapolation-vs-generalization-b8e8dcf5fd4b.

[118] Michael McCartney, Matthias Haeringer, and Wolfgang Polifke. Comparison of Machine Learning Algorithms in the Interpolation and Extrapolation of Flame Describing Functions. *Journal of Engineering for Gas Turbines and Power*, 142(6), 6 2020. ISSN 15288919. doi: 10.1115/1.4045516. URL http://asmedigitalcollection.asme.org/gasturbinespower/article-pdf/142/6/061009/6538416/gtp_142_06_061009.pdf.

[119] Subham S Sahoo, Christoph H Lampert, and Georg Martius. Learning Equations for Extrapolation and Control. In *Proceedings of the 35th International Conference on Machine Learning*, pages 4442–4450. PMLR, 7 2018. URL http://proceedings.mlr.press/v80/sahoo18a.html.

[120] Thomas G. Dietterich. Should we expect machine learning to extrapolate?, 2018. URL https://medium.com/@tdietterich/should-we-expect-machine-learning-to-extrapolate-89f1196f63f2.

[121] Randall Munroe. Extrapolating. URL https://xkcd.com/605/.

[122] Tom Diethe, Tom Borchert, Eno Thereska, Borja Balle, and Neil Lawrence. Continual Learning in Practice. *arXiv*, 3 2019. URL http://arxiv.org/abs/1903.05202.

[123] Syed Muslim Jameel, Manzoor Ahmed Hashmani, Hitham Alhussain, Mobashar Rehman, and Malaysia Arif Budiman. A Critical Review on Adverse Effects of

Concept Drift over Machine Learning Classification Models. *(IJACSA) International Journal of Advanced Computer Science and Applications*, 11(1), 2020. URL www.ijacsa.thesai.org.

[124] Andrés R Masegosa, Ana M Martínez, Darío Ramos-López, Helge Langseth, Thomas D Nielsen, and Antonio Salmerón. Analyzing concept drift: A case study in the financial sector. *Intelligent Data Analysis*, 24:665–688, 2020. ISSN 1571-4128. doi: 10.3233/IDA-194515.

[125] Daniel Langenkämper, Robin van Kevelaer, Autun Purser, and Tim W. Nattkemper. Gear-Induced Concept Drift in Marine Images and Its Effect on Deep Learning Classification. *Frontiers in Marine Science*, 7:506, 7 2020. ISSN 22967745. doi: 10.3389/fmars.2020.00506. URL www.frontiersin.org.

[126] Jan Zenisek, Florian Holzinger, and Michael Affenzeller. Machine learning based concept drift detection for predictive maintenance. *Computers and Industrial Engineering*, 137:106031, 11 2019. ISSN 03608352. doi: 10.1016/j.cie.2019.106031.

[127] Klaus Ackermann, Hareem Naveed, Jason Bennett, Joe Walsh, Andrea Navarrete Rivera, Michael Defoe, Adolfo De Unánue, Sun Joo Lee, Crystal Cody, Lauren Haynes, and Rayid Ghani. Deploying machine learning models for public policy: A framework. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, volume 18, pages 15–22. Association for Computing Machinery, 7 2018. ISBN 9781450355520. doi: 10.1145/3219819.3219911. URL https://doi.org/10.1145/3219819.3219911.

[128] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden Technical Debt in Machine Learning Systems. In *Advances in neural information processing systems*, pages 2503–2511, 2015. URL https://web.kaust.edu.sa/Faculty/MarcoCanini/classes/CS290E/F19/papers/tech-debt.pdf.

[129] Docker Inc. Docker Docs, 2021.

[130] Emre Ceylan. Docker in a Nutshell. What is Docker? *Medium*, 2020. URL https://medium.com/dev-jam/docker-in-a-nutshell-f2e315211195.

[131] The Kubernetes Authors. Kubernetes. URL https://kubernetes.io/.

[132] Run:ai. Run:AI Platform - Build and Train Models with Unlimited Compute. URL https://www.run.ai/platform/.

[133] George Anadiotis. Run:AI takes your AI and runs it, on the super-fast software stack of the future. *Big on Data, ZDNet*, 2019. URL https://www.zdnet.com/article/take-your-ai-and-run-it-on-the-super-fast-software-stack-of-the-future/.

[134] The Apache Software Foundation. Apache Kafka, . URL https://kafka.apache.org/.

[135] Google. Dataflow | Google Cloud. URL https://cloud.google.com/dataflow.

[136] Databricks. Databricks. URL https://databricks.com/product/open-source.

[137] Ian Pointer. What is Apache Spark? The big data platform that crushed Hadoop . *InfoWorld*, 2020. URL https://www.infoworld.com/article/3236869/what-is-apache-spark-the-big-data-platform-that-crushed-hadoop.html.

[138] The Apache Software Foundation. Apache Spark - Unified Analytics Engine for Big Data, . URL `https://spark.apache.org/`.

[139] MLflow Projects. MLflow - A platform for the machine learning lifecycle. URL `https://mlflow.org/`.

[140] Thomas Elliott. The State of the Octoverse: machine learning, 2019. URL `https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/`.

[141] Anaconda Software Distribution, 2020. URL `https://docs.anaconda.com/`.

[142] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[143] Charles R Harris, K Jarrod Millman, Stefan J.van der Walt, Ralf Gommers, Pauli Virtanen, Cournapeau David, Eric Wieser, Julian Taylor, Berg Sebastian, Nathaniel J Smith, Robert Kern, Matti Picus Hoyer, Stephan, Marten H van Kerkwijk, Brett Matthew, Allan Haldane, Jaime Fernandez del Rio, Mark Wiebe, Pearu Peterson, Gerard-Marchant Pierre, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 9 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[144] Joel A. E. Andersson, Joris Gillis, Greg Horn, James B. Rawlings, and Moritz Diehl. CasADi - A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 2018. URL `https://web.casadi.org/`.

[145] J D Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[146] J.G. van de Vusse. Plug-flow type reactor versus tank reactor. *Chemical Engineering Science*, 19(12):994–996, 1964. ISSN 0009-2509. doi: https://doi.org/10.1016/0009-2509(64)85109-5. URL `https://www.sciencedirect.com/science/article/pii/0009250964851095`.

[147] Hong Chen, H Kremling, and Frank Allgöwer. Nonlinear Predictive Control of a Benchmark CSTR. *Proceedings of the 3rd European Control Conference, Rome-Italy.*, pages 3247–3252, 4 1995.

[148] Jiri Vojtesek and Petr Dostál. From steady-state and dynamic analysis to adaptive control of the CSTR reactor. *Simulation in Wider Europe - 19th European Conference on Modelling and Simulation, ECMS 2005*, 4 2005.

[149] Jiri Vojtesek, Petr Dostal, and Vladimir Bobal. Control of nonlinear system - Adaptive and predictive control. In *IFAC Proceedings Volumes (IFAC-PapersOnline)*, volume 7, pages 898–903. IFAC Secretariat, 2009. ISBN 9783902661548. doi: 10.3182/20090712-4-tr-2008.00147.

[150] Jiri Vojtesek and Petr Dostal. Adaptive Control of Chemical Reactor. In *International Conference Cybernetics and Informatics*, 2010.

[151] Iman Hajizadeh and Seyed Mohsen Hosseini. Artificial Neural Networks Based Control of CSTRs with Van de Vusse Reaction. *Intelligent Automation and Soft Computing*, 2014. URL https://www.researchgate.net/publication/273754874.

[152] Eric M L Luz and Brunno F Santos. Development of Intelligent Models for the Prediction the Dynamics of Nonlinear Process. *Chemical Engineering Transactions*, 74:763–768, 2019. doi: DOI:10.3303/CET1974128.

[153] S. Engell and K. U. Klatt. Nonlinear control of a non-minimum-phase CSTR. In *American Control Conference*, pages 2941–2945. IEEE, 1993. ISBN 0780308611. doi: 10.23919/acc.1993.4793439. URL https://ieeexplore.ieee.org/abstract/document/4793439/.

[154] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[155] Bjarne Grimstad and NTNU. TTK28 Modeling with neural networks - Institutt for teknisk kybernetikk, NTNU, 2020. URL https://www.itk.ntnu.no/emner/fordypning/TTK28.

[156] Neil D. Lawrence. Modern Data Oriented Programming. In *Advances and Challenges in Machine Learning Languages, Centre for Mathematical Sciences, Cambridge*, 2019. URL http://inverseprobability.com/talks/notes/modern-data-oriented-programming.html.

[157] Tom Borchert. Milan: An Evolution of Data-Oriented Programming, 2020. URL https://tborchertblog.wordpress.com/2020/02/13/28/.

# Appendix

## A Program files for miniature machine learning system

### README.md

README.md file for miniature machine learning system, referred to in chapter 7. Explains the structure of the system.

# Miniature Machine Learning System

This system provides a configurable neural network class, and a simulator for a CSTR with a van de Vusse reaction, which can generate data to be used for training, and testing the neural network.

The neural network can be applied to a different process, by configuring its input features and outputs. Data must then be fetched from a different source, or the van de Vusse-simulator can be used as a template for developing your own simulator for a different process.

## Info

The system is build with Python, uses Conda for packet management, and the PyTorch machine learning framework. The CasADi framework is used to develop the simulator.

The system as it is provides basic functionality for performing different tests of various ML techniques or technologies. The system would massively benefit from further development to create more extensive functionlity.

## Contents

The system consist of:

- config.json, in which desired configurations for the system are performed.
- neural_net_class.py - implements the class for the neural net.
- train_model.py - splits the data set, trains the model, and subjects the trained model to the validation set.
- test_model.py - also splits data set, as the split data sets aren't stored. Tests model, and visualizes reults.
- make_predictions_batch.py - performs predicitons on batches of data. Pretty much similar functionality to test_model.py, other then performing on a completely separate data set. Provides a frame for if real production data would be made available at a later iteration.
- generate_input_vandevusse.py - generates input data for the simulator in a selection of different ways.
- simulate_vandevusee.py - performs the simulation of the CSTR with van de Vusse reaction subject to the generated input data. Output data is stored, so as to be used by neural network.
- steady_state_extraction.py - extracts only data points from intervals where a specified feature is in steady state.
- plotting.py - contains different plotting functions that help streamline the visualization process.

## Configurability

The system is created so that any necessary configurations can be performed in config.json, without requiring much knowledge about the code base. The options are as follows:

### Neural net

- "further_train_existing_net": true/false - decides whether to perform training on an already existing net, or to initialize a new one.
- "prev_net_file": "path/to/file" - the path to the net that should be further trained if above option is set to true.
- "trained_net_storage_file": "path/to/file" - where to store the trained net. Also the net that is loaded when testing the model.
- "data_set": "path/to/file" - path to the data set that should be trained, validated, and tested on.
- "input_cols": ["feature1", "feature2", ....] - the features that make up the input of the neural net.
- "output_cols": ["output1", ...] - the output feature of the neural net
- "all_output_cols": ["all possible output features"] - not used, but kept for the purpose of keeping track of possible outputs.
- "hidden_layers": [size_l1,size_l2, ....] - the number of hidden layers in the neural net and their sizes.
- "training_batch_size": x - training batch size
- "shuffle_training_data": true/false - whether or not to shuffle traiing data prior to training.
- "n_epochs": n - number of epochs for the training procedure.
- "learning_rate": x - learning rate in training procedure.
- "l2_regularization": x - regularization coefficient in training procedure.

### Vdv Model - The process in the simulator

- "load_initial_state_values_from_file": true/false - whether or not to load initial values from a file.
- "initial_state_values_file": "path/to/file" - file for initial values if above is true. Also where the last last state values in the simulation are stored. In case one wants to "continue" a simulation.
- "initial_state_values": [x1, x2, x3, ...] - initial values for the states in the process, if not loaded from file.
- "simulation_result_dataset_storage_file":"path/to/file" - where to store the simulation results.
- "n_iterations": n - number of iterations to simulate for
- "simulation":
    - "samples_per_hour": t - how many samples per hour in the simulation process. dt = 1/"samples_per_hour". Decides the step size of the integrator in the simulator.

### Input generation

- "options":
    - "input_interval_size": n, how many iterations the input should stay constant for before changing,
    - "perturbation_style": "single"/"double"/"F_only"/"Qk_only"- decides how the perturbation should ensue. "single" perturbates one of the inputs, chosen randomly, at the end of each interval. "double" changes both inputs. "F_only" changes only the input feed. "Qk_only" changes only the jacket cooling rate.
    - "F_min":fl - minimum value for input feed
    - "F_max":fu - maximum value for input feed
    - "Fin_0":f0 - initial value for input feed
    - "max_change_Fin": df - maximum change in input feed at each interval.
    - "Qk_min":ql - minimum value for jacket cooling rate
    - "Qk_max":qu - maximum value for jacket cooling rate

    - "Qk_0":q0 - initial value for jacket cooling rate
    - "max_change_Qk": dq - maximum change in jacket cooling rate at each interval
- "storage_file": "path/to/file" - where to store the generated input sequence

### Data extraction

- "steady_state_variable": "feature1" - which feature should be in a steady state in the extracted data points.
- "state_change_threshold": x - the threshold change in feature value allowed in steady state
- "steady_period_duration_threshold": t - how many iterations must the feature be in steady state to make up a steady state period.
- "raw_data_file": "path/to/file" - file to fetch data to perform extraction on.
- "steady_state_data_storage_file": "path/to/file" - storage for extracted data.

### Predictions

- "prediction_net_file": "path/to/file" - net to use for predictions.
- "prediction_input_data": "path/to/file" - file to data to predict on.

## Pipelines

### Training pipeline

Invoke the modules in the following order:
generate_input_vandevusse.py -> simulate_vandevusse.py -> steady_state_extraction.py -> train_model.py -> test_model.py

### Prediciton pipeline

Invoke the modules in the following order:
generate_input_vandevusse.py -> simulate_vandevusse.py -> make_predictions_batch.py

## Further work

The system consists of several components, which forms ML pipelines. More components could be implemented into the system, such as hyperparameter optimization, more extensive data pre-processing, etc. The system is modular, and integrating extra components should provide few problems.

The steady_state_extraction.py module could also benefit from more extensive functionality, e.g. by performing analyses of the dynamics of the simulation process to implement more helpful steady state extraction. This could include investigating the time constant, allowing for extracting data based on the steady state of the input, which are the features we know are always measurable.

It would be interesting to integrate other technologies into this system, such as Docker, in order to create a containerized system that could be deployed somewhere. Kafka is also an option in order to enable the functionality of an event-driven system.

Automating the pipelines would also be auspicious, and a step in the right direction of creating a system that could be subjected to a proper form of MLOps.

### Configuration

Configuration file for miniature machine learning system, referred to in chapter 7.


```
{
    "neural_net":
    {
        "further_train_existing_net": false,
        "prev_net_file": "neural-nets/trained_nn",
        "trained_net_storage_file": "neural-nets/trained_nn_wo_temps",
        "data_set": "data/vdv_steady_state_dataset.csv",
        "input_cols": ["CAin", "CBin", "CCin", "CDin","Fin", "Qk"],
        "output_cols": ["Tout"],
        "all_output_cols": ["CAout", "CBout", "CCout", "CDout", "Tout",
        ↪  "Tkout"],
        "hidden_layers": [50,50],
        "training_batch_size": 10,
        "shuffle_training_data": true,
        "n_epochs":100,
        "learning_rate":0.001,
        "l2_regularization":0.01
    },
    "vdv_model":
    {
        "initial_state_values": [2.2291, 1.0417, 0.91397, 0.91520, 79.591,
        ↪  77.69],
        "initial_state_values_file": "data/vdv_initial_states.csv",
        "simulation_result_dataset_storage_file":"data/vdv_dataset.csv",
        "load_initial_state_values_from_file": false,
        "n_iterations": 288000,
        "simulation":
        {
            "samples_per_hour": 3600
        },
        "input_generation":
        {   "options":
            {

                "input_interval_size": 14400,
                "perturbation_style": "single",
                "F_min":10,
                "F_max":150,
                "Fin_0":14.9,
                "max_change_Fin": 5,
                "Qk_min":-8500,
                "Qk_max":-500,
                "Qk_0":-1113.5,
                "max_change_Qk": 25
            },
            "storage_file": "data/vdv_input_sequence.csv"
        },
```

```
46          "data_extraction":
47          {
48              "steady_state_variable": "Tout",
49              "state_change_threshold": 0.005,
50              "steady_period_duration_threshold": 120,
51              "raw_data_file": "data/vdv_dataset.csv",
52              "steady_state_data_storage_file":
                ↪  "data/vdv_steady_state_dataset.csv"
53          }
54      },
55      "predictions":
56      {
57          "prediction_net_file": "neural-nets/trained_nn_wo_temps",
58          "prediction_input_data": "data/prod/vdv_dataset.csv"
59      }
60
61  }
```

### Simulator

Simulator for CSTR with van de Vusse reaction for miniature machine learning system, referred to in chapter 7.

```
1   from casadi import *
2   import numpy as np
3   import matplotlib.pyplot as plt
4   import pandas as pd
5   import json
6
7   # local imports
8   from utils import plotting
9
10  # Fetch configuration
11  with open ('config.json') as config_file:
12      config = json.load(config_file)
13  #print(config)
14  try:
15      input_sequence_file =
        ↪  config['vdv_model']['input_generation']['storage_file']
16      initial_state_values_file =
        ↪  config['vdv_model']['initial_state_values_file']
17  except:
18      input_sequence_file = "data/vdv_input_sequence.csv"
19      initial_state_values_file  = "data/vdv_initial_states.csv"
20
21  output_result_file = "data/vdv_simulation_results.csv"
22  dataset_file = "data/vdv_dataset.csv"
23  prod_batch_file = "data/prod/vdv_dataset.csv"
24
25  # Hard coded values
```

```
26  #n = 14400 # number of iterations # decided by input sequence
27  #TF = n*dt*60 # Time horizon
28  #dt = 1/3600 # step size # Sample once per second
29
30  # Fetched from config.json
31
32  n = config['vdv_model']['n_iterations']
33  dt = 1/config['vdv_model']['simulation']['samples_per_hour']
34
35  # model parameters
36  params = {
37      "CA0":  5.1,
38      "T0":104.9,
39     "k10": 1.287e12,
40      "k20": 1.287e12,
41      "k30": 9.043e9,
42      "E1": 9758.3,
43      "E2": 9758.3,
44      "E3": 8560,
45      "dHr1": 4.2,
46      "dHr2": -11,
47      "dHr3": -41.85,
48      "rho": 0.9342,
49      "Cp": 3.01,
50      "kw": 4032,
51      "AR": 0.215,
52      "VR": 10.0,
53      "mK": 5,
54      "CpK": 2
55  }
56
57
58  nx = 6 # number of states in plant
59  nu = 2 # number of input control variables
60
61  x = SX.sym('x', nx)  # Concentration of A, B, C, D, temperature in CSTR, T
    ↪  and temp of cooling jacket T_k
62  p = SX.sym('p', nu)  # Feed rate F_in and Jacket cooling rate, Qk
63
64  # reaction rates for A->B, B->C and 2A -> D
65  k1 = params["k10"]*np.exp(-params["E1"]/(x[4] + 273.15))
66  k2 = params["k20"]*np.exp(-params["E2"]/(x[4] + 273.15))
67  k3 = params["k30"]*np.exp(-params["E3"]/(x[4] + 273.15))
68
69  r1 = k1*params["VR"]*x[0]
70  r2 = k2*params["VR"]*x[1]
71  r3 = k3*params["VR"]*(x[0]**2)
72
73
74  # ODEs
75  # concentrations
```

```
76  dCA = (-r1 - 2*r3 + p[0]*(params["CA0"]-x[0]))/params["VR"]
77  dCB = (r1 - r2 - p[0]*x[1])/params["VR"]
78  dCC = (r2 - p[0]*x[2])/params["VR"]
79  dCD = (r3 - p[0]*x[3])/params["VR"]
80
81  # temperatures
82  dT = (p[0]*(params["T0"]-x[4])/params["VR"] +
    ↪  (params["kw"]*params["AR"]*(x[5]-x[4])
83  -r1*params["dHr1"] - r2*params["dHr2"] -
    ↪  r3*params["dHr3"])/(params["VR"]*params["rho"]*params["Cp"]))
84  dTk = (p[1] + params["kw"]*params["AR"]*(x[4] -
    ↪  x[5]))/(params["mK"]*params["CpK"])
85
86
87  # Create ODE set and integrator
88  xdot = vertcat(dCA, dCB, dCC, dCD, dT, dTk)
89  ode = {'x':x, 'ode': xdot, 'p': p}
90  opts = {'tf': dt} # Sets correct step size in integrator
91  ode_solver = integrator('F', 'cvodes', ode, opts)
92
93
94  # Function for simulation
95  def simulate_vdv(x0, u):
96      states = [x0]
97      for k in range(u.shape[0]):
98          res = ode_solver(x0=x0, p = u[k])
99          x0 = res["xf"]
100         states.append(x0)
101     return np.concatenate(states, axis = 1)
102
103 ## Beginning of simulation procedure
104
105 # Input sequence for simulation
106 try:
107     df_u = pd.read_csv(input_sequence_file, index_col=0)
108 except:
109     print('Error reading input sequence file. Defaulting to constant u.')
110     # Values for u yielding optimal concentration level of A
111     u = [14.19, -1113.5]
112     u = np.tile(u, (n,1))
113     df_u = pd.DataFrame(data=u, columns = ['Fin', 'Qk'])
114
115 u = df_u.values
116 #print(u)
117
118 # Inital state values
119 if config["vdv_model"]["load_initial_state_values_from_file"]:
120     try:
121         x0 = np.concatenate(pd.read_csv(initial_state_values_file,
            ↪  index_col=0).to_numpy())
122         print("x0 read from file:\n", x0)
```

```
123     except:
124         x0 = config['vdv_model']['initial_state_values']
125         print("Could not load initial values from file, x0 defaults
         ↪  to:\n", x0)
126 else:
127     x0 = config['vdv_model']['initial_state_values']
128     print("x0 initialized to:\n", x0)
129
130
131 x0 = DM(x0)
132
133 states = simulate_vdv(x0, u)
134 TF = dt*u.shape[0]*60 #Time horizon in minutes for visualization
135 plotting.plot_vdv(states, u, TF, source='Casadi Model')
136
137
138 df_states = pd.DataFrame(data=states.T, columns=['CA', 'CB', 'CC', 'CD',
     ↪  'T', 'Tk'])
139 df_last_states = pd.DataFrame(data=states.T[-1].reshape((6,1)).T,
     ↪  columns=['CA0', 'CB0', 'CC0', 'CD0','T0', 'TK0'])
140
141
142 data_set = np.concatenate((states.T[:-1], u, states.T[1:]), axis = 1)
143
144 ## Uncomment to shuffle data set
145 #np.random.shuffle(data_set)
146
147 df_data_set = pd.DataFrame(data=data_set,
148 columns=['CAin', 'CBin', 'CCin', 'CDin', 'Tin', 'Tkin', 'Fin', 'Qk',
     ↪  'CAout', 'CBout', 'CCout', 'CDout', 'Tout','Tkout'])
149
150 # Write results to files
151 try:
152     df_last_states.to_csv(initial_state_values_file) # final state values
153     df_states.to_csv(output_result_file) # time series for all states
154     df_data_set.to_csv(dataset_file) # data set for training ML model
155     #df_data_set.to_csv(prod_batch_file) # same as above, but to seperate
         ↪  file to simulate production setting
156 except:
157     print("Error, could not save simulation results to file")
```

### Neural Net Class

Neural net class for miniature machine learning system, referred to in chapter 7.

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import torch
4 from torch.utils.data import DataLoader
5 from math import sqrt
```

```
6
7
8   class NeuralNet(torch.nn.Module):
9       """
10      PyTorch offers several ways to construct neural networks.
11      Here we choose to implement the network as a Module class.
12      This gives us full control over the construction and clarifies our
    ↪   intentions.
13      """
14
15      def __init__(self, layers):
16          """
17          Constructor of neural network
18          :param layers: list of layer widths. Note that len(layers) =
    ↪   network depth + 1 since we incl. the input layer.
19          """
20          super().__init__()
21
22          self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
23
24          assert len(layers) >= 2, "At least two layers are required (incl.
            ↪   input and output layer)"
25          self.layers = layers
26
27          # Fully connected linear layers
28          linear_layers = []
29
30          for i in range(len(self.layers) - 1):
31              n_in = self.layers[i]
32              n_out = self.layers[i+1]
33              layer = torch.nn.Linear(n_in, n_out)
34
35              # Initialize weights and biases
36              a = 1 if i == 0 else 2
37              layer.weight.data = torch.randn((n_out, n_in)) * sqrt(a / n_in)
38              layer.bias.data = torch.zeros(n_out)
39
40              # Add to list
41              linear_layers.append(layer)
42
43          # Modules/layers must be registered to enable saving of model
44          self.linear_layers = torch.nn.ModuleList(linear_layers)
45
46          # Non-linearity (e.g. ReLU, ELU, or SELU)
47          self.act = torch.nn.ReLU(inplace=False)
48
49      def forward(self, input):
50          """
51          Forward pass to evaluate network for input values
52          :param input: tensor assumed to be of size (batch_size, n_inputs)
53          :return: output tensor
```

```
54          """
55          x = input
56          for l in self.linear_layers[:-1]:
57              x = l(x)
58              x = self.act(x)
59
60          output_layer = self.linear_layers[-1]
61          return output_layer(x)
62
63      def get_num_parameters(self):
64          return sum(p.numel() for p in self.parameters())
65
66      def save(self, path: str):
67          """
68          Save model state
69          :param path: Path to save model state
70          :return: None
71          """
72          torch.save({
73              'model_state_dict': self.state_dict(),
74          }, path)
75
76      def load(self, path: str):
77          """
78          Load model state from file
79          :param path: Path to saved model state
80          :return: None
81          """
82          checkpoint = torch.load(path, map_location=torch.device("cuda" if
            ↪   torch.cuda.is_available() else "cpu"))
83          self.load_state_dict(checkpoint['model_state_dict'])
```

### Training procedure

Training module for miniature machine learning system, referred to in chapter 7.

```
1   import matplotlib.pyplot as plt
2   import pandas as pd
3   import torch
4   from torch.utils.data import DataLoader
5   from math import sqrt
6   import numpy as np
7   from pathlib import Path
8   import json
9
10  # local imports
11  from flow_net_class import FlowNet as Net
12  from utils import plotting
13
14  # We set a fixed seed for repeatability
```

```
15 random_seed = 12345
16 torch.manual_seed(random_seed)
17
18 ## Get path to source directory
19 src_dir = Path(__file__).parent.parent.absolute()
20
21 with open (src_dir / 'config.json') as config_file:
22     config = json.load(config_file)
23
24 # Load dataset
25 #df = pd.read_csv(src_dir / 'data/vdv_steady_state_dataset.csv',
   ↪  index_col=0)
26 try:
27     df = pd.read_csv(src_dir / config["neural_net"]["data_set"],
       ↪  index_col=0)
28 except:
29     print("Unable to load dataset")
30
31 # Split data sets
32 low_index = int(df.shape[0]*0.65)
33 up_index = int(df.shape[0]*0.95)
34 test_set = df.iloc[low_index:up_index]
35
36 # Define the target and features
37 INPUT_COLS = config['neural_net']['input_cols']
38 OUTPUT_COLS = config['neural_net']['output_cols']
39
40 # Make a copy of the dataset and remove the test data
41 train_val_set = df.copy().drop(test_set.index)
42
43 # Sample validation data without replacement (10%)
44 val_set = train_val_set.sample(frac=0.1, replace=False,
   ↪  random_state=random_seed)
45
46 # The remaining data is used for training (90%)
47 train_set = train_val_set.copy().drop(val_set.index)
48
49 # Check that the numbers add up
50 n_points = len(train_set) + len(val_set) + len(test_set)
51 print("Verify:\nLength of dataset = Length of training set + Length of
   ↪  validation set + length of test set")
52 print(f'{len(df)} = {len(train_set)} + {len(val_set)} + {len(test_set)} =
   ↪  {n_points}')
53
54
55
56 def train(
57         net: torch.nn.Module,
58         train_loader: DataLoader,
59         val_loader: DataLoader,
60         n_epochs: int,
```

```
61          lr: float,
62          l2_reg: float,
63  ) -> torch.nn.Module:
64      """
65      Train model using mini-batch SGD
66      After each epoch, we evaluate the model on validation data
67
68      :param net: initialized neural network
69      :param train_loader: DataLoader containing training set
70      :param n_epochs: number of epochs to train
71      :param lr: learning rate (default: 0.001)
72      :param l2_reg: L2 regularization factor (default: 0)
73      :return: torch.nn.Module: trained model.
74      """
75
76      # Define loss and optimizer
77      criterion = torch.nn.MSELoss(reduction='mean')
78      optimizer = torch.optim.Adam(net.parameters(), lr=lr)
79
80      # Train Network
81      for epoch in range(n_epochs):
82          for inputs, labels in train_loader:
83              # Zero the parameter gradients (from last iteration)
84              optimizer.zero_grad()
85
86              # Forward propagation
87              outputs = net(inputs)
88
89              # Compute cost function
90              batch_mse = criterion(outputs, labels)
91
92              reg_loss = 0
93              for param in net.parameters():
94                  reg_loss += param.pow(2).sum()
95
96              cost = batch_mse + l2_reg * reg_loss
97
98              # Backward propagation to compute gradient
99              cost.backward()
100
101             # Update parameters using gradient
102             optimizer.step()
103
104         # Evaluate model on validation data
105         mse_val = 0
106         for inputs, labels in val_loader:
107             mse_val += torch.sum(torch.pow(labels - net(inputs), 2)).item()
108         mse_val /= len(val_loader.dataset)
109         print(f'Epoch: {epoch + 1}: Val MSE: {mse_val}')
110
111     return net
```

```
112
113
114
115
116 # Get input and output tensors and convert them to torch tensors
117 x_train = torch.from_numpy(train_set[INPUT_COLS].values).to(torch.float)
118 y_train = torch.from_numpy(train_set[OUTPUT_COLS].values).to(torch.float)
119
120 x_val = torch.from_numpy(val_set[INPUT_COLS].values).to(torch.float)
121 y_val = torch.from_numpy(val_set[OUTPUT_COLS].values).to(torch.float)
122
123 # Create dataset loaders
124 # Here we specify the batch size and if the data should be shuffled
125 train_dataset = torch.utils.data.TensorDataset(x_train, y_train)
126 train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=10,
    ↪ shuffle=True)
127
128 val_dataset = torch.utils.data.TensorDataset(x_val, y_val)
129 val_loader = torch.utils.data.DataLoader(val_dataset,
    ↪ batch_size=len(val_set), shuffle=False)
130
131
132 # # Construct and initialize the model
133
134 # TODO: load previously trained model instead of creating new if wanted.
135
136 layers = [len(INPUT_COLS), 50, 50, len(OUTPUT_COLS)]
137 net = Net(layers)
138
139 print(f'Size of layers: {layers}')
140 print(f'Number of model parameters: {net.get_num_parameters()}')
141
142
143 # # Train the model
144 # Hyperparameters
145 n_epochs = config['neural_net']['n_epochs']
146 lr = config['neural_net']['learning_rate']
147 l2_reg = config['neural_net']['l2_regularization']
148
149 # Load existing net if specified
150 if config["neural_net"]["further_train_existing_net"]:
151     try:
152         net.load(src_dir / config["neural_net"]["prev_net_file"])
153         print("Loaded existing net from " +
            ↪ config["neural_net"]["prev_net_file"])
154     except:
155         print("Unable to load existing net from " +
            ↪ config["neural_net"]["prev_net_file"])
156
157
158 net = train(net, train_loader, val_loader, n_epochs, lr, l2_reg)
```

```
159 net.save(src_dir / config["neural_net"]["trained_net_storage_file"])
160
161 # # Load model instead of training
162
163 # net.load(src_dir / "neural-nets/trained_nn")
164
165 # # Evaluate the model on validation data
166
167 # Predict on validation data
168 pred_val = net(x_val)
169
170 # Plot prediction
171 dt = 1/config["vdv_model"]["simulation"]["samples_per_hour"]
172 input_sequence_val = val_set[['Fin', 'Qk']].values[1:]
173
174 # Limit to 0:50 to avoid data point overload
175 time = np.linspace(0, (df.shape[0]-1)*60*dt, df.shape[0])
176 time = time[0:50]
177 sources = {'Neural Net': pred_val.detach().numpy()[0:50].T, 'Model':
    ↪ y_val.detach().numpy()[0:50].T}
178 plotting.vdv_plot_states_cmp(sources, time = time, output_cols =
    ↪ OUTPUT_COLS, dt = dt)
179
180 # plt.figure(figsize=(16, 9))
181 # plt.plot(y_val.numpy()[0:100], label='Missing T')
182 # plt.plot(pred_val.detach().numpy()[0:100], label='Estimated T')
183 # plt.legend()
184 # plt.show()
185
186 # Compute MSE, MAE and MAPE on validation data
187 print('Error on validation data')
188
189 mse_val = torch.mean(torch.pow(pred_val - y_val, 2))
190 print(f'MSE: {mse_val.item()}')
191
192 mae_val = torch.mean(torch.abs(pred_val - y_val))
193 print(f'MAE: {mae_val.item()}')
194
195 mape_val = 100*torch.mean(torch.abs(torch.div(pred_val - y_val, y_val)))
196 print(f'MAPE: {mape_val.item()} %')
```

### Testing procedure

Testing module for miniature machine learning system, referred to in chapter 7.

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import torch
4 from torch.utils.data import DataLoader
5 from math import sqrt
```

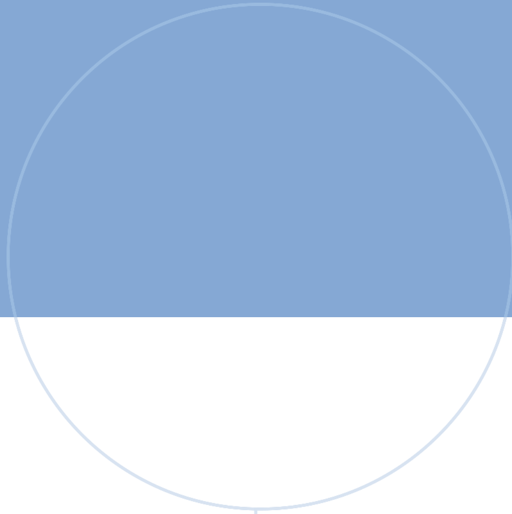
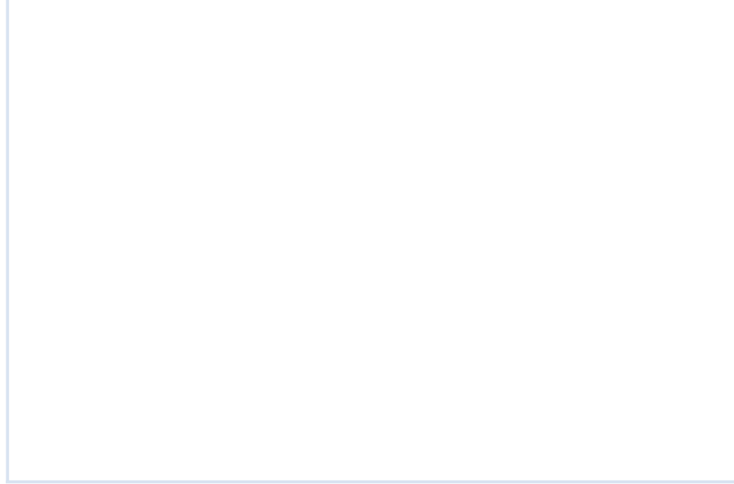
```
6  from pathlib import Path
7  import numpy as np
8  import json
9
10 #local
11 from flow_net_class import FlowNet as Net
12 from utils import plotting
13
14
15
16
17 ## Get path to source directory
18 src_dir = Path(__file__).parent.parent.absolute()
19
20 # TODO: absolute path
21 with open (src_dir / 'config.json') as config_file:
22     config = json.load(config_file)
23
24 dt = 1/config['vdv_model']['simulation']['samples_per_hour']
25
26 #INPUT_COLS = ['CAin', 'CBin', 'CCin', 'CDin', 'Tin', 'Tkin','Fin', 'Qk']
27 #OUTPUT_COLS = ['CAout', 'CBout', 'CCout', 'CDout', 'Tout', 'Tkout']
28 INPUT_COLS = config['neural_net']['input_cols']
29 OUTPUT_COLS = config['neural_net']['output_cols']
30
31 layers = [len(INPUT_COLS), 50, 50, len(OUTPUT_COLS)]
32 net = Net(layers)
33
34 ## Load Net
35 #net.load(src_dir / "neural-nets/trained_nn")
36 net.load(src_dir / config["neural_net"]["trained_net_storage_file"])
37
38 ## Load dataset and define test_set
39 #df = pd.read_csv(src_dir / 'data/vdv_steady_state_dataset.csv',
   ↪  index_col=0)
40 try:
41     df = pd.read_csv(src_dir / config["neural_net"]["data_set"],
       ↪  index_col=0)
42 except:
43     print("Unable to load dataset")
44
45 low_index = int(df.shape[0]*0.65)
46 up_index = int(df.shape[0]*0.95)
47 test_set = df.iloc[low_index:up_index]
48
49 # Fetch time array to plot against
50 df_time = pd.read_csv(src_dir / 'data/vdv_steady_state_time.csv',
   ↪  index_col = 0)
51 time = np.concatenate(df_time.values, axis = 0)
52 time = time[low_index:up_index]
53
```

```
54 # # Evaluate the model on test data
55
56 # Get input and output as torch tensors
57 x_test = torch.from_numpy(test_set[INPUT_COLS].values).to(torch.float)
58 y_test = torch.from_numpy(test_set[OUTPUT_COLS].values).to(torch.float)
59
60 # Make prediction
61 pred_test = net(x_test)
62
63 # Visualization
64 input_sequence_val = test_set[['Fin', 'Qk']].values[:-1]
65
66 #plot_vdv(states=pred_test.detach().numpy().T, u= input_sequence_val,
   ↪ TF=input_sequence_val.shape[0]*dt, source='Neural Net')
67 #plotting.vdv_plot_states(states=pred_test.detach().numpy().T,
   ↪ source='Neural Net')
68 #plotting.vdv_plot_states(states=y_test.detach().numpy().T, source =
   ↪ 'Model')
69
70 # All datapoints
71 #sources = {'Neural Net': pred_test.detach().numpy().T, 'Model':
   ↪ y_test.detach().numpy().T}
72
73 # Only datapoints with steady state
74
75 ss_predictions = pred_test.detach().numpy().T
76 ss_measurements = y_test.detach().numpy().T
77 sources = {'Neural Net': ss_predictions, 'Model': ss_measurements}
78
79 plotting.vdv_plot_states_cmp(sources, time=time, output_cols=OUTPUT_COLS)
80
81 # Separated steady state periods:
82
83 if np.argwhere(np.diff(time)>1.9*60*dt).shape[0] > 0:
84     #print(np.argwhere(np.diff(time)>(60+1)/config['vdv_model']['simulatio
       ↪ n']['samples_per_hour']).shape)
85     periods = np.split(np.arange(0, time.shape[0], 1),
       ↪ np.concatenate(np.argwhere(np.diff(time)>1.9*60*dt)+1))
86     #periods = np.split(np.arange(0, time.shape[0], 1),
       ↪ np.concatenate(np.argwhere(np.diff(time)>0.017)+1))
87
88     print("Number of periods: ", len(periods))
89     plotting.vdv_plot_states_cmp_period_sep(sources, time=time,
       ↪ output_cols=OUTPUT_COLS, periods=periods)
90
91
92 # Compute MSE, MAE and MAPE on test data
93 print('Error on test data')
94
95 mse_test = torch.mean(torch.pow(pred_test - y_test, 2))
96 print(f'MSE: {mse_test.item()}')
```

```
97
98  mae_test = torch.mean(torch.abs(pred_test - y_test))
99  print(f'MAE: {mae_test.item()}')
100
101 mape_test = 100*torch.mean(torch.abs(torch.div(pred_test - y_test,
    ↪  y_test)))
102 print(f'MAPE: {mape_test.item()} %')
```

## B GoogLeNet

Figure 1: GoogLeNet, created and illustrated by Szegedy et al. [36].

NTNU
Norwegian University of
Science and Technology