Decoupling Machine Intelligence from Application in IoT devices

Borys Plyenkov

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 24.2.2019

Supervisor

Prof. Valeriy Vyatkin

Advisor

MSc Edgar Ramos

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Borys Plyenkov

Title Decoupling Machine Intelligence from Application in IoT devices

Degree programme Master's Programme in Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC0007

Supervisor Prof. Valeriy Vyatkin

Advisor MSc Edgar Ramos

Date 24.2.2019 **Number of pages** 66+3 **Language** English

Abstract

Currently, the most prominent model for developing intelligent applications for IoT devices is to have intelligence embedded into the application. This model is characterized by strong coupling between application logic and intelligence implementations in the code of the intelligent application. Alternatively, the intelligence can be taken out of the application and turned into a cloud service that application logic can utilize via standardized Web APIs. This model is characterized by weak coupling between application logic code and intelligence implementation. Strong coupling model makes lifecycle management of intelligence difficult. To update intelligence, usually the whole application must be updated. Cloud based weak coupling model also has multiple faults like the need for constant connectivity to the central cloud or data privacy concerns.

In this thesis, local on-device weak coupling model for building intelligent applications and its prototype implementation are presented. The model is based on the concept of intelligent layer. Intelligent layer is a layer between operating system and application layer that provides intelligent services to the processes in application layer. Presented prototype implementation is called intelligence layer service. It is able to serve limited type of machine learning models represented by Open Neural Network Exchange (ONNX) format.

Keywords Artificial Intelligence, Machine Intelligence, Machine Learning, IoT, ONNX, PMML, Neural Networks, Machine Learning Model

Preface

I would to like to thank my supervisor, Prof. Valeriy Vyatkin, and my advisor Edgar Ramos, for guiding me in writing this thesis. Without Edgar's Intelligent Layer framework this thesis would have never emerged out of Ericsson's R&D Nomadic lab in Finland. Nomadic lab's R&D section and people working there are truly remarkable. I would also like to thank my former colleague Victor Morales for giving initial push for this work.

Otaniemi, 24.2.2018

Plyenkov B.

Contents

Symbols and abbreviations

Symbols

$\mathcal{Z}$	Set representing domain of raw data
$\mathcal{X}$	Feature space
$\mathcal{Y}$	Label space
$\mathcal{H}$	Hypothesis space
R^d	d-dimensional real vector space

Operators

$\mathbf{x} \cdot \mathbf{y}$	Dot product between two real space vectors $\mathbf{x}$ and $\mathbf{y}$
$\circ$	Function composition operator

Abbreviations

AI	Artificial Intelligence
DSL	Domain Specific Language
IoT	Internet of Things
IPC	Interprocess Communication
IR	Intermediate Representation
MI	Machine Intelligence
ML	Machine Learning
NN	Neural Network
ONNX	Open Neural Network Exchange
PMML	Predictive Model Markup Language

List of Figures

List of Tables

Listings

1 Introduction

The recent achievements of Artificial Intelligence (AI) [1] excited companies and government agencies [2] to develop strategic initiatives of applying AI in different domains, to create new types of intelligent systems, to provide new and improve current services, enhance and automate processes, and extract value from the massive amount of data that is being collected and stored in big datacenters all over the world. Internet of Things (IoT) is not an exception to the trend, especially considering massive amount of data that is forecasted by many to be produced by IoT devices and systems.

1.1 What is IoT today?

By reviewing the literature on IoT, it is difficult to form a clear and definite understanding of what IoT represents, because the IoT concept is usually linked to some of its building blocks and not to the complete system of all required building blocks. Adding to the confusion are companies that are re-branding their products under IoT name for marketing purposes. [3]

Based on the definition of IoT provided in [3], the building blocks that form a conceptual framework of IoT are:

- Global connectivity infrastructure, like **Internet** network, that enables connectivity and interoperability between computing devices.
- Physical objects or things, that are equipped with sensor and actuators, from the environment around us or on us, like assembly line on a factory floor, power grid, buses, cars, thermostats, light bulbs, mobile phones, watches, hearing aid equipment or household appliances, and their virtual representation in the cyber-space. These virtual representations are usually referred to as cyber-twins, digital twins or device shadows.
- Technologies that enable autonomy and self-management of things. It is expected that the thing will continue to operate in the sensible manner even after connectivity to other things or global network infrastructure is lost or if one of the objects will start to malfunction. Self-management is the form of initial configuration and life-cycle management in terms of onboarding devices and receiving software updates without end-user interaction.
- Effective human-to-thing and thing-to-thing interfaces that allow physical things embedded with electronics to cooperate with other things, humans and virtual things located in cloud.
- Solutions that enable the coexistence and cooperation of heterogeneous compute devices and networking technologies.
- Services associated to things. For example, camera connected to Internet providing intrusion detection service or thermostat proving a service of heating

control and energy consumption optimization. The services might be combined so that thermostat can communicate to intrusion camera to detect the presence of the owner and adjust temperature accordingly. At the same time it might monitor the geolocation of owner by requesting location from owner's phone or watch. This is an example of service composition in the application domain.

The use-cases of IoT are smart homes, smart factories [4], smart electrical grids, smart cities and event smart labs [5]. IoT applied in the industrial or enterprise environment is usually named Industrial Internet of Things (IIoT) to differentiate it from consumer market IoT. Some of the use-cases have overlapping objectives. For example, the energy consumption optimization is crucial service throughout the domain of different human activities.

Tangible example of IoT related to smart city use-case are air quality monitoring stations around Helsinki Metropolitan area Figure 1.1 **(a)**. The information from these sensors can be viewed by citizens during their subway trip to work or from the Web using Web-Browser[1]. Another example, again from Helsinki Metropolitan area, are open APIs provided by Helsinki Regional Transport Authority (HSL in Finnish Helsingin Seudun Liikenne) that allow, among other things, to programmatically track real-time geolocation and other information about public transport on the roads in Helsinki Metropolitan area Figure 1.1 **(b)**. By using these APIs it is possible to write application that gathers traffic data or allows user of the application to make real time decisions like choosing an optimal route to work[2].

Staying with the smart-city use-case, major Finnish energy company Fortum is installing small mobile phone size devices to the households in the city of Espoo Finland that use AI based control system to ensure stable temperature in apartments and make the heating more sustainable Figure 1.1 **(c)**. The service provided by Fortum also allows people, who have the device installed, to monitor humidity and temperature in their apartments using a Web-Browser or mobile phone application. All these activities are part of Fortum's SmartLiving program.

One example of IIoT is open data access provided by Finnish national electricity transmission grid operator FINGRID. Near real-time time grid operation measurements are available in machine readable form. The list of all available open data is long, next only two representative examples are mentioned: frequency of the power system, power production from different source of energy like nuclear or hydro[3].

Above real-world examples, that are in common use today, are all about collecting and presenting data from sensors connected to the Internet. But, as it was mentioned previously, IoT is not just a platform to collect and visualize data from different sources. It is also about, of what to do with the collected data, extracting valuable information out of it, using sensors measurements to achieve some goal and many other things that are presented in the list above. Every example of the use-case presented is prefixed with word smart, hinting that IoT systems are expected to be embedded with intelligence.

[1]More information available at https://www.hsy.fi/en/residents/theairyoubreathe/Pages/default.aspx

[2]More information about APIs is available at https://www.hsl.fi/en/opendata

[3]Open data sets and APIs are available at https://data.fingrid.fi/en/

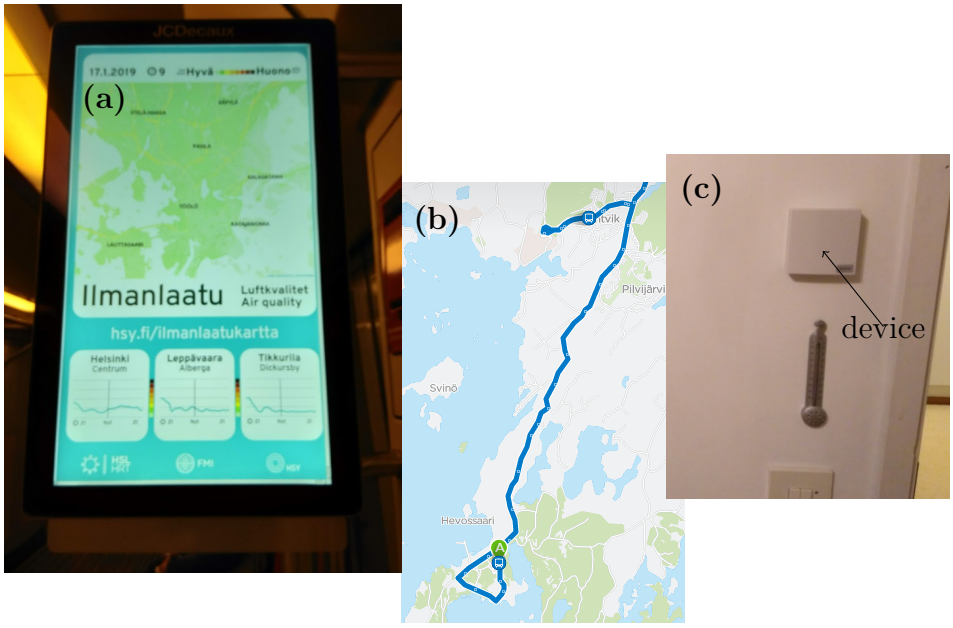

Figure 1.1: IoT use-cases around us: **(a)** Screen in the metro wagon showing air quality report in Helsinki Metropolitan area. **(b)** Tracking of a bus in real-time. Service provided by HSL. **(c)** Devices installed by Fortum as part of their SmartLiving program, in apartment houses in the city of Espoo Finland.

1.2 Artificial Intelligence in IoT

One easy to understand application of artificial intelligence in IoT is Machine Learning (ML) applied to automated data analysis, also known under the name of data mining or big data. A challenging problem, that has a potential to be solved by IoT machine learning is collection of heterogeneous data from many different sensors and devices, all connected to the same global network connectivity infrastructure, turning data into valuable information from which long-term knowledge and proactive decision making can be deduced.

Until recently, the state-of-the-art approach of doing IoT machine learning was to collect data from the IoT devices, forward it and process it in the central cloud, either real-time or after data is collected. In other words, performing ML model training, historical analytics or real-time analytics in the cloud where data and compute power are highly available and distributed over many nodes. From the need to handle these large amounts of data, not always produced by IoT devices, emerged software for doing computation in such distributed systems like Hadoop and Spark and later followed extensions to those systems that enable applying machine learning in these highly distributed environments.

It was realized that the cloud approach is not applicable for use-cases where data-privacy, real-time actuation response, intermittent connectivity to cloud, bandwidth constrained data transport, are to be considered. Autonomous car would not send data to cloud for processing to decide does it see a red light or not, the latency is simply too high and will not allow the car to make a timely decision regardless of

the connection quality to the cloud. Also, it would not make sense to send all of the telemetry data generated by the car without any pre-processing, the volume is simply too great, and the value of sending all the data is usually not enough to justify bandwidth and storage capacity usage. That is the reason of why the current state-of-the-art in IoT machine learning is to combine massive capabilities of the cloud with the ability to pre-process the data already on the edge devices or do the inference based on the measurements locally. It might also be the case that security concerns or national regulation prohibit sending data for processing and storage to a cloud, hence making edge processing the most acceptable option. [6]

The data analysis applications, that are using machine learning techniques, are usually trained in the cloud or in the local computing infrastructure of developers and data scientists. After training and tuning process, the application is deployed to production environment, meaning edge device or a cloud compute instance using cloud-based management system or by some other means, for example integrating machine learning code into the application code.

How to distribute the intelligence so that it can be implemented in one environment and then deployed to various other environments is import subject that will be studied in this thesis.

1.3 Goals and Scope of the Thesis

In this thesis, the deployment of trained machine learning models to IoT devices is explored. IoT devices that are considered in this thesis are at least capable of executing Linux operating system. An example of such a device might be a mobile phone, IoT board with Intel processor for embedded applications or even a personal computer.

A prototype application was developed, in an attempt to decouple the code responsible for application logic like user interface from implementation of intelligence. The work is based on the concept of Intelligence Layer (IL) framework proposed in [7]. The prototype is called intelligence layer service. One of the tasks of intelligence layer, as presented in [7], is to provide intelligence as a local service to the applications that need it. The concept of IL is not restricted to just being a platform for serving intelligence to client applications, but in this thesis machine learning models' description and serving aspects of IL framework are explored.

Decoupled trained ML model can be represented by a programming code in the form of a library or package for some specific programming language. Other approach is to have a compute instance in a central cloud, executing model code on HTTP request from the client program and replying with execution result. Third option is to use standardized Intermediate Representation (IR) format to represent a ML model. Special service will read a model file, turn it into runnable code and expose it to client processes via some Interprocess Communication Protocol (IPC). In this work, the third approach is explored.

Standardized IR format describes machine learning model in a library and platform independent way. Two state-of-the-art platform-independent model description formats are presented in Chapter 3. One of them is used for the IL service prototype.

The main benefits from decoupling the intelligence from application using standardized IR format, is to allow machine learning solution providers and data scientists, to use any existing machine learning library, that supports the standardized format, to develop their models and then export them in the standardized format to central model repository.

Users wanting to integrate the intelligence into their IoT environment production code, will be able to buy or download the model from the central repository and use it in their IoT environment to perform intelligent tasks. The user can use any machine learning library or runtime that is capable to translate standardized IR into library's or run-time's native IR format and execute the computations on a target device. This will allow the user to choose the computation platform that is most suitable for his task and compute requirements.

This approach has also a potential to make a deployment of machine learning models easier and democratize the machine learning space by not bounding model producers and consumers into specific libraries, run-times or cloud services. The approach can also allow model users to switch between different models and model providers if they choose to do so, without major changes to their code.

The major contribution and goal of this work is to study the possibility of decoupling the intelligence from applications and research state-of-the-art solutions that enable this decoupling.

1.4 Structure of the thesis

Chapter 2 and **Chapter** 3 provide a general background information related to machine learning. **Chapter** 2 covers the theory behind artificial intelligence and machine learning. It also has as subsection covering the most frequently used machine learning models in IoT data analyzing applications.

Chapter 3 presents practical aspects of machine learning like what state-of-the-art libraries are used to create machine learning applications and two current state-of-the-art formats for distributing machine learning models: Open Neural Network Exchange (ONNX) and Portable Model Markup Language (PMML). Some examples are provided showing how machine learning code, for simple ML problems, looks like. It is assumed that reader is relatively familiar with `Python` and `C++` programming language and examples are not explained in detail.

Chapter 4 describes a prototype implementation of IL machine learning model serving concept. The software architecture of the IL service implementation is presented and implementation details of the architecture are explained.

In **Chapter** 5 results of using IL service implementation from **Chapter** 4 for serving computationally demanding image classification ML model are described. In the chapter a quick comparison between PMML and ONNX IR formats is provided and motivation for using ONNX as a model format for the prototype implementation is explained. It is also explained, how ONNX model must be modified in order to prepare it for successful provisioning to the device executing IL service.

2 Machine Intelligence and Machine Learning

Machine learning is a subset of AI techniques and methods for building artificially intelligent systems [8]. In recent years, one particular class of function-based machine learning methods called deep learning was used successfully to solve artificial intelligence tasks that were difficult to solve using classical function-based machine learning techniques or other traditional model-based AI methods, like rule based- or represent-and-reason methods [9, 10]. Solutions to problems related to object recognition and localization in images or natural speech recognition were improved significantly by applying deep learning techniques, and new applications in different domains emerge all the time [11]. Even more impressive are combination of model- and function-based techniques as shown by current artificial champion AlphaGo in the game of Go [10, 12]. These recent advances in AI created a lot of interest in general public outside the scientific community, in research community, industry and government to apply machine learning and AI to various application domains.

In this chapter, fundamental ideas related to artificial intelligence and machine learning are presented. First, the concepts of machine intelligence and intelligent agent are considered. After that, it is presented how to formally specify machine learning problem. Some classical machine learning models and basic building blocks of a deep learning models are also covered.

2.1 Machine Intelligence

The original big goal of AI was to create a machine that is capable of achieving human level of intelligence, reaching the human's level of performance in cognitive tasks [10]. **Machine Intelligence (MI) is limited AI**. The objective of MI is not to reach human level of intelligence, but to solve tasks that are considered intelligent. Classifying a piece of text as a funny story can be considered as machine intelligence. Many orders of magnitude more difficult task for a machine, of writing a funny story is outside of the scope of machine intelligence. Automation of tasks currently done by humans like controlling a vehicle in the real-world environment, doing customer support via phone or chat are also examples of domains where machine intelligence is in the process of being applied or is already part of the production service [13].

It is not easy to define machine intelligence precisely because definition of intelligence itself is difficult. The Oxford English Dictionary defines intelligence as a faculty of understanding. The word faculty, in the context of intelligence, means natural or acquired ability to do something. Expanding the definition by including meaning of faculty one gets:

Intelligence is natural or acquired ability to understand.

If in front of the definition above word machine is added, then it is better to replace word natural with intrinsic. There is nothing natural in computer program or current state-of-the-art electronic computing hardware created by humans. Natural things are created by nature. Things created by humans are artificial.

Living beings act based on their reflexes and understanding to survive, spread and achieve some objectives and machines can be programmed to do the same. Based on presented chain of reasoning the machine intelligence can be defined as:

Machine Intelligence is an intrinsic or acquired ability of a computing machine or a computer program to understand and carry out actions to achieve objectives and goals of its creator.

In the definition above, creator is considered to be a person or a group of people that designed and implemented machine intelligence. The intrinsic ability might be represented by a model or function suitable for a specific problem. The process of acquiring ability to understand is learning. Learning might represent fitting a curve to available training data or finding model parameters. The learned model can be used to perform an intelligent task.

The notion of intelligent agent is fundamental and useful mental tool for analysing and designing systems with machine intelligence capability [14]. Next subsection explains the notion.

2.2 Intelligent Agents

The term agent is used in many closely related areas of science and technology but doesn't have a single universally accepted definition [15]. In [15] two views of agency are presented: a weak notion and a strong notion.

In a weak notion of agency, the agent is a software process or a combination of software and hardware that has the following four properties [15]:

1. Autonomy: agents have control over their internal states and actions and can operate without external interaction by other agents or humans.

2. Social ability: agents are capable of communicating with each other using some common communication protocol.

3. Reactivity: ability to understand their environment and ability to react to changes in the environment.

4. Pro-activeness: Agent can take an initiative and act preventively in order to achieve their goals.

For a stronger notion of agency, in addition to the properties from the weaker notion, agent is represented using concepts more applied to humans like knowledge, believe, desire and intention [15]. The stronger notion of agency is nearer to the big goal of AI as it was presented in the introduction to this chapter. Hence, for this thesis, a weaker notion of agency provides suitable scope for the work.

One other important property that is treated in the context of agent systems is rationality [15]. In popular university text book [14] about AI by Russell and Norvig, authors base their presentation of AI systems on the notion of rational agent. The general definition of an agent in [14] coincide with weaker notion of agency presented

in [15]. According to [14] agent is expected to: act autonomously, apprehend their environment, maintain persistency over its lifespan, adjust its behaviour according to changes in the environment, create and achieve goals. The rational agent is an agent as described in the previous sentence that acts to achieve its objective as well as possible [14].

In [8] instead of the agent the concept of artificial intelligent system that acts rationally is used. The system is rational if as a result of interaction with its environment it can take actions to maximize a long-term return.

In the description of an intelligent agent general concept like: actions, interaction, communication, environment, goals, objectives, long-term return were used. It is important to understand that when these concepts are discussed on the conceptual level, there is a lack of concreteness. For the specific machine intelligence problem, all these concepts become very concrete.

Let's consider an imaginary problem from domain of predictive maintenance in petroleum industry, related to transporting gas over the pipelines lying deep on the seabed. The task might be to predict leaks in pipes and malfunctions in the gas pumping equipment to prevent natural disasters or to achieve a continuous operation and delivery. The environment is composed by sea, pipes, gas, equipment that enables operation of the whole system. The long-term objective of the agent is to prevent natural disasters and maximize continuous delivery of gas. The agent perceives the environment via the network of sensors scattered over gas delivery system. Actions of an agent imply to communicate to the human operator via computer screen or other digital means of communication that there is a high probability of an emergency situation. One other possibility is to place intermediate agents between agent that detect anomalies and human operators. One intermediate agent might suggest the optimal set of action that might trigger actions of the other agent that is a robot. The robot will go on-site to gather more information so that human operator can choose the best series of actions to maximize performance or mitigate the risks in timely manner. The agents must have a common protocol to communicate with each other and to the human operator.

The main task of rational agent is to map precept sequence to actions effecting environment in a way that maximizes performance measure. The percept sequences are obtained from sensors and actions are applied to environment via actuators. The agent is described mathematically via agent function. The agent function maps from a set of percept sequences S to a set of actions A:

$$f : S \to A$$

The actuators and sensors are not necessary things that effect physical world directly. They might be buffers in computer memory or sockets in a network connection. The full description of performance measure, environment, actuators and sensors is called task environment. The machine intelligence problem is usually approached by first specifying task environment. The software implementation of agent function is called agent program. The computing devices with sensors, actuators

and system level software that form and execution environment for an agent program is called architecture. [14]

Currently the preferred method for creating agents is to create a learning agent [14]. One might even consider saying that it is impossible to implement machine intelligence systems in complex task environment without machine learning [8].

2.3 Machine Learning

Machine learning is mostly concerned with fitting a mathematical model to data and improving the model over-time using the observed evidence from the new data. Every component of an intelligent agent can be improved or implemented by using machine learning techniques [14]. The model might be represented by a function mapping inputs to outputs. In the introduction to this chapter, this approach is named the function-based machine learning. On the other hand, the model might contain some prior knowledge about the environment represented by probability distributions or general first-order logic. Probability theory and first-order logic are formal languages for describing environments. Machine learning techniques that utilize these formalisms are called model-based machine learning techniques. [8, 14, 10] In this thesis, only function-based machine learning models are considered. For a complete review of function- and model-based ML models reader can refer to classical text on AI like [14] by Russell and Norvig.

There are three principal types of machine learning: supervised, unsupervised and reinforcement [14]. The difference is best understood by considering three components that are required to formally specify the ML problem [8]. These components are:

1. Raw data points, features extracted from raw data points and labels related to data points.

2. Hypothesis space.

3. Loss function.

Data points are raw data precepted by an agent utilizing its sensors. The set containing all possible values that can be assumed by a data point $\mathbf{z}$ is denoted by letter $\mathcal{Z}$. The labels are answers to questions or predictions that the model gives as output. Domain of all label value $\mathbf{y}$, sometimes word target is used instead of label, is denoted by $\mathcal{Y}$. Most often data points are pre-processed before they are given to the model. As result of this pre-processing, data point is transformed into a vector of features $\mathbf{x}$, extracted from the data point. Symbol $\mathcal{X}$ is assigned to a space of all the possible feature vectors. [8] Using the mathematical notation, the mapping from data points to labels can be represented by the following expression:

$$f : \mathcal{Z} \to \mathcal{Y} \tag{1}$$

What is referred to as a model is usually a mapping $m : \mathcal{X} \to \mathcal{Y}$, hence f in 1 can be rewritten as $f = f_{\mathbf{x}} \circ m$ where mapping $f_{\mathbf{x}} : \mathcal{Z} \to \mathcal{X}$ is a feature vector extractor.

The hypothesis space $\mathcal{H}$ is a set of all computationally feasible models that are considered as candidates for the final model m. Hypothesis h, also called predictor map, from $\mathcal{H}$ is usually a mapping from feature space $\mathcal{X}$ to label space $\mathcal{Y}$. [8]

$$\mathcal{H} = \{h^{(i)} : \mathcal{X} \to \mathcal{Y} | h^{(i)} \text{ is computationaly feasible}\} \tag{2}$$

In order to choose an optimal hypothesis h that will be the final model m, from hypothesis space 2, a measure of how good the model is performing is required. The goodness of a hypothesis is evaluated using loss function l 3. The loss function is mapping from set-cross product of sets $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{H}$ to real numbers R. The hypothesis h that minimizes the average loss function is considered optimal and it is chosen as final model m to make inferences.

$$l : \mathcal{X} \times \mathcal{Y} \times \mathcal{H} \to R \tag{3}$$

In **supervised** learning problems labeled data points $(\mathbf{z}^{(i)}, y^{(i)}) \in \mathbb{Z} \subset \mathcal{Z} \times \mathcal{Y}$ are provided by the hypothetical teacher. These labeled data points form a numerable training set $\mathbb{Z}$. The job of the model creator is to find the best possible model $m \in \mathcal{H}$ that is consistent on training data and generalizes well on new data points that are not in $\mathbb{Z}$. Model is said to be consistent if it achieves near 100% accuracy over the training set. Usually, it is not possible to achieve 100% consistency and get high prediction accuracy on new unseen data points. If the model is predicting accurately labels for data points that it has not used for training, it is said to generalize well to new data. If the model is consistent but does not generalize well to new data, it is said to suffer from over-fitting.

In **unsupervised** learning, only data points are provided $\mathbf{z} \in \mathbb{Z} \subset \mathcal{Z}$. There are no labelled data points available. The job of the model is to group the data points based on some metric of similarity.

In **reinforcement** learning, the model or agent learns by receiving reinforcements from the environment. Unlike in supervised learning, there is no teacher to provide correct labels to the model. Based on the input received from the environment model must decide how it should adjust itself in order to improve the performance. AlphaGo, current artificial champion in the game of Go, is an example of applying reinforcement learning problem to solve real world problem [12]. No human was able to beat the AlphaGo to date. In this chapter, only supervised and unsupervised learning models are considered.

Inputs and outputs of a model can be represented by **quantitative** or **qualitative** variables. Quantitative variables are measurable quantities represented by real numbers like pressure, voltage or velocity vector of a point. Note that quantitative variables are not necessary scalars. For example, velocity vector of a point in an open space is represented by three real number. Qualitative variables take values from a finite countable set of distinct elements. Elements of the set define categories. For that reason, qualitative variables are also called **categorical**. An example might be a set of integer number from 0 to 9 representing decimal digits.

The models outputting categorical labels are called classifiers and the models with quantitative labels are called regression models or predictors. Models that have both types of outputs are called mixed models [8].

For some machine learning classification problems that have by definition discrete label space the hypothesis space might contain mappings which map input to continuous variable. The final prediction is then made using a **decision function** that maps the values returned by hypothesis to labels. If d is decision function, then f from 1 can be rewritten as the following mapping composition:

$$f = f_{\mathbf{x}} \circ m \circ d$$

One more distinction exists between the machine learning models. They can be **parametric** or **non-parametric**. Parametric models are described using a finite set of parameters. For example, if scalar value is predicted by the model that is represented by polynomial function, then coefficients are parameters. Knowing the coefficients, prediction can be calculated. Non-parametric models use feature vectors obtained from the data points to do prediction or classification.

The loss function 3 averaged over available feature vectors from $\mathbb{X}$ is called **empirical risk**, where $\mathbb{X}$ is defined by:

$$\mathbb{X} = \{(f_{\mathbf{x}}(\mathbf{z}^{(i)}), y^{(i)}) | \quad (\mathbf{z}^{(i)}, y^{(i)}) \in \mathbb{Z}\} \tag{4}$$

The empirical risk is usually defined so that the main objective of machine learning algorithm is to minimize the risk. In mathematical terms empirical risk is a high dimensional function. Machine learning algorithm finds the best possible function from the hypothesis space, given observed data, that produces the global or local minimum of the high dimensional empirical risk function.

Now, that the basic theory of ML is covered, some common machine learning models used in IoT domain will be considered.

2.4 Machine Learning in IoT domain

The major use of machine learning in IoT domain is data analytics. In [16] an extensive evaluation and literature review of most frequently used machine learning techniques applied to data produced in the smart city environment was conducted. The result of this work is presented in Table 2.1. The table is modified from its original form. As can be seen from the table, most of methods can be used for both classification and regression tasks and there are more supervised than unsupervised methods. This reflects the fact that supervised learning techniques are much more developed than unsupervised methods. Supervised methods are also more widely used in applications.

The first column of Table 2.1 describes the name of the model. Second column describes the type of machine learning: supervised or unsupervised. Third column describes is model used for classification or regression tasks. Note that most models

Table 2.1: Common machine learning models for IoT.

Model Name	**Type**	**Usage**	**Parametrization**
K-Nearest Neighbors	Supervised	Classification Regression	Non-parametric
Naïve Bayes	Supervised	Classification	Parametric
Support Vector Machine	Supervised	Classification Regression	Non-Parametric
Linear Regression	Supervised	Regression	Non-parametric
Decision Trees	Supervised	Classification Regression	Non-parametric
Bagging	Supervised	Classification Regression	Depends on the base model
Random Forest	Supervised	Classification Regression	Non-parametric
K-means	Unsupervised	Classification	Parametric
Principal Component Analysis	Unsupervised	Dimensionality reduction Feature extraction	Parametric
Feed Forward Neural Networks	Supervised Unsupervised	Classification Regression	Parametric

can be used of both tasks. Fourth column describes, is the model parametric or non-parametric.

Table 2.2 shows typical feature space, label space and hypothesis space for models presented in Table 2.1. Second column after the name column is for input feature space. Third column describes output label space. Fourth column is a short description of model with mathematical description of its hypothesis space or decision function. Note that dimensions and elements of the spaces are picked randomly. The space of real numbers can be replaced by any other metric space. Metric space is set of elements that have a definition for a measure of distance between any two elements from the set. The hypothesis spaces presented in Table 2.2 might not match precisely with other formulation provided in the literature. Because neural networks are currently among the most popular machine learning methods, they are covered in more detail in next section. Next sections also shows how the concepts presented in Machine Learning section apply to the most basic type of neural network.

Table 2.2: Description of common machine learning models from Table 2.1.

Model Name	**Feature Space $\mathcal{X}$**	**Label Space $\mathcal{Y}$**	**Description and hypothesis space**
K-Nearest Neighbors Classification	$\mathbf{x} \in R^2$	$y \in \{a, b, c\}$	$\hat{y} = h(\mathbf{x}, k, \mathbb{X})$ where is $\hat{y}$ is predicted label, k is number of neighbors and $\mathbb{X}$ is training set with feature vectors extracted from raw data points and labels as defined in 4. Using some metric like Euclidean measure, distances between input vector $\mathbf{x}$ and k nearest vectors from $\mathbb{X}$ are calculated and label is assigned using majority vote. For example if $k = 3$ and 3 nearest neighbors have labels a, a, b then $\hat{y} = a$.
K-Nearest Neighbors Regression	$\mathbf{x} \in R^2$	$y \in R$	For regression, label space is continuous and $\hat{y}$ is equal to the mean value calculated from the labels of the k nearest neighboring points from training set $\mathbb{X}$.
Naïve Bayes	$\mathbf{x} \in R^2$	$y \in \{a, b, c\}$	The hypothesis space is parametrized by conditional probability distributions of the elements of feature vector conditioned on the label values and the so called prior probability distribution of labels. The decision function is: $$\hat{y} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} P(y)P(x_1|y) * P(x_2|y)$$ The feature vector $\mathbf{x}$ is assumed to be random variable and its elements x_1 and x_2 independent random variables. This assumption of independence is the reason why the method is called naive. Prior $P(y)$ is usually estimated by the frequency of occurrence in training set. If size of the training set $\mathbb{X}$ is N and point with label a occurs n_a times in $\mathbb{X}$ then prior $P(a)$ is estimated by $P(a) = \frac{n_a}{N}$. Conditional probability $P(x_i|y)$ is usually estimated by a probability distribution in the form of a function like Gaussian or exponential probability distribution. The parameters for chosen distribution, like mean and variance are estimated from training data.

Support Vector Machine Classification	$\mathbf{x} \in R^d$	$y \in \{-1, 1\}$	Two class classification problem is provided as an example. This model can be extended to multiple classes using one-versus-rest formulation. The classifier is defined by: $$h(\mathbf{x}) = sgn(\sum_i^n \alpha_i k(\mathbf{x}^i, \mathbf{x}) + b)$$ Here α are real numbers solved by the learning algorithm using the training data. k is kernel function defined by: $$k(\mathbf{x}, \mathbf{x}') = \theta(\mathbf{x}) \cdot \theta(\mathbf{x}')$$ Where θ is some vector valued function and $\cdot$ denote vector dot product. Hence the classification function in hypothesis space depend on the choice of θ and samples from training data. b is bias and sgn signum function that returns 1 if input is positive number and -1 if input is negative number. [17]
Support Vector Machine Regression	$\mathbf{x} \in R^d$	$y \in R$	The hypothesis space for SVM regressor resembles hypothesis for linear regression with feature vector replaced by $\theta(x)$. $$h(x) = \mathbf{w} \cdot \theta(x) + b$$ $\mathbf{w}$ vector and intercept b are learned from training data. [17]
Linear regression	$\mathbf{x} \in R^2$	$y \in R$	The most basic technique for machine learning and a text book favorite. The hypothesis is defined by $$h(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = w_1 x_1 + w_2 x_2 + b$$ The model is parametrized by weight vector $w \in R^2$ and intercept $b \in R$.

Decision Trees Classification Regression	$\mathbf{x} \in R^2$	$y \in R$ $y \in \{0, 1, 2\}$	Decision tree learning algorithm partitions the feature space into non-overlapping subspaces, it stratifies the feature space. For two-dimensional feature space in this example, the partition is easy to visualize with a figure. Figure I. below, shows recursive binary splitting of a feature space restricted to a unit square $[0, 1] \times [0, 1]$. Points in Figure I. represent samples from the training set. Color and shape of the points represent labels. Red circle is 0, green diamond is 1 and blue square is 2. Areas in each strata form subspaces R_i. The splitting lines are defined by values t_i. [18]

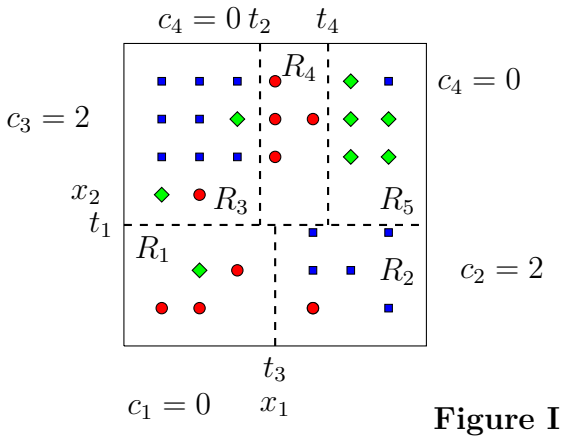

Figure I.

For each region a simple model is fitted to the data. In the case of regression, it is constant calculated as a mean of all training samples inside the region. For classification decision tree the most dominating label among the training points in the region can be chosen as a label for the whole region. In Figure I classification problem is assumed. The dominating labels are denoted by c_i in the figure. [18]
The hypothesis space is composed of all possible partitions of feature space or its subspace. The hypothesis for the classifier and predictor estimating continues values have identical equations. Equation 5 depicts hypothesis map for partition presented in Figure I. I is a function that returns 1 if $\mathbf{x}$ is inside region R_i and 0 if it doesn't belong to the region. [18]

$$h(\mathbf{x}) = \sum_{i=1}^{5} c_i I(\mathbf{x} \in R_i) \tag{5}$$

Decision Trees Classification Regression	$\mathbf{x} \in R^2$	 $y \in R$ $y \in \{0, 1, 2\}$	The decision tree is grown by recursively splitting the features space into non-overlapping regions. The technique is called decision tree because the hypothesis can conveniently be represented by a tree structure. [18]
Bagging			Bagging is also referred to as bootstrap aggregation. In bootstrap aggregation the prediction of a hypothesis map of a base model is averaged over a collection of bootstrap samples with goal of reducing the variance of estimation. [18] If there is M bootstrap samples $\mathbb{X}^{*b}$ partitioning the training set $\mathbb{X}$, the base model is fitted to each sample getting the optimal base hypothesis map m^{*b}. The final prediction is an average over base hypothesis maps [18]: $$h(\mathbf{x}) = \frac{1}{M} \sum_{b=1}^{M} m^{*b}(\mathbf{x})$$ The above formula is for regression problem. For classification problem for each class indexed by integer k the proportion of models $p_k(\mathbf{x})$ predicting class k is calculated for data point $\mathbf{x}$. Bagged classifier returns value that has the highest proportion. [18]
Random Forest			Random forest model is a modification to bagging with a decision tree as a base predictor. It constructs a large collection of de-correlated trees and then averages the predictions from them to calculate final result like in bagging. The predictor map is similar to what was presented for bagging. The correlation is reduced by random selection of features from a feature vector at each step of growing a regression tree. [18].

K-Means	$\mathbf{x} \in R^2$	$y \in 1, ..., k$	In k-means machine learning technique the algorithm clusters the points from training set into k clusters, calculating the centroid for each cluster, using some similarity measure to determine which point should belong to same cluster. k is hyperparameter representing the assumption about the data. For the data points in R^2 the similarity might be Euclidean distance or the dot product of two feature vectors. This method doesn't require labeled data. Clustering can be interpreted as extreme case of classification problem when labels for the data points in training set are not available. [8] The classifier function assigns a label to new input data point based on what centroid is nearest to the data point. The hypothesis space if formed by all possible cluster assignments and the decision functions might be: $$h(\mathbf{x}) = min(\underset{y \in \mathcal{Y}}{\operatorname{argmin}} ||\mathbf{x} - m_y||)$$ Here, m_y are cluster centers. min functions picks the class with minimal index. If distance between $\mathbf{x}$ and multiple centers is the same and $argmin||x - m_k||$ is a set that has more than 1 element then strategy is needed to choose only one class, for that min is used. [8]
Principal Component Analysis	$\mathbf{x} \in R^{13}$	$\mathbf{y} \in R^2$	Principal Component Analysis is used to reduce the number of features in the high dimensional feature vector that is fed to machine learning model as input. Using high dimensional feature vectors requires a lot of compute and memory resources to execute the model. For some models using large number of features might produce an overfitting effect. Reducing the number of features to two is sometimes useful to visually inspect data. For these reasons reducing the number of feature is beneficial. [8] The hypothesis map for PCA is linear transformation define by matrix $\mathbf{W}$. For our example this is 13 by 2 matrix. How this matrix is calculated is out of scope for our work. Interested reader can read [8] to find out more about PCA.

Feed Forward Neural Networks	$\mathbf{x} \in R^2$	$\mathbf{x} \in R$	A type of neural network without feedback loops. The hypothesis space is defined by network topology and a set of numerical parameters. The output of a neural network is usually continuous. For classification problem it can indicate the score or probability of belonging to different classes in a classification problem. Decision function for classification problem picks the class that has highest probability. Because of the current popularity of neural networks they will be treated in more detail in section 2.5.

2.5 Neural Networks

The most fundamental building block of any computational neural network is Rosenblatt's perceptron, named after a psychologist Frank Rosenblatt who published a paper presenting it in 1958. The perceptron represents a single unit of computation, that applies an affine mathematical operation and optional non-linear mathematical function to one or more scalar inputs producing a single scalar output. The perceptron is a single neuron of a neural network, and a neural network is built out of such neurons with connected inputs and outputs. [19]

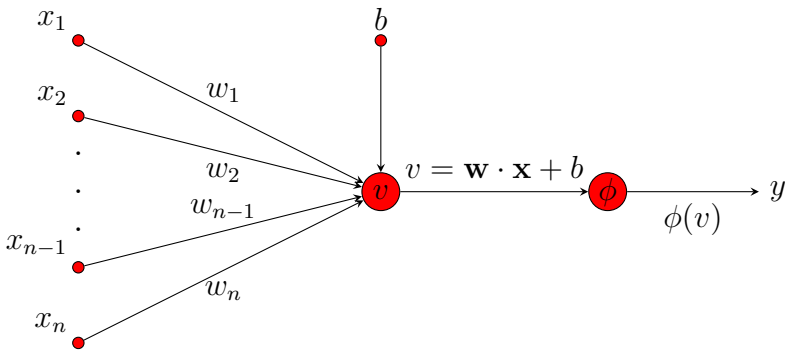

Figure 2.1: Graph diagram of Rosenblatt's perceptron.

Figure 2.1 illustrates Rosenblatt's perceptron as a directed graph. Scalars x_i, where $x_i \in R$, are inputs and can be thought of as components of m-dimensional input vector $\mathbf{x} \in R^m$. The directed arrows starting at inputs and converging at a single point are called synapses and a scalar number assigned to each synapse is called a synaptic weight. One other converging arrow with bias value b is not part of the input and is internal to the perceptron, it can be thought of as a translating component of an affine transformation. Using synaptic weights and bias perceptron does affine transformation taking a dot product between input vector $\mathbf{x}$ and a vector of synaptic weights $\mathbf{w}$ and adding bias b to the result. The result of affine operation is called local field v. Formula for local v field is:

$$v = \mathbf{w} \cdot \mathbf{x} + b$$

The nonlinear operation can be any real valued function ϕ that takes local field as input and produces final output of neuron. ϕ is also called activation function. Some choices of activation function where found to be better than others for numerical stability and theoretical inferences like mathematical proofs. In practice only, couple of well-known and well-studied function are used for activation of a neurons. The symbolic mathematical expression for the perceptron is simply:

$$y = \phi(v) = \phi(\mathbf{w} \cdot \mathbf{x} + b) \tag{6}$$

Rosenblatt's perceptron can be trained to classify a linearly separable pattern in a finite number of steps [19]. If a feature space is a subspace of R^2 and a label space consists of only two labels, the pattern is linearly separable if points belonging to different classes can be separated by a line. In higher dimensional space the line is a hyperplane.

Next, to highlight the basic concepts of machine learning presented in previous sections, an example of a perceptron that does binary classification of a linearly separable pattern in two-dimensional feature space is presented. It is assumed, that the features are already extracted from the raw data and feature space is $\mathcal{X} = R^2$. Usually the process of identifying how to extract features and what features are import is quite involved process but it is skipped here, because it is a very domain specific problem. The label space consists of two number $\mathcal{Y} = \{-1, 1\}$, number 1 identifies belonging to class $\mathfrak{C}_1$ and number -1 to class $\mathfrak{C}_2$. The model is described by a neuron with two inputs and activation function is an identity. The decision function is a signum function that outputs positive one for positive value produced by a neuron and respectively negative one for negative number. The hypothesis space is parametrized by two weights and a bias values. For that reason, the hypothesis space can be identified with R^3.

At this point: feature space, label space, model, hypothesis space and a decision function are defined. To train the model a loss function is needed, empirical risk and training data. Discrete training set of N training samples is denoted by $\mathbb{X}$. Elements of this space are tuples $(\mathbf{x}^{(i)}, y^{(i)})$ where $\mathbf{x}^{(i)} \in \mathcal{X}$ and $y^{(i)} \in \mathcal{Y}$. The loss function needs to be small when model classifies input feature vector correctly, indicating the correct prediction and the value of the loss function needs to be large when the model misclassifies the feature vector. For the example problem the following loss function l can be used:

$$l((\mathbf{x}, y), h) = \ln\left(1 + e^{-yh(\mathbf{x})}\right) \tag{7}$$

This function is called logistic loss function. The hypothesis map h is as in expression 6 with ϕ set to identity function. If the value of y is fixed to -1 and loss function is plotted as function of h, it can can be seen the function has large values

when h is positive indicating the error in classification and it is decreases rapidly when h has negative values. If y is set to unity the effect is reversed. Plots for both cases are presented in Figure 2.2.

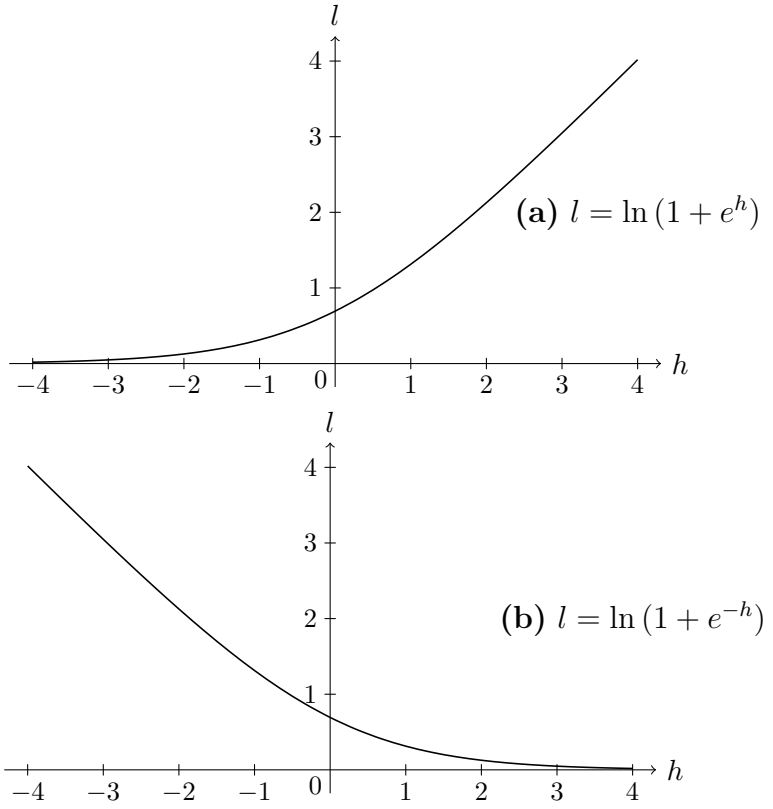

Figure 2.2: Logistic loss function 7 as a function of h for $y = -1$ in **(a)** and $y = 1$ in **(b)**.

The empirical risk `EMR` is an average of a loss function 7 over all samples from the training set and can be expressed by the formula 8. In this formula the mathematical expression is expanded showing explicitly dependence of `EMR` on elements of weight vector $\mathbf{w}$, bias b and training data from $\mathbb{X}$.

$$\texttt{EMR} = \frac{1}{N} \sum_{(\mathbf{x}^{(i)}, y^{(i)}) \in \mathbb{X}} l((\mathbf{x}, y), h) = \frac{1}{N} \sum_{(\mathbf{x}^{(i)}, y^{(i)}) \in \mathbb{X}} \ln\left(1 + e^{-y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)}\right) \tag{8}$$

$$= \frac{1}{N} \sum_{(\mathbf{x}^{(i)}, y^{(i)}) \in \mathbb{X}} \ln\left(1 + e^{-y^{(i)}(w_1 x_1^{(i)} + w_2 x_2^{(i)} + b)}\right) \tag{9}$$

Now that all three components of a machine learning problem are specified the model can be trained on the available training data. The loss function 7 has a special property. It is differentiable with respect to hypothesis h. Since h is also differentiable with respect to its parameters from the theory of calculus it is known that composition or sum of differential function is a differentiable function. Hence, the loss function 7 and empirical risk 8 are differentiable functions.

By minimizing the empirical risk with respect to parameters the best hypothesis, that can be used for inference, is obtained. The parameters $\mathbf{w}$ and b that minimize empirical risk are the ones that define the hypothesis m that is used to make inferences after training. Since EMR is differentiable function it is possible to find a critical point of empirical risk function by equating is partial derivatives to zero as in equation 10. Critical point can be local/global minimum, maximum or saddle point. Additional investigation must be performed in order find out the type of critical point. For function that are convex, like a parabola in two-dimensional Cartesian real space opening towards positive y-direction, the critical point is minimum but not necessary global minimum.

$$\frac{\partial \mathtt{EMR}}{\partial w_1} = \frac{\partial \mathtt{EMR}}{\partial w_1} = \frac{\partial \mathtt{EMR}}{\partial b} = 0 \tag{10}$$

The author presented the most simple type of feed-forward neural network consisting of only single neuron. Neural networks used for real world problems are much more complex utilizing other types of neurons that express for example convolution operation. These different types of neurons can form composite block called layer and layers can still be connected further forming very deep topologies. The outer layers are usually referred to as input and output layers, and inner layers are called hidden layers because they are not directly visible to the user of the network. Having many layers is the main reason why neural networks are called deep. Basically any network that has more than one hidden layer is considered deep.

Although in practice networks can be much more complex, as was explained in previous paragraph, the process of defining and training the network follows the same steps as presented for Rosenblatt's perceptron. In the example presented, optimal weights and bias can be calculated analytically. For complex networks and other types of machine learning algorithms, it is usually near impossible to solve optimization problem of minimizing empirical risk analytically, that is why numerical methods must be used.

If empirical risk is differentiable the most popular methods of solving for optimal values are family of methods based on gradient descent. Gradient descent method is an iterative algorithm utilizing gradient of empirical risk to iteratively, step by step improve the estimate of a critical point. Gradient is vector field, the values of its components are derivatives taken at some point on the surface of empirical risk with respect to parameters and bias. Geometrically speaking empirical risk defines a surface in high dimensional space similar to equation of a sphere in 3-dimensional space. Being able to calculate derivatives is critical in utilizing gradient descent method and most of software for creating and training neural networks implement efficient data structures and algorithms for calculating gradients of complex function representing neural networks.

3 Machine Learning Libraries and Formats

In the beginning of this chapter two popular state of the art software libraries for creating, training and executing machine learning algorithms are presented. After that, two library independent formats for distributing machine learning models are considered. Because of the recent achievements of deep learning models [1], the most popular and hyped machine learning libraries are heavily specialized for creating, training and executing, data and computing hungry, deep learning architectures. Most of the libraries are available as open source software, but behind each popular deep learning library stands one of the IT industry giants like Facebook, Google or Amazon. Many deep learning libraries exist, only a few are mentioned next: TensorFlow (Google) [20], Caffe2 (Facebook), PyTorch (Facebook) [21], MXNet (Amazon) [22] and more exotic based on OCaml functional programming language OWL [23].

One can classify deep learning libraries based on the programming model provided for defining computations to a software developer and data scientist. Currently, two programming models, also word paradigm can be used, exist: declarative approach and imperative approach. In Caffe2 library, data scientists can only use the declarative approach. In PyTorch, developers can only use the imperative programming model for writing ML code. MXNet supports both paradigms and for that reason it will be presented in this chapter, after the section about declarative and imperative approaches.

In this chapter classical machine learning `Python` library scikit-learn is also presented. scikit-learn library [24] is scientific computation environment for doing machine learning and statistics calculations that can be considered a traditional imperative library for doing data-analytics and creating machine learning applications.

Common, to almost all machine learning libraries, is that they are partially written in `C`, `C++` or `Fortran`[4], that is so called back-end implementation of the library, for efficiency, but APIs for the users of the library are usually exposed via one or multiple other languages that are considered more convenient to use. Python being the most popular interfacing language at the time of writing this thesis.

After understanding how ML models are created, formats for enabling interoperability between different machine learning libraries are covered. Interoperability is to be understood as having a common format for sharing ML models between different ML libraries. Two formats are presented: Open Neural Network Exchange (ONNX) [25] and Predictive Model Markup Language PMML [26]. The ONNX format will be used as a base for the proof of concept intelligence layer service prototype in later chapters of this work.

3.1 Differences between declarative and imperative approaches

In most state-of-the-art deep learning libraries, neural networks are represented by Directional Acyclic Graphs (DAGs) also called dataflow graphs. This is done to allow

[4]Mathematical libraries written in `Fortran` are still used for scientific calculations, especially linear algebra e.g. `LAPACK`.

efficient execution of neural network in training and inference and to make possible efficient calculation of partial derivatives by reverse mode algorithmic differentiation. As was described in Chapter 2, the calculation of partial derivatives is a necessary step for stochastic gradient descent, main technique used for training neural networks and other machine learning models when the loss function is differentiable.

In **declarative approach** to programming deep learning models, the computational graph is defined before any computations take place. After the graph is defined, it is compiled into executable code and then used on every iteration either for inference or training. The graph is **static**. The declarative approach is also often described as **define-and-run** method or **symbolic** approach to machine learning.

Imperative approach can be characterized by the phrase **define-by-run**, notice the **by** in the phrase. The graph is constructed **dynamically** on every iteration. This allows developer to use familiar style of imperative programming for constructing a graph, mixing construction of graph with control constructs of the host programming language like conditionals: *if*, *else* and looping statements: *for*, *while*. It also allows to get intermediate results of calculation much easier than with declarative approach.

The imperative approach is more flexible than symbolic approach. It allows to write and experiment with different deep learning parameter optimization techniques, variation of stochastic gradient descent for example, and network typologies much easier than with rigid static graphs. Declarative approach with static DAG allows for more performant code that can be parallelized and optimized for faster execution much easier than code written with imperative approach hence making training and inference faster.

Some libraries combine two approaches to benefit from both. For example, OWL and MXNet use both techniques. In the code Listings 3.1 and 3.2 the author presents MXNet code for training a one neuron perceptron classifier with two inputs and a logistic loss function. The code in Listing 3.2 is an example of imperative approach. Code in Listing 3.1 is an example of declarative approach. Code for generating synthetic training data is omitted from the listings.

Listing 3.1: Declarative MXNet code for training Rosenblatt's perceptron with two synapses, for a binary classification problem using gradient descent method.

```
1  import mxnet as mx
2  import training_data
3
4  # Symbol represents matrix of feature vectors from training set.
5  x = mx.sym.Variable('x')
6  # Symbol represents vector of labels from the training set
7  # corresponding to feature vectors from x.
8  l = mx.sym.Variable('l')
9  # Symbol represents synaptic weights of the perceptron.
10 w = mx.sym.Variable('w')
11 # Symbol represents bias to the perceptron.
12 b = mx.sym.Variable('b')
13
14 # Induced local field of perceptron.
```

```
v = mx.sym.broadcast_add(mx.sym.dot(x, w), b)
# Logistic loss function.
loss =  mx.sym.log(1 + mx.sym.exp(-v * l))
# Empirical Risk.
emr = mx.sym.mean(loss)
# Binding values from training data and initial values for synapses and bias.
ex = emr.bind(ctx=mx.cpu(), args={'x': mx.nd.array(training_data.X),
                                  'l': mx.nd.array(training_data.y),
                                  'w': mx.nd.array([[1], [1]]),
                                  'b': mx.nd.array([1])},
                            args_grad={'b': mx.nd.zeros(1), 'w': mx.nd.zeros((2, 1))})

learning_rate = 0.1
for epoch in range(100):
    # Calculating empirical risk and derivatives.
    ex.forward(is_train=True)
    ex.backward(mx.nd.ones(1))
    ex.arg_dict['w'] -= learning_rate * ex.grad_dict['w']
    ex.arg_dict['b'] -= learning_rate * ex.grad_dict['b']

print("Synaptic Weights")
w_final = ex.arg_dict['w'].asnumpy()
print("w_1 = {:.3}, w_2 = {:.3}".format(w_final[0,0], w_final[1,0]))
print("Bias")
print("b = {:.3}".format(ex.arg_dict['b'].asnumpy()[0]))
```

3.2 MXNet

MXNet is a library for creating large-scale deep neural networks. It is possible to do general mathematical computation with MXNet, but the library was designed to help developers to utilize the full computational capabilities of multiple GPUs and cloud compute infrastructure for creating neural networks. After-all, Amazon is the main stakeholder for the project. Library offers device placement to allow the developers to easily specify in what memory data structure used in computations are stored. For example, if computations are executed on the GPU then data structures must be in GPU's memory and amount of copying between main memory and GPU memory must be minimal to avoid high latency when chucks of memory are moved via slow external buses[5]. It is possible to scale computation over multiple GPUs or network clusters. Library also provides automatic differentiation for calculating derivatives to optimize neural network's parameters.

MXNet provides predefined neural network layers. These layers are optimized for speed. They can help developers to create efficient deep neural networks out of the ready-made building blocks without the need to write code for implementing common computations. For example, MXNet has predefined layer for fully connected

[5]External to CPU cache, GPU memory and main memory. That is buses which allow to move data between different memory types.

layer where each neuron has synaptic connections to all outputs of a previous layer of neural network to which it belongs.

Listing 3.2: Imperative MXNet code for training Rosenblatt's preceptron with two synapses for a binary classification problem using gradient descent method.

```
from mxnet import nd, autograd
import training_data

# NDArray contains matrix of feature vectors from training set.
x = nd.array(training_data.X)
# NDArray contains vector of labels from the training set
# corresponding to feature vectors from x.
l = nd.array(training_data.y)
# NDArray for synaptic weights.
w = nd.array([[1], [1]])
# Allocating memory for gradient
w.attach_grad()
# NDArray for bias
b = nd.array([1])
# Allocating memory for gradient
b.attach_grad()


learning_rate = 0.1
for epoch in range(100):
    # Recording graph
    with autograd.record():
        v = nd.dot(x, w) + b
        loss = nd.log(1 + nd.exp(-v * l))
        emr = nd.mean(loss)
    # Calculating gradient
    emr.backward()
    w[:] = w - learning_rate * w.grad
    b[:] = b - learning_rate * b.grad

print("Synaptic Weights")
print("w_1 = {:.3}, w_2 = {:.3}".format(w[0,0].asscalar(), w[1,0].asscalar↩
    ()))
print("Bias")
print("b = {:.3}".format(b.asscalar()))
```

3.2.1 MXNet Model Server

The implementation of intelligence layer service presented in Chapter 4 will be used to serve ML models locally using some local inter process communication protocol that enables communication between IL service and applications requesting intelligent services. MXNet Model Server is part of the Apache MXNet project. It is software for serving machine learning models using HTTP endpoints as services providing endpoints. Hence, MXNet model server provides functionality that might seem similar to IL at first, but there are some differences.

MXNet model server is oriented more towards cloud environment compared to IL that is supposed to run on device and utilize cloud service as extra functionality. The way how models are provisioned, in the implementation of IL presented in this work, are also very different. In MXNet model server, a model that is executed, is represented by a `Python` program. Allowing for a lot of flexibility in terms of data post- and pre-processing and libraries used to do inference. This approach adds work for intelligence model provider to write additional code around the machine learning model. Implementation of intelligence layer in this work is expecting just an ONNX model file produced as an artefact during model training and development process.

It must be reminded to the reader that IL is not just about model serving [7]. But in this thesis, only the model serving aspects of IL are considered.

3.3 Scikit-learn

In the data science and machine learning community, scikit-learn machine learning library is considered rightly, an entry point into the science of data analysis and machine learning, because of its nice API design [27] and very comprehensive documentation. [24] In contrast to MXNet, scikit-learn has implementation for many different classical machine learning models and its domain is medium-scale learning. Scikit-learn is a `Python` library and it uses Numpy package's `ndarray` for representing low and high dimensional mathematical objects like scalars, vectors, matrices and tensors. These two facts make the library beginner friendly and lower the entry barrier into the machine learning world. In this subsection, the main features of scikit-learn are highlighted without considering its details. Only one example of very simple application will be provided. The interested reader can refer to extensive scikit-learn's online documentation to get more informative overview of the library.

The scikit-learn library supports supervised and unsupervised classical machine learning models like generalized linear models, decision trees, support vector machines, clustering, gaussian mixture models, fully connected multi-layer perceptron and many more other models. It also supports model selection and validation techniques like different metrics to evaluate model performance e.g. least mean square measure and cross validation like k-fold cross validation. The transformation of the data like dimensionality reduction or normalization are supported as well.

The main conceptual object of the scikit-learn library is an estimator. Estimators are objects that can be trained from the data and used for predictions. Estimators usually have *fit* method. For supervised learning, the parameters to the *fit* method are an array of data points and array of labels that correspond to the data points. For unsupervised learning estimators, only data points are provided. Most of estimators have *predict* and *score* methods to generalize to new data points and evaluate performance of the model. Transformers like PCA have *transform* method to transform the data. Transformers and estimators can be combined into a composite estimator by `Pipeline` objects.

The code in Listing 3.3 presents a simple machine learning application that is used to predict the price of one crypto-currency based on the price of other crypto currency using linear regression model. The machine learning algorithm is represented by

a pipeline that first uses transformer to standardize, scale and move all the points so that the mean and variance of the input data is 0 and 1 respectively, and then applies linear regression to data. Part of the data is used for training and part for testing the model. Note, that label data is also standardized.

Listing 3.3: Python code, showing how scikit-learn library can be used to train linear model to predict closing price of Etherium crypto-currency based on the closing price of Bitcoin crypto-currency.

```
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
import pandas as pd
from sklearn.pipeline import Pipeline

# Loading and reshaping data for the training set
# Closing prices for Ethereum and Bitcoin are stored in separate CSV files
df_bc = pd.read_csv("BTC-USD.csv", parse_dates=['Date'])
df_eth = pd.read_csv("ETH-USD.csv", parse_dates=['Date'])

bcv = df_bc.Close.values.reshape(-1, 1)
etv = df_eth.Close.values.reshape(-1, 1)

# Split the features into training and testing sets
X_train = bcv[:-20]
X_test = bcv[-20:]

# Split the labels into training and testing sets
y_train = etv[:-20]
y_test = etv[-20:]

scaler_for_labels = StandardScaler().fit(y_train)

pipe = Pipeline(steps=[('data_transformer', StandardScaler()), ('model', ↩
    linear_model.LinearRegression())])

# Training the model
pipe.fit(X_train, scaler_for_labels.transform(y_train))
# Testing the model
print(pipe.score(X_test, scaler_for_labels.transform(y_test)))
```

Scikit-learn has a very well designed APIs. It is reasonable to use Scikit-learn's API as a motivating example for designing APIs for model serving solutions like IL.

3.4 Formats for representing machine learning functions and models

Machine learning libraries usually use their own internal data structures to represent calculations or even multiple different data structures suitable for each machine learning technique. These structures are usually expressed by Domain Specific Languages (DSL) or native Intermediate Representations (IR) specific to a library.

Different libraries might also support different programming languages for creating machine learning applications.

The environment where ML model is developed and trained and where it is supposed to be executed might be drastically different. The image recognition model might be developed and trained in the cloud where computational resource can be acquired on demand. But the deployment environment, where inference on local data is performed, for the same image recognition model, might be a mobile phone or IoT device like Raspberry Pi. These facts make it difficult to share models between different libraries and deploy and integrate machine learning solutions into production quality applications.

Because of the differences between ML model development and execution environment, it is important from performance and interoperability perspective to be able to develop ML algorithms using one library and then execute or improve it using another library. This requires having a common standardized IR for representing ML models. ML library providers can use this common IR to write converters and backends to enable interoperability with other frameworks. The job of the converter is to translate from the library's native IR to standardized open IR. Backend is supposed to be able to read the model description in standardized IR and execute it or provide the access to native IR format if the purpose is to improve the model by training on different data or evaluating its performance. The standardized IR for ML models might also be considered as standard that enables distribution and provisioning of machine learning models. The aspects of distribution and provisioning of machine learning models are a key theme of this thesis.

In this section two IR formats are considered. Those formats are listed in the list below. Both of these formats are open. ONNX uses protocol buffers to define the format and encode the models. PMML is based on XML. The model expressed with ONNX is distributed as protocol buffer file and PMML model is distributed as an XML file. Based on the study of these formats, one format is picked as base for the prototype of intelligence layer.

- Open Neural Network Exchange (ONNX) format
- Predictive Model Markup Language (PMML) format

3.4.1 Protocol buffers

The Extensible Markup Language XML format used for representing PMML models is a very common encoding for documents and data structures that is human readable and machine readable. It will not be covered in this thesis. Readers who are not familiar with XML can refer to [28] for more information about XML. Protocol buffers are less well known. Protocol buffers are explained in this subsection.

Protocol Buffer (PB) is an alternative format to XML overcoming some of its shortcomings. Protocol buffer format was developed internally by Google and made available for general use as an open source project. [29] On the protocol buffer's project web page, PB is described as:

Language-, platform-neutral extensible mechanism for serializing structured data that is faster, smaller and simpler than XML.

PB is a format for serializing and deserializing data structures. It is used in communication protocols, data storage and for other things. Use-case for this thesis is: representing machine learning models.

The data structures are defined in the text files postfixed with `proto` or `proto3` postfixes. The `proto` file contains message types. The messages define structure of the data. Each message has a type and is defined by numbered name-value pairs, as shown in code Listing 3.4. Value type can be integer, floating-point, boolean, string, enumeration, raw bytes or other messages. The name-value pair can have a qualifier in front: required, optional or repeated. The required pairs are required for a message to be valid. Optional fields are as a name suggested optional. Repeated fields are kind of arrays. The message can have zero or mores repeated fields. There are two version of protocol buffer language used to define messages, that is reason to have `3` in the `proto3` prefix. The full description of PB language can be found online at [30].

Listing 3.4: Snippet from protocol buffer definition of a ONNX Tensor. Used to show basic constructs of PB message definition.

```
message TensorProto {
  enum DataType {
    UNDEFINED = 0;
    // Basic types.
    FLOAT = 1;   // float
    UINT8 = 2;   // uint8_t
    INT8 = 3;    // int8_t
    // Other types...
  }

  // The shape of the tensor.
  repeated int64 dims = 1;

  // For very large tensors, we may want to store them in chunks, in which
  // case the following fields will specify the segment that is stored in
  // the current TensorProto.
  message Segment {
    optional int64 begin = 1;
    optional int64 end = 2;
  }
  optional Segment segment = 3;

  repeated float float_data = 4 [packed = true];

  // For int32, uint8, int8, uint16, int16, bool, and float16 values
  // float16 values must be bit-wise converted to an uint16_t prior
  // to writing to the buffer.
  // When this field is present, the data_type field MUST be
  // INT32, INT16, INT8, UINT16, UINT8, BOOL, or FLOAT16
  repeated int32 int32_data = 5 ;
```

```
31 }
```

As depicted in Figure 3.1 the messages definitions file is compiled into language specific classes that can be used to serialize and deserialize data in the variety of data streams. Protocol buffer supports many languages: `C++`, `Python`, `Java` and others. That is, it provides compiler to compile `proto` files and produce native code for all these languages.

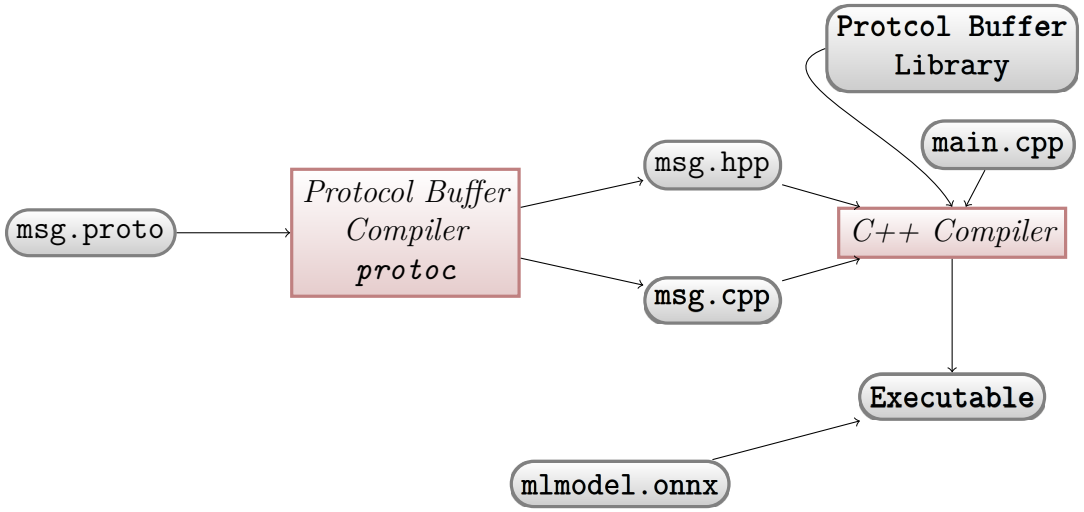

Figure 3.1: Process of producing `C++` code from protocol buffer definition file and using it in a `C++` program to read ONNX file that represents ML model.

Figure 3.1 depicts the process of compiling Protocol Buffer messages definition into `C++` classes that can be used to serialize and deserialize PB files. `msg.proto` defines the messages. `protoc` compiles `msg.proto` to `C++` code representing `C++` classes that can be used to serialize/deserialize messages and access encoded data via members provided by produced classes. Classes also have methods to view textual representation of a message. In order to use the classes in the main program represented by main.cpp, it is needed to include produced header and code files and also link to protocol buffer `C++` library. This is all done by `C++` compiler that produces `Executable` that is able to read and deserialize binary encoded `mlmodel.onnx` file.

The serialized PB message can be represented in the textual or binary format. To get all benefits of protocol buffers the binary format must be used. Using binary format reduces the size of the message that is transferred overt the wire and enables really fast parsing compared to XML. One thing to note is that `proto` file is required in order to serialize and deserialize data. Without the native code, produced from proto file by the PB compiler the binary encoded message is meaningless. Now, that the basic of protocol buffers are covered, it is possible to move on to considering ONNX that uses protocol buffers to define the IR format for ML models.

3.4.2 ONNX

ONNX is an open source format used to represent trained machine learning models. Trained ML models are ready to be used for inference. As was mentioned in the

previous subsection the ONNX model files are binary encoded non-human readable protocol buffer messages. It means that syntax for ONNX format is completely specified in the `proto` files. To created and read ONNX models it is necessary to have classes produced from ONNX `proto` files by PB compiler. In other words, these classes are used to serialize and deserialize ONNX models. The latest `proto` files describing the format are available at ONNX project's GitHub repository [31]. ONNX format was released in 2017, making it a relatively new project.

The ONNX `proto` files have message definitions for representing machine learning model's meta-data, standard data types, computational graph, that is the DAG mentioned in section 3.1, and built-in operators that are nodes of computational graph. A snippet from ONNX `proto` file specification defining the `ModelProto` message is provided on the left side of Figure 3.2. The `ModelProto` PB message is a top-level container defining the model. It is container for meta-data related to model and computational graph. An example of serialized model in textual format is presented on the right-hand side of Figure 3.2. The model represents a linear regression model that takes batch of two-dimensional floating point valued vectors as feature vectors and returns a batch of predictions. Most of the fields of `ModelProto` are self-explanatory. The list below, explains less important fields:

- `ir_version` is the version of the ONNX format. Since current version is 3. This field has value 3.
- `producer_name`: is the name of the tool used to create the model. The model from Figure 3.2 was created using scikit-learn library and exported using onnxmltools `Python` package. For that reason the field has value `OnnxMLTools`.
- `producer_version`: indicates version of the tool from previous item. In our case, version of onnxmltools package is as reported by pip `Python` package manager.
- `domain`: Reverse domain name indicating name space of the model. Together with model version and graph name, these fields are used to uniquely identify model.
- `model_version`: used to define model version.
- `doc_string`: is a textual model description.

The more interesting fields of the `ModelProto` message are `graph`, `opset_import` and `meta_props`. The `graph` defines computational graph and `opset_import` the set of operators that can be used as nodes in the graph. `meta_props` can be used to specify additional meta-data like for example authors of the model or more interestingly some additional fields to describe semantic meaning of model's inputs and outputs. For example, an $3 \times 256 \times 256$ input tensor $\mathbf{T}$ might represent color image using `T[0,0:255, 0:255]` for red color channel, `T[1,0:255,0:255]` for green channel and `T[2,0:255,0:255]` for blue channel. The type of `opset_import` filed is an array of `OperatorSetIDProto` messages. The definition of the message is provided in Listing 3.5.

```
message ModelProto {
  optional int64 ir_version = 1;

  optional string producer_name = 2;

  optional string producer_version = 3;

  optional string domain = 4;

  optional int64 model_version = 5;

  optional string doc_string = 6;

  repeated OperatorSetIdProto opset_import = 8;

  optional GraphProto graph = 7;

  repeated StringStringEntryProto metadata_props = 14;
};
```

```
ir_version: 3
producer_name: "OnnxMLTools"
producer_version: "1.2.2.0129"
domain: "onnxml"
model_version: 0
doc_string: ""
graph {
  node {
    input: "input"
    output: "variable"
    name: "LinearRegressor"
    op_type: "LinearRegressor"
    // Some fields are removed
    domain: "ai.onnx.ml"
  }
  name: "linear_model_graph"
  input {
    name: "input"
     type {
      tensor_type {
        elem_type: FLOAT
        shape {
          dim {
            dim_param: "None"
          }
          dim {
            dim_value: 1
          }
        }
      }
    }
  }
  output {
    name: "variable"
    type {
      tensor_type {
        elem_type: FLOAT
        shape {
          dim {
            dim_param: "None"
          }
          dim {
            dim_value: 1
          }
        }
      }
    }
  }
}
opset_import {
  domain: "ai.onnx.ml"
  version: 1
}
```

Figure 3.2: Snippet from ONNX `proto` definition of a top-level model container. Right side shows textual representation of complete deserialized ONNX model.

Listing 3.5: Definition of `OperatorSetIDProto` message, used to specify operator set.

```
// Operator Sets
//
// OperatorSets are uniquely identified by a (domain, opset_version) pair.
message OperatorSetIdProto {
  // The domain of the operator set being identified.
  // The empty string ("") or absence of this field implies the operator
  // set that is defined as part of the ONNX specification.
  // This field MUST be present in this version of the IR when referring to any other operator set.
  optional string domain = 1;

  // The version of the operator set being identified.
  // This field MUST be present in this version of the IR.
  optional int64 version = 2;
```

```
14 }
```

As can be seen, `OperatorSetIDProto` message is defined by two fields, domain that is a string and version that is an integer. Each node in the `graph` must belong to one of the domains with version specified in `opset_import` for the model to be valid. In the example depicted in Figure 3.2 the operator domain is `ai.onnx.ml`.

Currently there exists two domains of operators. Domain for deep learning model is called `ai.onnx`. If domain is not specified `ai.onnx` is assumed by default. The domain of the model from Figure 3.2 defines set of operators for classical machine learning models. Since ONNX has its focus on deep learning models the `ai.onnx` domain has much larger set of operators than `ai.onnx.ml`. 133 operators in ai.onnx domain against 18 operators in `ai.onnx.ml` domain. The set of operators for both domains are provide in Appendix A and are also available on the Web at project's GitHub page. It must be noted that default domain also includes operators representing common mathematical operations like addition or multiplication.

The operators set is not the only difference between classical and deep machine learning models in ONNX. Operators that are nodes of the computational graph can have multiple inputs and multiple outputs. For operators from default domain only dense tensor types for inputs and outputs are supported. Classical machine learning operators in addition to dense tensors, support sequence type and map type.

In ONNX code base, the `proto` files for base structures of the model are available as regular textual files. For the operators, unfortunately, the case is different. They are created programmatically, making adding additional operators or reviewing the existing ones difficult. In theory it is possible to extended ONNX with the set of custom operators by defining the new operator set domain. In practice, at the time of writing this thesis there is no additional operator sets to `ai.onnx` and `ai.onnx.ml`.

For the machine learning library to support ONNX it must be able to export and import ONNX models. Exporting means creating the ONNX file from model in the library's native format to ONNX format. Importing implies to load and parse an ONNX file into library's native format and using model in the native format for inference. Currently, the libraries listed in Table 3.1 claim to have support for ONNX.

Unfortunately, there is no systematic research done on how well each of the libraries from Table 3.1 supports the ONNX and how fluent is the process of moving model from one library to the other. The metric to evaluate the level of support might be count of operators that are implemented or count of successful imports from the ONNX Model Zoo [33]. Most of the libraries and converters libraries presented in Table 3.1 are deep learning libraries.

The ONNX Model Zoo [33] is a collection of pretrained models available in ONNX format hosted on GitHub. Again, the emphasis is on state-of-the-art deep learning models related to image processing tasks like image classification. These models can be used for testing purposes or as a source of machine intelligence models.

The set of utility tools were developed around ONNX. One useful utility is `onnxmltools package` that provides converters for some of the libraries from Table 3.1. It provides converters from Apple Core ML, scikit-learn, Keras and LightDBM

Table 3.1: Libraries that claim to support ONNX format either internally or via converters. [32]

Neural Network Libraries	**Classical ML libraries**
Caffe2	Scikit-Learn
Chainer	LibSVM
Cognitive Toolkit	Apple ML Core
PyTorch	dmlc XGBoost
CoreML	
MATLAB	
SAS	
PaddlePaddle	
TensroFlow	
Keras	
NCNN	

to ONNX. The example provided on the right side of Figure 3.2 was created by converting scikit-learn linear regression model to ONNX. While using these converters, it must be understood that only limited subset of all libraries native capabilities are usually exportable to ONNX. Although the development is rapid and converters are being improved all the time.

Although an exporter for scikit-learn library exists there is no importer that is capable to import ONNX model to scikit-learn. The author was able to find only one software package that supports models created using `ai.onnx.ml` operator set. The software is Windows Machine Learning or WinML for short [34]. The code in `onnxmltools` package for converting scikit-learn models to ONNX format was actually contributed to the project by Microsoft.

WinML [34] is a runtime that provides an inference engine to evaluate ONNX models locally on the Windows device and allows Windows applications written in `C#`, `C++` or `JavaScript` to use ONNX models as embedded intelligence. This project has similar goals to the IL when it comes to serving models locally. Decoupling the machine intelligence from the application into a separate layer that is providing ML models locally on device. Since the project is still in its infancy the information about it is limited and the only reference the author is able to provide is a Web Page [34].

The approach that is taken in WinML is different to the one used in prototype presented in Chapter 4. In WinML, user has to deal with raw ONNX models directly by loading them from the code. In author's prototype implementation, the intelligence layer is supposed to provide functionality of ONNX models to the user via interprocess communication protocol. User might not be even aware that models are provisioned using ONNX format.

3.4.3 PMML

Predictive Model Markup Language is XML-based open standard for representing data mining models and statistical models. The data mining term in the context of IoT usually means predictive analytics or anomaly detection. The main goal of the language is to allow data scientist who design the models and software developers who write production code that utilizes the models, made by data scientists, to easily share models between each other and between different application and platforms, avoiding the incompatibility and proprietary issues that might arise when different tools and libraries are used to design and integrate models into products. [26]

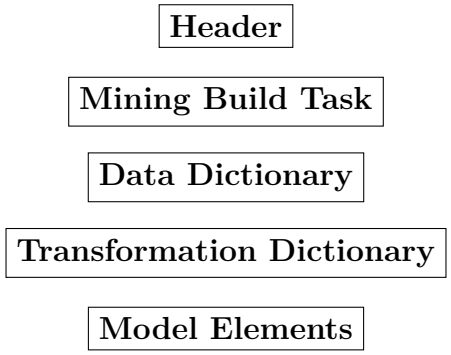

Figure 3.3: Structure of PMML model's XML document.

The XML file describing PMML model has a structure depicted in Figure 3.3. The structure of PMML file maps well to the typical machine learning inference problem as described in Chapter 2. The **header** describes meta-data related to the model: when the model was created and what tool was used to create it.

Mining build task is an optional element and is not required. It can contain any XML elements related to model creation or training. The information from this element is primarily used for visualization and maintenance purposes.

In **data dictionary**, fields used by the mining models and the types and ranges of those fields are defined. In PMML variables are called fields.

Transformation dictionary defines data transformation function that can be used in any model that is part of PMML document. The function described in transformation dictionary might represent calculation for feature extraction required by machine learning model. One example of such operation is normalization of data where the data is transformed to have zero mean and unity variance during the training process and naturally same transformation must be applied when model is used to make prediction on new data.

Next block in the structure of Figure 3.3 is a sequence of top level **model elements**. The PMML document can have 0 or more models, also called top-level models. Model elements describe the model. Each model is required to have an attribute describing what kind of machine learning it performs: classification, regression, clustering, time series analyzing, sequences analyzing, association rules and mixed. The model also has an optional model name element. The consumer of the model can select what model to use based on its name. If no name is specified,

the default behaviour is to select first model from the sequence.

The model has child elements that are common to all the models and some model specific elements. The common child elements are: mining schema, output, model statistics, local transformations, verification. Fields used by the model are defined in mining schema. Output element contains output field elements that define what kind of values are returned from the model. Model statistics provides statistics related to the fields of the model. Targets holds information on target values. Local transformation define transformation, similar to transformation dictionary, that are local to the model. Model verification elements define elements that can be used by the model consumer to check that results generated by the model are accurate regardless of the platform where the model is executed. Note that all elements except mining schema might not appear in PMML model description element. It is model dependent if they appear or not. The full list of supported models is provided in Table 3.2.

Table 3.2: Models that are supported by PMML format. [35]

Association Rule Model
Bayesian Network
Baseline
Clustering
Gaussian Process
General Regression
k-Nearest Neighbors
Naive Bayes
Neural Network
Regression
Ruleset Model
Scorecard Model
Sequences
Text Models
Time Series
Trees
Vector Machine

Formally the structure and elements of PMML document are defined by XSD (XML Schema Definition) document. The schema doesn't define PMML document completely. There are also field naming conventions that have to be followed in order for PMML document to be valid, like requirement for fields in data dictionary and transformation dictionary to be unique. The reason for this requirement is that fields defined in these elements have global scope, they are visible to all models. If two fields are named the same then a name collision happens, it will be impossible for application to known what field to use. A snippet from PMML definition is provide in Listing 3.6. The complete document description of the schema definition can be found at [36].

Listing 3.6: Snippet from XML Schema specification.

```
<xs:element name="PMML">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Header"/>
      <xs:element ref="MiningBuildTask" minOccurs="0"/>
      <xs:element ref="DataDictionary"/>
      <xs:element ref="TransformationDictionary" minOccurs="0"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:group ref="MODEL-ELEMENT"/>
      </xs:sequence>
      <xs:element ref="Extension" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="version" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

The PMML format was first introduced as part of data mining toolkit called Rattle (`R` Analytics Tool To Learn Easily). Later it was separated from Rattle into its own `R` package. [26] Hence `R` has best support for PMML. It is also possible to export some of the machine learning models from `scikit-learn` to PMML using `sklearn2pmml` `Python` package. Some converters exist for other ML libraries but they are usually one man prototype projects on `GitHub` lacking proper documentation and capability description. Among notable ML libraries only Apache Spark has official support for PMML but just for 5 models types. Major deep learning libraries frameworks like TensorFlow, MXNet or PyTorch do not support PMML. Data mining groups that is in charge of developing and extending PMML has recently, in 2015, introduced new standard called Predictive Format for Analytics (PFA) that will most probably replace PMML [37].

Listing 3.7: PMML document representing linear model for predicting closing prise of Ethereum cryptocurrency based on the price of Bitcoin cryptocurrency.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<PMML xmlns="http://www.dmg.org/PMML-4_3" xmlns:data="http://jpmml.org/jpmml-model/InlineTable" version="4.3">
        <Header>
                <Application name="JPMML-SkLearn" version="1.5.8"/>
                <Timestamp>2019-01-03T11:40:57Z</Timestamp>
        </Header>
        <MiningBuildTask>
                <Extension>PMMLPipeline(steps=[('data_transformer', StandardScaler(copy=True, with_mean=True, with_std=True)),
      ('model', LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
        normalize=False))])</Extension>
        </MiningBuildTask>
        <DataDictionary>
                <DataField name="y" optype="continuous" dataType="double"/>
                <DataField name="x1" optype="continuous" dataType="double"/>
        </DataDictionary>
        <TransformationDictionary>
                <DerivedField name="standard_scaler(x1)" optype="continuous" dataType="double">
                        <Apply function="/">
                                <Apply function="-">
                                        <FieldRef field="x1"/>
                                        <Constant dataType="double">2620.7572983006326</Constant>
                                </Apply>
                                <Constant dataType="double">3874.711285592038</Constant>
                        </Apply>
                </DerivedField>
        </TransformationDictionary>
        <RegressionModel functionName="regression">
                <MiningSchema>
                        <MiningField name="y" usageType="target"/>
                        <MiningField name="x1"/>
                </MiningSchema>
                <Output>
                        <OutputField name="predict(y)" optype="continuous" dataType="double" feature="predictedValue" isFinalResult="false"/>
                        <OutputField name="standard_scaler(predict(y))" optype="continuous" dataType="double" feature="transformedValue">
                                <Apply function="/">
                                        <Apply function="-">
                                                <FieldRef field="predict(y)"/>
                                                <Constant dataType="double">-0.5806909030705703</Constant>
                                        </Apply>
                                        <Constant dataType="double">0.003634611081731621</Constant>
                                </Apply>
                        </OutputField>
                </Output>
                <RegressionTable intercept="-1.6963320349816126E-16">
```

```
                <NumericPredictor name="standard_scaler(x1)" coefficient="0.9145197468684679"/>
            </RegressionTable>
        </RegressionModel>
</PMML>
```

To end this subsection on PMML, the author presents an example of the linear regression model with input data standardization written in `scikit-learn` and converted to PMML using `sklearn2pmml` package. The python code and PMML document are provided in Listings 3.8 and 3.7. In order to port the model from `scikit-learn` one must use `PMMLPipeline` class that is inherited class from `scikit-learn`'s `PipeLine` class defined in `sklearn2pmml` package. The PMML pipeline is converted to PMML XML document using `skleran2pmml` function.

Listing 3.8: Python code to convert linear regression model to PMML file presented in Listing 3.7.

```
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
import pandas as pd
from sklearn2pmml import PMMLPipeline
from sklearn2pmml import sklearn2pmml


# Loading and reshaping data
# Closing prices for Ethereum and Bitcoin are stored in separate CSV files
df_bc = pd.read_csv("BTC-USD.csv", parse_dates=['Date'])
df_eth = pd.read_csv("ETH-USD.csv", parse_dates=['Date'])

bcv = df_bc.Close.values.reshape(-1, 1)
etv = df_eth.Close.values.reshape(-1, 1)

scaler_for_output = StandardScaler().fit(etv)

scaler_for_prediction = StandardScaler()
scaler_for_prediction.mean_ = -1.0 * scaler_for_output.mean_ / ↩
    scaler_for_output.scale_
scaler_for_prediction.scale_ = 1 / scaler_for_output.scale_

pmml_pipe = PMMLPipeline(steps=[('data_transformer', StandardScaler()), ('↩
    model', linear_model.LinearRegression())],
                         predict_transformer=scaler_for_prediction)
# Training the model
pmml_pipe.fit(bcv, scaler_for_output.transform(etv).reshape(-1,))
print('Learned parameters')
print(pmml_pipe.steps[1][1].coef_)
print(pmml_pipe.steps[1][1].intercept_)

# Testing prediction
print(pmml_pipe.predict([[1.0]]))
print(pmml_pipe.predict_transform([[1.0]]))

# Exporting Model to PMML
sklearn2pmml(pmml_pipe, 'etherium_price_redict_model.xml', with_repr=True)
```

3.5 Intelligence Layer

Intelligence layer is a concept proposed in a paper [7] that came out off Ericsson research team in Finland. One central idea of intelligence layer is to decouple as much as possible the intelligence from application. Other important aspects are composition of intelligence services and life cycle management related to intelligence, such as intelligence setup and intelligence decomposition. But for this thesis, the decoupling aspects are more relevant.

Currently intelligence is implemented as part of the application layer tightly integrated into application's source code. To update the intelligence usually the whole application must be updated.

Another approach is to use intelligence as a cloud service. Whenever application needs some intelligent function like image classification or text translation it makes an HTTP request. The data required to do inference is sent to HTTP endpoint. HTTP endpoints sends back the result as an HTTP reply. In this way the intelligence is external to the application, but other difficulties arise:

- Intelligence is not owned by application.
- Sending data to cloud for evaluation and getting the result back might introduce latency and high bandwidth usage.
- Costs related to using cloud service and network bandwidth.
- Privacy issues related to sending data to cloud.
- Availability of connectivity at all time for continuous operation.

These difficulties can be avoided by having intelligence locally available, only connecting to cloud or Internet for updates or to get new intelligence. This concept of local availability is one of the key aspects of intelligence layer framework. IL platform captures many more other aspects of machine learning platform, interested reader can review all the concepts in [7].

4 Implementation

In this chapter a proof of concept implementation of the intelligence layer is presented. The intelligence layer is implemented as an application D-Bus service running on Ubuntu Linux operating system. D-Bus is Inter-Process Communication (IPC) system that is used mostly on Linux based operating systems. The intelligence layer service reads ONNX models from the disk and creates D-Bus object for each model. User applications can access these objects via interfaces defined by the IL service and call methods to execute the model and get back the results of inference. The concepts of D-Bus, like: service, object, interface, and how the protocol is used to create an intelligence layer service is explained in the next subsection.

To execute ONNX models representing neural networks the nGraph [38] software library is used. nGraph is used for converting neural networks represented with library specific format or library independent formats into nGraph's format and performing inference or training on the supported hardware backends. The hardware backend is a processing unit like GPU, CPU or some other computational hardware designed specifically for machine learning. For the prototype implementation, only inference capabilities of nGraph are used. In principle some of the nodes from `ai.onnx.ml` operator set is also possible to implement using nGraph's native set of operators, but it is left for future work.

4.1 Base Architecture

Diagram in Figure 4.1 represents the high-level architecture of intelligence layer service when it starts. During the initialization process IL reads ONNX model files from the special directory, each model is represented by single file with .onnx file extension. The directory is given to the program as a command line parameter. The process depicted in the diagram happens for every model found in the directory.

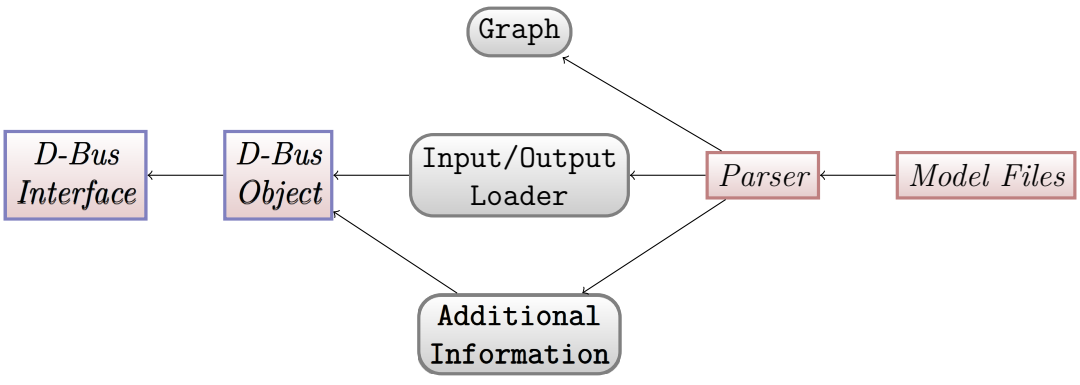

Figure 4.1: Architecture of IL service daemon.

ONNX model file is deserialized and parsed. Three pieces of information are crucial for creating an intelligent service out of deserialized ONNX model. The **graph** that describes the actual computations that must be performed on input data to produce a prediction or classification. The **input and output loaders** for

converting data supplied by the user into presentation that can be fed to graph and converters the do a reverse operation. That is, converting the result produced by executing the graph on converted input data, to the format that can be sent back to the user. Last piece of information is **additional information** like documentation provided with the model, version and other information that is not directly related to graph execution. These three pieces of information are represented in Diagram 4.1 by rounded rectangles with corresponding titles with arrows ending at the boxes and diverging from parser rectangle.

As depicted in the diagram additional information and input/output information is used to create a D-Bus object. Using terminology from Chapter 2 on intelligent agent, D-Bus object represents actuators and sensors of an intelligent agent. The intelligent agent can be identified with intelligent service that intelligent layer is providing for users to utilize. In theory, one could consider other intelligent service as a user of other intelligent service to achieve for example reinforcement learning. But this topic is outside the scope of this thesis.

4.1.1 D-Bus IPC system

D-Bus is a system for local IPC [39]. It is built on top of `AF_UNIX` sockets introducing interprocess communication concepts on the higher level of abstraction like buses, services, clients, objects, interfaces, methods, signals and properties:

- A message **bus** is a source of services and an application that allows other applications to communicate with each other. D-Bus system usually provides two buses: system bus and user bus. System bus is for system services and there is only one system bus on host Linux operating system. User bus is for user services and each user has his own bus. The IL service runs on the user bus.

- A **service** is a program that provides APIs to utilize the service on a bus. A service has a unique name specified in reverse domain notation. APIs for IL service are available at `fi.ericsson.nomadiclab.IntelligenceLayer`.

- A **client** is a user application that uses APIs provided by a service. The client of IL service can view what intelligence services are available, query additional information about selected service and provide service with input to get back the result.

- An **object** is identified by an object path and it belongs to a service. Service can have multiple objects that form directory tree like hierarchy. Object path looks like a file system path. For example, an object path that is responsible for IL service managment might have a path `/fi/ericsson/nomadiclab/intelligence_layer`. And for each intelligence service an object is created forming a list of objects:

  ```
  /fi/ericsson/nomadiclab/intelligence_layer
  /fi/ericsson/nomadiclab/intelligence_layer/models/image_classification
  /fi/ericsson/nomadiclab/intelligence_layer/models/face_recognition
  /fi/ericsson/nomadiclab/intelligence_layer/models/health_monitoring
  ```

- Each object has one or more **interfaces**. Interface is defined by its members and provides APIs to utilize the object. A member can be a method, a signal or a property. Interfaces are identified by reverse domain name notation. Interface for image classification model from previous example might be `fi.ericsson.nomadiclab.IntelligenceLayer.Classifier`

- A **method** of an interface is a function that can take inputs and produce an output. For example `Classifier` interface might have `DoPrediction` method to classify input data. A set of parameters that a method takes and returns is specified by method's **signature**. The signature is encoded as a series of characters. For `DoPrediction` method of `Classifier` interface, signature might be `s` for input and `as` for output, telling the user of the method that `DoPrediction` takes string as input and returns array of strings to the client that invoked the method. Input string can represent a link to a resource, for example a file system path to an image. Output might be an array of all categories sorted in descending order starting with category with the highest classification score.

- Methods are for one-to-one, request-response communication between a client and a service. A **signal** can be used for one-to-many, or also one-to-one, notifications. For example, IL service might send a signal to all the clients that a new model is introduced or that one of existing models was updated to a newer version. One can also imagine having intelligence service that do not require an input from the client and just sends the result of some computation based on data that it receives from the system.

- A **property** is just a variable. The type of property is defined with a signature using the same syntax as for methods. Interface for intelligence service object might have properties exposing meta information provided by ONNX model like documentation string of a model and a model version.

The users of IL service, that are clients, can introspect which objects are available under the model's path and use interfaces provided by those objects to call methods that execute the machine learning model and get back the result or examine properties that contain description of the model and some additional information about the model.

4.1.2 The Graph

The nGraph software library is used to extract the graph from ONNX model. nGraph allows us to convert graph that is part of ONNX model to the graph represented by nGraph's IR format, compile the converted graph and execute produced code on the hardware that nGraph supports.

nGraph can be used for training and inference [38]. Training means calculation of gradient with respect to neural network parameters. Four the prototype implementation presented in this thesis only inference capability is used.

In order to run inference code nGraph must be asked to allocate memory for input and output. Then, fill memory reserved for input with data and apply the graph to the input data. After the graph is applied to data, result can be read from reserved memory and sent to client.

The D-Bus method call used by client for invoking inference execution might have a signature specifying that client must supply raw data as a parameter to the method call. The raw data is suitable, if model provides a purely computational service. Array of numbers goes in and modified arrays of numbers is returned to client.

Intelligence layer is assumed to support higher level data types. For example, input data type might be a string, representing a link to a resource. Resource might be a file system path, URL or path to the device. The IL must have capability to extract raw data from the resource and write to memory allocated by nGraph for input.

Current implementation only supports strings pointing to resources on a file system. For example, a path to a location for image that client wants to classify.

4.1.3 Input and Output Loaders

In nGraph, computational graphs are represented by functions. Function takes as input a multidimensional array filled with numbers. The data represented by an array of numbers does not have any semantic meaning. To give semantic meaning to data ONNX has an experimental feature called denotations. A tensor in ONNX can have optional denotation represented by string literal. Intelligence layer uses this denotation to use correct loaders to extract raw data, pointed to by resources that are supplied by a calling client with a method call, and fill input memory of the nGraph's function.

After nGraph function is executed, the produced purely numerical data must be translated into a sensible reply to user. For that again, an output tensor denotation is used.

Denotations are also used to assign a correct interface to D-Bus object created for a model. For example, if input tensor is denotated as image and output tensor is denotated categorical then IL service adds classification interface to the model object it creates. If denotation for input and output tensor are missing function interface is created.

4.1.4 Other information

Client might be interested in a short description of a model or what kind of input model expects. This information can be conveniently presented as properties of D-Bus model object. The implementation of IL service in this work provides almost all descriptive model information available in `ModelProto` protocol buffer message:

- Version of the ONNX IR
- Tool used to produce model

- Version of the tool use to produce model
- Domain to which model belongs
- Name of the author if available
- Input Denotation
- Output Denotation

4.2 Software design

In order to deliver continuous operation and low latency the intelligent layer service must be implemented as a concurrent program. It is assumed that IoT device supports multithreading. In concurrent system different procedures are running on multiple threads of execution. These procedures need to communicate with each other and share different resources. An additional complication comes from using `C++` as implementation language that was chosen for implementing IL. `C++` allows to write efficient code with low overhead but it is regarded as a complex language, requiring a lot of low level knowledge to create concurrent programs.

One way to implement concurrent program is to use raw threads and synchronization mechanism like mutexes and conditional variables. Other approach, that was chosen for the prototype implementation, is to use actor model [40]. In actor model many system components can be represented as actors. Actors might be `C++` objects that run on the same or different threads. Actors communicate with each other by exchanging messages. Based on received messages actor react by sending new message or doing some computations. Actors don't share any state among each other, hence making race condition that are common source of bugs in concurrent programs less probable. All mechanism required to synchronize the message exchange is handled by framework providing the implementation of the actor model.

In the prototype implementation for this thesis, `SObjectizer C++` actor model framework is used. It provides enough features to implement intelligence layer service prototype. One restriction that `SObjectizer` has compared to some other actor frameworks is that it doesn't allow distributing actor among different computers scattered over computer network. But for the prototype implementation presented, this functionality is not required.

The more detailed diagram, based on the actor model, of architectural diagram presented in Figure 4.1 is presented in Figure 4.2. The diagram in Figure 4.2 presents four different types of actors: `model discoverer`, `model manager`, `dbusio` and `model executor`. Exactly one instance of actor exits for discoverer, manager and dbusio. Zero or more instances of executor actor are dynamically created by model manager when users make request to execute the model via dbusio actor. After executing the nGraph model function and returning the result to dbusio, model executors actors are destroyed.

Model discoverer scans the model repository periodically to detect changes. Model might be added, removed or updated. When model discoverer detects change to the

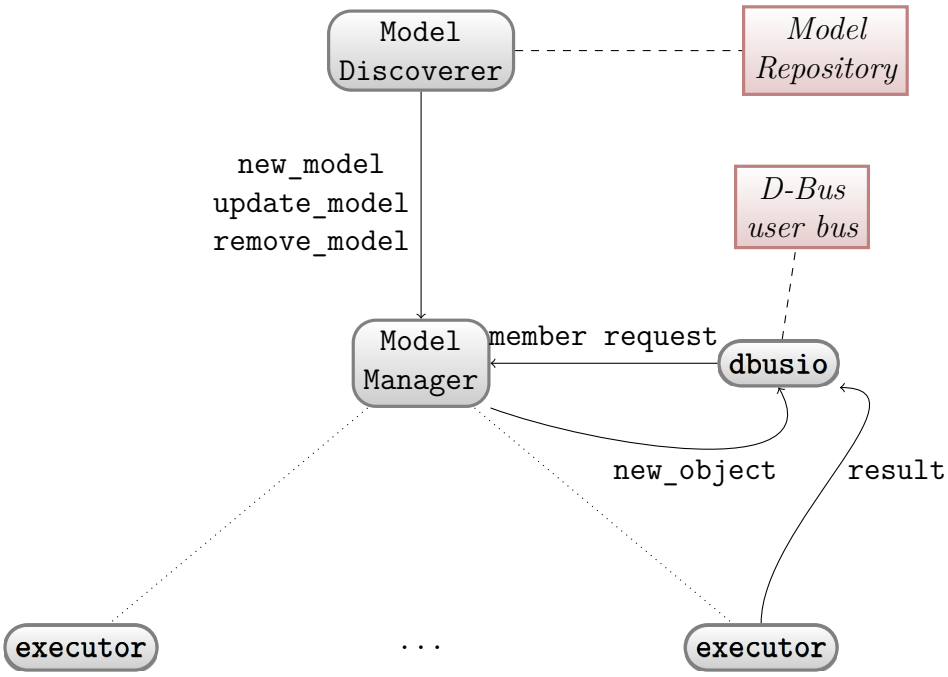

Figure 4.2: Actor model-based design of intelligence layer. Arrows represent message sending. Dotted lines represent creation of executor actors by model manager. Dashed lines represent access to external resource like file system and D-Bus user bus.

repository, it parses the ONNX model files if model was updated or added and sends appropriate message to model manager. As is presented in the Figure 4.2 discoverer can send three different messages:

- `new_model` to add new model to the model list of model manager.
- `update_model` to update existing model.
- `remove_model` to remove model from the list of model manger.

`new_model` and `update_model` messages contain all information required to execute the model: like model name, the types of input and output, nGraph function and other information related to model like documentation string. For removing the model all this information is not required and only name of model is attached to the message.

When the model manager receives a `new_model` messages, it adds model to its list of models and sends `dbusio` actor `new_object` message to create a D-Bus object for the model. `dbuisio` exposes the appropriate interface of the object to clients of IL service. The member invocations requested by clients are routed to model manager. These invocations are represented by `member request` in Figure 4.2 but they might be implemented by multiple different messages.

For the D-Bus object's property requests or method calls that do not require any computation model manager actor doesn't create executor actor. If method request requires execution of nGraph function then model manger creates executor actor

potentionaly on a different thread of execution so that model manger can still handle incomming messages while executor is doing computations. When executor is done, it sends `result` message to dbusio actor to relay the result of model execution in the reply to client that invoked the method.

5 Results and Conclusions

Previous chapter presented the implementation details of IL service. In this chapter, the test results and final conclusions of using IL service prototype implementation to serve image classification ONNX model are presented.

First the test ONNX model is described then tools and hardware used for testing the IL service are presented. The chapter ends with conclusions and feature work sections.

5.1 Experimental setup and model preparation

To test the implementation an ONNX model was needed. The place where one can find ready-made pre-trained ONNX models is called ONNX Model Zoo [33]. It is a GitHub repository with the models that have been developed with some framework, like MXNet, and then exported to ONNX format. Each model from the ONNX Model Zoo has a descriptive Web-page. The Web-page contains information about what framework was used to create the model, where the data that was used for training came from, and how pre-process and post-process the raw data required and produced by model execution process.

Version 2 of the ResNet [41] image classification neural network model was selected for testing IL service prototype implementation. The model takes image as input and outputs the class of the major object in the image. The model was trained on the ImageNet dataset which contains images from 1000 classes. The largest available model Resnet-152 was used. It is approximately 220 MB in size and is most accurate among other Resnet models with fewer layers.

Image classification is an important use case for IoT. One can image camera monitoring the production line for faulty products taking a snapshot image of a product and requesting a prediction from intelligence layer service to classify the product as good or bad.

The Resnet-152 model couldn't be used as it was provided in ModelZoo. It was assumed that pre-processing and post-processing procedures are done in code by the user who imports the model in his or her code. Model was just expecting an $1 \times 3 \times 224 \times 224$ dimensional tensor as input and produced 1000 dimensional tensor as output.

According to Resnet model description, input tensor should contain data of a color image and output tensor scores for each class. It is assumed that user of ResNet ONNX model will know enough information about the model to load an image in some specific format, for example PNG, extract pixel values for each channel using some libraries and normalize pixel values to range $[0, 1]$.

Similar assumptions were made for the output. It was user's job to download the file with all the names of categories and connect result output tensor with categories.

In order to fix these issues, the author had to add computational pre- and post-processing nodes to the model. Pre-processing nodes to do data standardization were added. Post-processing node to convert scores to probabilities with softmax function was added.

```
name: "input"
type {
  tensor_type {
    elem_type: FLOAT
    shape {
      dim {
        dim_value: 1
        denotation: "DATA_BATCH"
      }
      dim {
        dim_value: 3
        denotation: "DATA_CHANNEL"
      }
      dim {
        dim_value: 224
        denotation: "DATA_FEATURE"
      }
      dim {
        dim_value: 224
        denotation: "DATA_FEATURE"
      }
    }
  }
  denotation: "IMAGE"
}
```

```
key: "Image.BitmapPixelFormat"
value: "Rgb8"
,
key: "Image.ColorSpaceGamma"
value: "SRGB"
,
key: "Image.NominalPixelRange"
value: "Normalized_0_1"
```

Figure 5.1: Textual representation of input denotation for ResNet model's input. Left side is snippet that represents input of the graph. On the right side the key-value pairs of the `ModelProto` `meta_data` field are presented related to the `IMAGE` type.

In order for intelligence layer to know that model expects an image prepared in a particular way author had to denotate input tensor and all the dimensions of the input tensor. Denotating dimensions was not necessary because prototype implementation doesn't support them yet, but it was done for completeness. Denotating means giving semantic description to tensors, so that intelligence layer service or any other application that consumes ONNX model can understand the meaning of a tensor and use appropriate loader. The ONNX has an experimental proposal for detonating tensors that is not part of the official format yet [42]. For that purpose, `TypeProto` message has `denotation` field that is a string. Three types of denotation are available: `TENSOR`, `IMAGE`, `AUDIO`, `TEXT`. Out of these four, only `TENSOR` and `IMAGE` have a somewhat complete definitions. Tensor types do not require denotations. If denotation is missing then tensor denotation type is assumed.

`TypeProto` is wrapper message around one of three possible types: Tensor, Sequence or Map. The type used is indicated by `type` field of `TypeProto`. For neural

networks only Tensor type can be used. Tensor type and denotation type are two different things and should not mixed.

For tensor type it is also possible to denotate dimensions. `Tensor` protocol buffer message has shape field of type `TensorShapeProto` that is an array of `Dimension` messages that have `denotation` field. The definition for dimension denotations are not well formalized yet and only three denotation types are presented in ONNX proposal: `DATA_BATCH`, `DATA_CHANNEL`, `DATA_FEATURE`.

The denotation for type and dimensions of input tensor of the modified ResNet model are provided in Figure 5.1. As can be seen, model expects an `IMAGE` as an input. The format of the image is specified on the right side of Figure 5.1. Using this information intelligence layer will know that an image it suppose to pass to the model has a specific pixel format, color space and that it should normalize pixel values to range $[0, 1]$ by scaling data from each color channel by 255. Dimensions are also denotated, but current version for intelligence layer disregards them.

```
name: "reshape0"
type {
  tensor_type {
    elem_type: FLOAT
    shape {
      dim {
        dim_value: 1000
        denotation: "DATA_PROBABILITY"
      }
    }
  }
  denotation: "CATEGORICAL_PROBABILITIES"
}
```

```
key: "Category_0"
value: "tench, Tinca tinca"
,
key: "Category_1"
value: "goldfish, Carassius auratus"
,
.....
,
key: "Category_998"
value: "ear, spike, capitulum"
,
key: "Category_999"
value: "toilet tissue, toilet paper"
```

Figure 5.2: Textual representation of output denotation for ResNet model's output. Left side is snippet that represents input of the graph. On the right side the key-value pairs in the `meta_data` field are presented related to `CATEGORICAL_PROBABILITY` type.

The output tensor of the ResNet model represents probability distribution over 1000 different categories. ONNX doesn't have denotation to represent tensor with such semantic meaning. For that reason the author had to invent a new deno-

tation type. It was named `CATEGORICAL_PROBABILITY`. To denotate a tensor as `CATEGORICAL_PROBABILITY` one must set its denotation filed to `CATEGORICAL_PROBABILITY` and add all the categories as map from category identifiers to category names in `meta_data` of the `ModelProto` message. Textual representation of output tensor is presented in Figure 5.2

The model was prepared for consumption by intelligence layer using Jupyter Notebook. In the notebook, ONNX model was imported into MXNet and missing operators were added for pre- and post-precessing. After this, model was exported again to ONNX and then loaded again to denotate the input and output tensors. For that `onnx Python` package was used that is part of official ONNX release.

5.2 Testing Implementation

The IL service implementation was tested by making it consume and serve the modified Resnet-152 deep neural network image classification model described in the previous section. The serving was tested by using two D-Bus debugging and monitoring applications: `busctl` and `D-Feet`. `busctl` is command line tool and `D-Feet` is graphical application.

Figure 5.3 shows a `DoPrediction` method call of a `fi.ericsson.nomadiclab.IntelligenceLayer.Classifier` interface owned by `/ericsson/nomadiclab/intelligence_layer/resnet152v_copy1` object which is part of `fi.ericsson.nomadiclab.IntelligenceLayer` service. The method call takes string as input and produces string as output. The input string points to the resource that in the case of Resnet is a path to the JPG image of a cat. The output string contains predicted class. As can be seen from the Figure, intelligence layer returns correct result.

Figure 5.3: Snippet from command line terminal showing the use of `busctl` and the picture of cat that was used for testing.

D-Feet allows to list and introspect D-Bus services via the graphical user interface. It can also be used to call methods of D-Bus objects and measure the time of a method call. Figure 5.4 shows the invocation of a DoPrediction method on a Resnet-152v2 D-Bus object with the same image from the previous example. As can be seen from Figure 5.4 the method call time is 0.6 seconds.

The tests were executed on a commodity computing hardware. The testing machine had the following specifications:

Ubuntu
Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz, 3501 Mhz, 4 Core(s)
16 GB RAM

The test results proof the feasibility of using ONNX format for provisioning machine learning models to device and serving them locally. Intelligent layer service is different from other model serving solution which are usually HTTP end-points and deployed in central cloud.

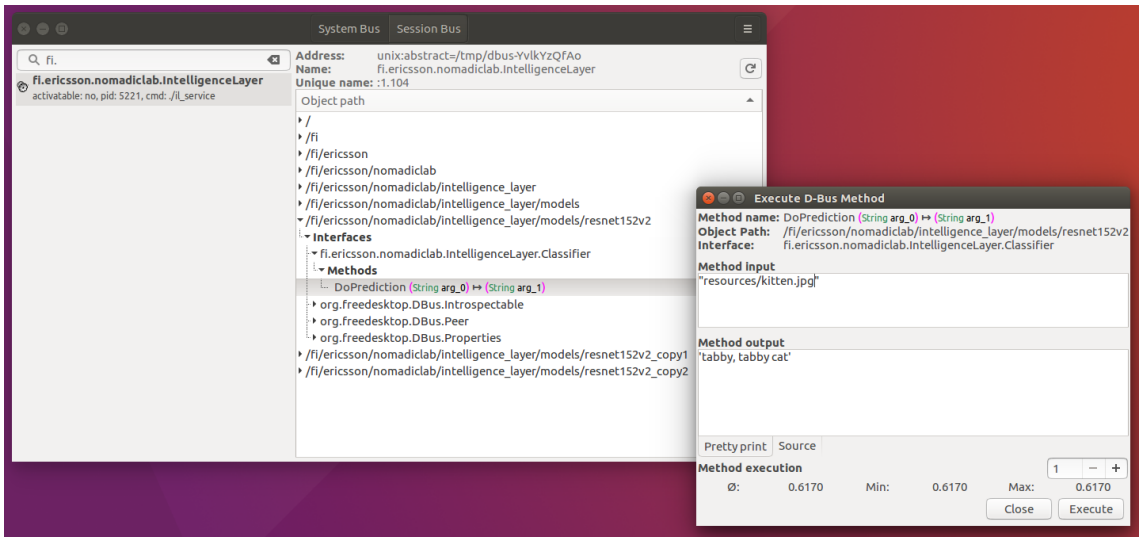

Figure 5.4: Using IL service with D-Feet.

Preparing ML model for such cloud-oriented ML model serving application usually requires writing a program with the code to load a ML model, and code to transform input and output data. In the approach proposed in this thesis, models are complete and all the complexity for IO transformation and API serving is hidden behind intelligence layer as long as creator of a model denotates the input and output tensors correctly.

5.3 Conclusions

It is clear that ONNX at the moment is an appropriate and popular format for representing neural network machine learning models intended for inference purposes. Many state-of-the-art deep learning frameworks support ONNX to some extent and new compilers and runtimes capable of executing ONNX NN models on variety of compute hardware, ranging from general purpose hardware like CPUs and GPUs to special purpose like Intel's Nervana [38] chip designed to accelerate NN computations, emerge all the time.

The list, available at ONNX's Web-page [32], of frameworks, run-times, compilers that ONNX team claims to support ONNX format is getting updated frequently. One recent development is the release of ONNX Runtime as an open source project by Microsoft. ONNX Runtime is a software library for executing ONNX models. In the Microsoft's blog post [43] accompanying the release of the project it is claimed that ONNX Runtime has full support for ONNX version 1.2 and later, including `ai.onnx.ml` set of operators. ONNX Runtime is first runtime to support both sets of operators `ai.onnx` and `ai.onnx.ml`.

ONNX Runtime would have been better solution compared to nGraph as a model executor for the implementation that was presented in Chapter 4. But the ONNX Runtime was released after the decision has been made to use nGraph for the implementation and for that reason it was decided to continue with nGraph.

Comparing the lists of available operators from `ai.onnx.ml` domain in Appendix A to the list of models in Table 2.1 one can see that lists have only four common models: linear regression, SVM, decision trees and bagging. Bagging can be considered and ensemble of trees. Most of the other operators of `ai.onnx.ml` domain are for data pre-processing and data post-processing. On the other hand, it must be kept in mind that, in theory, the operators from different domains can be mixed, if model executor supports them both. Using operators from the default domain `ai.onnx`, it is possible to express any computations that do not require back-loops. So it can be hypothesized with high probability that most of the operators from Table 2.1 are implementable in ONNX, for example if training is not considered, PCA from section 2.4 is just a matrix multiplication operation. Even if some operation is not implementable with standard set of operators, ONNX can be extended with custom domains of operators and then it will depend on the executor what operators it can and can't run.

On the other hand, if we look at Table 3.2 presenting models that are supported by PMML we can see that it is more compatible with Table 2.1. One might think that for that reason PMML is better format for representing machine learning models. But it must be taken into account that PMML has been around since 1999, when version 1 of the format was released, and until these days PMML is not widely adopted especially from the tools used to create and execute neural networks. The protocol buffer format used for ONNX is superior to XML used in PMML in terms of size of serialized models and parsing speed. For that reasons, the author thinks that ONNX is a better choice for representing also classical machine learning models and not just neural networks.

The author of this thesis does realize that more systematic comparison of PMML and ONNX must be conducted in order to decide which format is better, but it would necessarily take more space and resource that are not available and hence this study is left for feature work. Right now, it looks clearly that industry tends to prefer ONNX over PMML with big corporations like Microsoft and Facebook driving development of ONNX.

5.4 Future Work

There are many things that can be improved in the prototype. Adding more denotation types and testing more ONNX models. Allowing to use different executors to execute ONNX models. Right now IL service uses nGraph and only supports neural networks models. Next logical step would be to add ONNX Runtime support.

IL service was not tested on real IoT hardware. One good candidate for testing is Intel's IoT board with Intel atom processor.

References

[1] K.-H. Tan and B. P. Lim, "The artificial intelligence renaissance: deep learning and the road to human-level machine intelligence," *APSIPA Transactions on Signal and Information Processing*, vol. 7, p. e6, 2018.

[2] S. G. of the Artificial Intelligence Programme, "Finland's age of artificial intelligence turning finland into a leading country in the application of artificial intelligence. objective and recommendations for measures," tech. rep., Ministry of Economic Affairs and Employment, 2017.

[3] L. Atzori, A. Iera, and G. Morabito, "Understanding the internet of things: definition, potentials, and societal role of a fast evolving paradigm," *Ad Hoc Networks*, vol. 56, pp. 122–140, 2017.

[4] S. Wang, J. Wan, D. Li, and C. Zhang, "Implementing smart factory of industrie 4.0: an outlook," *International Journal of Distributed Sensor Networks*, vol. 12, no. 1, p. 3159805, 2016.

[5] J. M. Perkel, "The internet of things comes to the lab," *Nature News*, vol. 542, no. 7639, p. 125, 2017.

[6] M. Stolpe, "The internet of things: Opportunities and challenges for distributed data analysis," *ACM SIGKDD Explorations Newsletter*, vol. 18, no. 1, pp. 15–34, 2016.

[7] E. Ramos, R. Morabito, and J.-P. Kainulainen, "Distributing intelligence to the edge and beyond," 11 2018.

[8] A. Jung, "Machine learning: Basic principles," *CoRR*, vol. abs/1805.05052, p. 4, 2018.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[10] A. Darwiche, "Human-level intelligence or animal-like abilities?," *CoRR*, vol. abs/1707.04327, 2017.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, p. 354, oct 2017.

[13] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[14] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Series in Artificial Intelligence, Upper Saddle River, NJ: Prentice Hall, third ed., 2010.

[15] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *Knowledge Engineering Review*, 1994. Submitted to Revised.

[16] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, "Machine learning for internet of things data analysis: A survey," *Digital Communications and Networks*, 2017.

[17] C. H. Lampert *et al.*, "Kernel methods in computer vision," *Foundations and Trends® in Computer Graphics and Vision*, vol. 4, no. 3, pp. 193–285, 2009.

[18] Z. John Lu, "The elements of statistical learning: data mining, inference, and prediction," *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, vol. 173, no. 3, pp. 693–694, 2010.

[19] S. S. Haykin, *Neural networks and learning machines*. Upper Saddle River, NJ: Pearson Education, third ed., 2009.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016.

[21] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[22] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[23] L. Wang, "Owl: A general-purpose numerical library in ocaml," *arXiv preprint arXiv:1707.09616*, 2017.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[25] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pp. 620–629, IEEE, 2018.

[26] A. Guazzelli, M. Zeller, W.-C. Lin, G. Williams, *et al.*, "Pmml: An open standard for sharing models," *The R Journal*, vol. 1, no. 1, pp. 60–65, 2009.

[27] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, *et al.*, "Api design for machine learning software: experiences from the scikit-learn project," *arXiv preprint arXiv:1309.0238*, 2013.

[28] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml) 1.0," 2008.

[29] G. Kaur and M. M. Fuad, "An evaluation of protocol buffer," in *IEEE SoutheastCon 2010 (SoutheastCon), Proceedings of the*, pp. 459–462, IEEE, 2010.

[30] "Protocol Buffers Reference." https://developers.google.com/protocol-buffers/docs/reference/overview. Accessed: 2019-01-22.

[31] "ONNX GitHub page." https://github.com/onnx/onnx/tree/master/onnx. Accessed: 2019-01-02.

[32] "ONNX Tools." https://onnx.ai/supported-tools. Accessed: 2019-01-02.

[33] "ONNX Model Zoo." https://github.com/onnx/models. Accessed: 2019-01-02.

[34] "Windows Machine Learning." https://docs.microsoft.com/en-us/windows/ai/. Accessed: 2019-01-02.

[35] "Pmml online specification." http://dmg.org/pmml/v4-3/GeneralStructure.html. Accessed: 2019-01-03.

[36] "XML Schema for PMML specification." http://dmg.org/pmml/v4-3/pmml-4-3.xsd. Accessed: 2019-01-03.

[37] "Portable Format for Analytics." http://dmg.org/pfa/index.html. Accessed: 2019-01-03.

[38] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, *et al.*, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," *arXiv preprint arXiv:1801.08058*, 2018.

[39] "D-Bus Specification." https://dbus.freedesktop.org/doc/dbus-specification.html. Accessed: 2019-01-21.

[40] G. A. Agha, "Actors: A model of concurrent computation in distributed systems.," tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[42] "ONNX Typedenotation proposal." https://github.com/onnx/onnx/blob/master/docs/TypeDenotation.md. Accessed: 2019-01-10.

[43] "Official release of ONNX Runtime." https://azure.microsoft.com/en-us/blog/onnx-runtime-is-now-open-source/. Accessed: 2019-01-09.

A Lists of ONNX Operators

In this appendix the list of ONNX operators from both standard and classical operator sets are provided. First listing has the operators from `ai.onnx` set. Second listing shows operators from `ai.onnx.ml` set.

```
ai.onnx (default)
Abs
Acos
Acosh
Add
And
ArgMax
ArgMin
Asin
Asinh
Atan
Atanh
AveragePool
BatchNormalization
Cast
Ceil
Clip
Compress
Concat
Constant
ConstantOfShape
Conv
ConvTranspose
Cos
Cosh
DepthToSpace
Div
Dropout
Elu
Equal
Erf
Exp
Expand
EyeLike
Flatten
Floor
GRU
Gather
Gemm
GlobalAveragePool
GlobalLpPool
GlobalMaxPool
Greater
HardSigmoid
Hardmax
Identity
If
InstanceNormalization
```

```
IsNaN
LRN
LSTM
LeakyRelu
Less
Log
LogSoftmax
Loop
LpNormalization
LpPool
MatMul
Max
MaxPool
MaxRoiPool
MaxUnpool
Mean
Min
Mul
Multinomial
Neg
NonZero
Not
OneHot
Or
PRelu
Pad
Pow
RNN
RandomNormal
RandomNormalLike
RandomUniform
RandomUniformLike
Reciprocal
ReduceL1
ReduceL2
ReduceLogSum
ReduceLogSumExp
ReduceMax
ReduceMean
ReduceMin
ReduceProd
ReduceSum
ReduceSumSquare
Relu
Reshape
Scan
Scatter
Selu
Shape
Shrink
Sigmoid
Sign
Sin
Sinh
```

```
Size
Slice
Softmax
Softplus
Softsign
SpaceToDepth
Split
Sqrt
Squeeze
Sub
Sum
Tan
Tanh
TfIdfVectorizer
Tile
TopK
Transpose
Unsqueeze
Upsample
Where
Xor
experimental ATen
experimental Affine
experimental Crop
experimental DynamicSlice
experimental GRUUnit
experimental GivenTensorFill
experimental ImageScaler
experimental ParametricSoftplus
experimental Scale
experimental ScaledTanh
experimental ThresholdedRelu
```

```
ai.onnx.ml
ai.onnx.ml.ArrayFeatureExtractor
ai.onnx.ml.Binarizer
ai.onnx.ml.CastMap
ai.onnx.ml.CategoryMapper
ai.onnx.ml.DictVectorizer
ai.onnx.ml.FeatureVectorizer
ai.onnx.ml.Imputer
ai.onnx.ml.LabelEncoder
ai.onnx.ml.LinearClassifier
ai.onnx.ml.LinearRegressor
ai.onnx.ml.Normalizer
ai.onnx.ml.OneHotEncoder
ai.onnx.ml.SVMClassifier
ai.onnx.ml.SVMRegressor
ai.onnx.ml.Scaler
ai.onnx.ml.TreeEnsembleClassifier
ai.onnx.ml.TreeEnsembleRegressor
ai.onnx.ml.ZipMap
```