UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA MECÁNICA

DEEP LEARNING HUB:
A NOVEL GRAPHICAL PROGRAMMING PLATFORM FOR BIG MACHINERY DATA ANALYTICS IN RELIABILITY ENGINEERING & PREDICTIVE MAINTENANCE

TESIS PARA OPTAR AL GRADO DE MAGISTER EN
CIENCIAS DE LA INGENIERÍA MENCIÓN MECÁNICA

MOHAMMAD HOSSEIN PISHAHANG

PROFESOR GUÍA
ENRIQUE LÓPEZ DROGUETT

MIEMBROS DE LA COMISIÓN
VIVIANA MERUANE NARANJO
RODRIGO PASCUAL JIMÉNEZ

SANTIAGO DE CHILE
2020

RESUMEN DE LA TESIS PARA OPTAR AL GRADO
DE: MAGISTER EN CIENCIAS DE LA INGENIERÍA,
MENCIÓN MECÁNICA
POR: MOHAMMAD HOSSEIN PISHAHANG
FECHA: 2020
PROF. GUÍA: ENRIQUE LÓPEZ DROGUETT

DEEP LEARNING HUB:
A NOVEL GRAPHICAL PROGRAMMING PLATFORM FOR BIG MACHINERY DATA ANALYTICS IN RELIABILITY ENGINEERING

The global trend towards Industry 4.0 and Industrial Internet of Things (IoT), besides the proliferation of inexpensive sensing technology, has pushed Physical Asset Management landscape more than ever to employ online big machinery data analytics. As a result, the customers are not only requiring their physical assets investment be reliable but are also requiring their assets diagnose and prognose faults and alert their maintenance staff when components need to be replaced.

The advances in prognostics and health management made Deep Learning an inevitable part of the reliability, predictive maintenance and digital twin areas. Although the development of integrated and mature frameworks like Tensorflow and Keras made the implementation and deployment of Deep Learning based fault diagnosis and prognosis models less challenging than ever, “coding” still remains the main barrier for many non-computer-scientists in the reliability and maintenance communities.

In order to tackle this challenge, here we introduce Deep Learning Hub (DLHub): an open access and web-based graphical programming environment for developing Deep Learning solutions in general, and for reliability and maintenance areas in particular. DLHub, implementing state-of-the-art models in an intuitive drag-and-drop manner, not only offers a fast prototyping and code-free platform by helping practitioners to focus on the concepts and problem solutions rather than the syntax, but also makes the opportunity to guide the users toward error-free programming.

Another contribution of DLHub is the standardization on the structure of Deep Learning based Predictive Maintenance and Reliability projects by offering a comprehensive path from preparing datasets all through the predictions and necessary assessments, besides the standard auto generated Tensorflow code for each step. The flexibility and usefulness of DLHub in the smart reliability and predictive maintenance is demonstrated by means of a case study on a highly sensorized offshore natural gas compressor.

RESUMEN DE LA TESIS PARA OPTAR AL GRADO
DE: MAGISTER EN CIENCIAS DE LA INGENIERÍA,
MENCIÓN MECÁNICA
POR: MOHAMMAD HOSSEIN PISHAHANG
FECHA: 2020
PROF. GUÍA: ENRIQUE LÓPEZ DROGUETT

DEEP LEARNING HUB: UNA NUEVA PLATAFORMA DE PROGRAMACIÓN GRÁFICA PARA ANÁLISIS DE CONFIABILIDAD EN BASE A BIG DATA

La tendencia global hacia la Industria 4.0 y el Internet de las cosas industrial (IoT), además de la proliferación de tecnología de detección de bajo costo, ha empujado al panorama de la Gestión de Activos Físicos más que nunca a emplear análisis big data y en línea de activos físicos. Como resultado, los clientes no solo exigen que su inversión en activos físicos sea confiable, sino que también exigen que sus activos diagnostiquen y pronostiquen fallas y alerten a su personal de mantenimiento cuando sea necesario reemplazar componentes.

Los avances en el pronóstico y la gestión de la salud hicieron el Deep Learning una parte inevitable de las áreas de confiabilidad, mantenimiento predictivo y digital twin. Aunque el desarrollo de marcos maduros e integrados como Tensorflow y Keras hizo que la implementación y el despliegue de modelos de diagnóstico y pronóstico de fallas basados en Deep Learning fueran menos desafiantes que nunca, la "codificación" sigue siendo la principal barrera para muchos en las comunidades de confiabilidad y mantenimiento.

Para abordar este desafío, aquí presentamos Deep Learning Hub (DLHub): un marco de referencia de programación gráfica de acceso abierto y basado en la web para desarrollar soluciones de Deep Learning en general, y para áreas de confiabilidad y mantenimiento en particular. DLHub, que implementa modelos de vanguardia de una manera intuitiva de arrastrar y soltar bloques, no solo ofrece una plataforma rápida de creación de prototipos y sin código al ayudar a los profesionales a centrarse en los conceptos y las soluciones de problemas en lugar de la sintaxis y programación, sino también entrega la oportunidad de guiar a los usuarios hacia una programación visual sin errores.

Otra contribución de DLHub es la estandarización de la estructura de proyectos de confiabilidad y mantenimiento predictivo basados en aprendizaje profundo al ofrecer una ruta integrada desde la preparación de conjuntos de datos hasta las predicciones y evaluaciones necesarias, además del código estándar de Tensorflow generado automáticamente para cada paso. La flexibilidad y utilidad de DLHub en la confiabilidad inteligente y el mantenimiento predictivo se demuestra mediante un caso de estudio involucrando un compresor de gas natural altamente sensorizado en plaforma offshore de producción de petroleo y gas.

to my parents…

Acknowledgment

I would like to express my special thanks of gratitude to my supervisor, Professor Enrique López Droguett, who changed significantly my interpretation of engineering. Working in his research team not only gave me the opportunity to collaborate with several hard-working students and engineers in a highly international atmosphere, but also helped me to improve my professional habits and attitude.

I also would like to thank Professor Viviana Meruane and Professor Rodrigo Pascual as the committee members, for the time they spent on reviewing the thesis and their priceless feedback to improve the quality of the work.

Many classmates and colleagues in the Department of Mechanical Engineering, and the Smart Reliability and Maintenance Integration Laboratory taught me a lot during my master's study. Especially, the collaboration with Sergio Cofré on preprocessing of the data used in the case study of the thesis was so enlightening for me. I hope I have the opportunity to work with them in the future.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Area

The complex nature of all modern industrial processes has made asset management as one of the main levels of decision making in any industry. The ISO 55000 series, as the main international standard in this area, defines asset as “an item, thing or entity that has potential or actual value to an organization” [1]. This definition includes a wide variety of asset types like physical, financial, human, information, and intangible ones. Based on this definition, asset management is any systematic approach which governs and realizes value from the assets.

Physical asset management is particularly applicable to businesses and industries that involve lands, plants, machinery, or generally any manufacturing, transportation, energy or water infrastructures. It tries to make sure that the physical assets work fine, are safe, and satisfy the business needs. It provides many advantages like an increase in the production and human safety and a decrease in different types of risk. Finally, all such consequences in economic terminology could be translated to reducing the costs and increasing the gains.

A successful physical asset management system includes a complete spectrum of activities in different levels and hierarchies, from very general management and financial considerations to more technical and specific issues. Dependability assessment plays a crucial role in this system, and it could be defined briefly as estimating how much an asset is available, reliable, and maintainable. Most often reliability engineers are the ones who focus on these aspects.

With the advent of the Industry 4.0, Industrial Internet of Things (IoT), the proliferation of inexpensive sensing technology and the advances in prognostics and health management, customers are not only requiring their physical assets investment be reliable, but are also requiring their assets diagnose and prognose faults and alert their maintenance staff when components need to be replaced.

Reliability engineering inherently is a data intensive procedure because it needs a continuous monitoring of the asset conditions or operational states. Therefore, it is heavily mixed up with technologies to acquire, collect, store and process a big amount of data. These assets often have substantial sensor systems capable of generating millions of data points a minute. Although this “big data” needs complicated modern algorithms for the analysis, it often results in a solid and reliable basis for the subsequent decisions to be made.

Artificial Intelligence (AI) in general, and Deep Learning (DL) in particular, arises as a powerful approach for smart, autonomous and on-line early fault detection and prognostics of remaining useful life as it provides tools for processing and analyzing Big Machinery Data, i.e., massive and multidimensional sensor data. The application of these tools in physical asset management and reliability engineering is very well developed in academia, but many industrial sectors are still a few steps behind to adopt them in their real-world projects.

Machine learning and deep learning cover a whole spectrum of algorithms and methodologies complementing each other in different applications. In physical asset management there are several levels of reliability assessment with distinct objectives, importance, and complexities. The following paragraphs present a brief list of methods that could be used in each level of prognostics and health management (PHM). The important point is that each problem could be solved by various equivalent methods, while each one works better on some specific input data types or for some specific purposes.

Anomaly detection is one the first algorithms to apply in diagnostics, where the idea is simply to discriminate anormal (faulty) and normal data samples. Although usually traditional machine learning methods like isolation forests [2] or random cut forests [3] are used in this step, recently, deep learning models are also employed as robust anomaly detectors especially in maintenance and condition monitoring [4]. Similar to many other areas, advanced non-supervised models like autoencoders and generative models also have been tested successfully for anomaly detection purposes in structural health monitoring [5].

Preventive maintenance is not limited to anomaly detection. As usually multidimensional machinery data is used in reliability engineering, dimensionality reduction algorithms play a key role in fault diagnosis. A good example is shown in reference [6] using variational autoencoders. On the other hand, damage recognition and identification has gotten quite mature as well. Semi-supervised generative adversarial networks now have proved applications in fault detection [7]. Moreover, computer vision technologies are used in fault type identification [8], and damage localization [9]. Finally, reinforcement learning has been used to select the optimum preventive maintenance policies in various cases [10], [11].

Predictive maintenance mainly focuses on estimating the remaining useful life (RUL) of the equipment. A comparison between vanilla artificial neural networks and recurrent neural networks with the traditional machine learning methods obviously proves the high potential of deep learning in prognosis [12]. Deep learning-based data fusion algorithms [13] and Capsule networks also have been used for remaining useful life estimation [14]. The huge consequences of deep learning methodologies in increasing safety and reliability and decreasing the costs motivate the vibrant community to adopt cutting-edge technologies in this area.

Having all these powerful models in reliability and predictive maintenance, encourages the experts in this area to use deep learning tremendously. However, implementing advanced DL models requires a good knowledge of computer programming and a variety of frameworks and libraries in data analysis, as well as additional skills to manage the codes in different environments and operating systems and solving complicated installation issues and version conflicts between the tools. All these steps keep the engineers away from their principal goals and engineering challenges like increasing productivity and safety and decreasing costs and risks. Here, some difficulties and obstacles that engineers may encounter using deep learning are listed:

- Deep learning is experiencing a steep growth and progress in the concepts, techniques, and applications by a vibrant community mainly in the area of computer science and engineering. This growth rate is usually much more than the capacity of many industrial sectors to keep themselves updated and to take advantage of the latest achievements in this area.

- Tens of powerful software solutions have been developed in areas such as computer vision, autonomous cars and medicine, which also can be used partially in reliability engineering and predictive maintenance. All these solutions are optimized for their own specific problems and cases. Consequently, they do not cover many reliability problems.
- Most of the new algorithms in deep learning have standard implementations in frameworks like Tensorflow [15], Keras [16] and PyTorch [17]. These frameworks are the most common and powerful tools and environments the data scientists use. Therefore, a good level in Python programming language [18] is a must. Although Python is a simple, high level and straight forward programming language to learn, getting a good level is a serious and time-consuming investment, especially for engineers out of the academia.
- For most of the practitioners, making the model prototypes via coding takes a long time, because the user is responsible to implement all details of the model, while many of such details are repetitive in various cases. On the other hand, usually finding a good neural network architecture is a try-and-error procedure. The general building blocks of the network could be determined through the problems' logics, but still finding an optimum architecture involves testing several arrangements of layers.
- The implementation of deep learning models via coding is highly prone to errors, especially when only a few team members are responsible for coding, and no one else revises and tests the codes. These errors could occur in the implementation of the models or be result of misunderstandings in fundamental concepts.
- Finally, one of the main challenges in interdisciplinary teams is an efficient communication terminology about the highly technical topics. Code scripts are not always the best way of communication about the AI models, especially while reliability engineers present their deep learning models to managers.

This research tries to address the abovementioned challenges and to find an efficient solution to them. Therefore, a tool is needed that gives reliability engineers the opportunity to adopt deep learning through a gentle learning curve, while they can put their focus on the technical and industrial considerations versus the data processing parts. An ideal solution must be:

- Accessible everywhere, on any machine and a wide range of operating systems;
- Easy-to-use with as less software dependencies as possible;
- Intuitive for users with a low level of programming skills;
- Progressive to improve to newer generations of algorithms and models;
- Extendable for users who want to dive deeper in the models;
- Safe to meet the data privacy constraints of users and industries.

The solution presented in this research to bridge the gap between pure deep learning and PHM is Deep Learning Hub (DLHub), an open access web-based visual programming framework that can be used by reliability and maintenance engineers for a fast code-free prototyping of their deep learning problems. Using DLHub, most of the complexities in managing codes, libraries and frameworks are done automatically and the implementation of complex deep models is possible through a visual and intuitive manner. Thus, engineers can keep their focus on their main tasks especially when the projects have tight timing schedules and cost constrains.

1.2 General Objective

Develop and implement a visual programming framework that lets reliability and maintenance engineers to prototype deep learning models for their prognostics and health management projects in a fast, intuitive, and code-free manner.

1.3 Specific Objectives

1. Develop the front-end user interface in HTML (Hyper-Text Markup Language), CSS (Cascading Style Sheets), and JavaScript.
2. Design a standard workflow for deep learning projects in PHM.
3. Develop a graphical programming module in JavaScript specifically for making data-flow diagrams.
4. Divide the dataset preparation process into separate blocks. Implement the blocks, their parameter tuning panes, and their automatic Python code generation functionality in JavaScript.
5. Implement separate blocks for each individual standard artificial neural network layer defined in Keras and Tensorflow 2.0.
6. Prepare a customized Jupyter Kernel Gateway for simpler installation and usage.
7. Adopt necessary Jupyter functionalities to assemble the online front-end and the local Python environment using HTTP and WebSocket protocols.
8. Set-up a Google Firebase project, and mount the application on it, to manage the app hosting, user authentication, and a cloud database for storing the users' projects.
9. Demonstrate the usefulness of the proposed platform in a case study involving fault diagnostics of a compressor in a vapor recovery unit of an offshore natural gas platform.

1.4 Thesis Structure

The remaining of the thesis is organized in 6 chapters. Chapter 2 reviews the fundamental topics and provides necessary definitions for the following chapters. Chapter 3 provides the reader with the methodology used to solve the problem. Chapter 4 introduces Deep Learning Hub (DLHub) software as the proposed solution. A simple classification example on a very standard publicly available dataset is employed to shows how to implement a diagnostic model through a vanilla neural network in DLHub. Chapter 5 shows how to implement more advanced and complicated deep learning architectures in DLHub. It brings examples of diagnosis, through Convolutional Neural Networks (CNNs) and Autoencoders (AEs), as well as examples of generative models like Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). Chapter 6 provides a real-worlds case study on fault diagnosis. The compressors in two vapor recovery units in an offshore petroleum industry sector have been monitored with more than 170 sensors. This chapter reviews all the necessary preprocessing steps on the data and presents the implementation of anomaly detection tasks on the data in DLHub. Finally, chapter 7 summarizes the thesis by some discussion and conclusions and suggests some future works.

Chapter 2

Theoretical Background

The most important component in any deep learning, or generally any AI, study is the data. A long list of requirements, considerations, and infrastructures are needed for a successful data-driven project. This list may start with data acquisition equipment, followed by transfer protocols, data security and privacy, cloud storage, data querying, preprocessing, and finally data modelling and inference. The major focus of this thesis is on the last blocks of this chain, as its main objective is to develop a new methodology for data modelling in prognostics and health management. The proposed platform supposes that the users already have access to the preprocessed data. This chapter briefly presents definitions, working steps, and necessary concepts for the rest of the thesis.

2.1 Dataset Structure

Any data processing project starts with raw data collection, which usually is stored in one or several computer files. The raw data could be generated or gathered in different ways. In social sciences it may be gathered via questionnaires filled by a group of participants. In social media it may be the registered activities and traces the users have left behind. In medicine raw data is usually originated from test results for many patients. In computer vision or multimedia technology data is usually recorded by cameras and voice recorders; and in many industries, data is the different parameters of the machinery recorded, registered or generated by a substantial sensor system installed on the equipment.

Data is a collection of data instances, which also often called observations or data samples. Each instance usually, but not always, consists of two main parts: features and targets. Features are the attributes and their values and make the main body of the data instance to be processed. Targets are the labels, states or the values the features correspond to. Commonly, a data processing model tries to learn how to predict targets for new set of unseen features. For example, in a fault detection system, a model receives and processes a set of equipment features and tries to predict if the machine is working in healthy or damaged states. Other models may try to predict the damage type, or the damage level from the same set of features.

Data can be stored in very different manners, each one with a certain set of advantages, disadvantages, and use cases. In a big picture, data collections could be classified as structured versus document-based ones. Structured dataset is organized in tables, where the rows correspond to data instances or observations, and the columns are features. Targets, if they exist, usually are stored in separate columns with the same number of rows. Tables in structured collections often must follow a predefined schema, where the data type and other constraints for each column (feature) are determined. The constraints may contain the format the feature must be presented, the measurement units or any other specific limitations about each feature.

On the other hand, document-based collections are organized as folders where the data instances are collected as separate files. This organization may be like how thousands of image or PDF files are stored in a folder for further processes, or how non-structured databases like MongoDB [19] or Google Firestore work. Document-based collections may or may not follow a specific schema. This collection organization has some advantages over the structured one, e.g. when the amount of data increases, it is simpler to be scaled up horizontally to store much more data on several machines. Datasets which do not follow a predefined schema let the users to store the data in a more flexible manner, i.e. the user can start storing new attributes from a certain time, without worrying about the previously stored data instances.

2.2 Standard Data Processing Workflow

The word "raw" is commonly used to emphasize the fact that the first version of the dataset has not passed any analysis or processing step and is exactly as it has been stored originally. Almost never the raw data could be fed directly into machine learning or deep learning models. A data processing project can be broken down into few main phases, each one modifying the data to make a more informative version. Each step forms a pipeline where the data flows and gets more value. Although the main stages are usually the same in many data processing projects, there is no standard pipeline for each one. The details of each phase are highly related to the problem's physics/logic and the expert's judgement, and even could be designed and implemented distinctly by different persons. In this section the standard workflow in any data processing project is explained.

2.2.1 Data Cleaning & Preparation

Almost always, the first step in a real data processing project is preparing a dataset suitable for further statistical analyses. This phase is considered as the foundation of the project and is usually the most complicated and time-consuming step. There is a long list of issues to be considered in data preparation, and many of those must be performed in a close collaboration between data scientists and field experts. Regarding the point that in reliability engineering the data usually is generated by multiple sensors, hereafter, when we talk about datasets mainly, we mean flat structured datasets organized in tables, where each row is an observation and each column corresponds to values of a sensor.

"Indexing" is the first thing to do on datasets. Indices are the names selected for the rows and columns and make it possible to address/select each element or subsets of the table. They are also very helpful when several datasets are aimed to be merged or concatenated. This process is called data alignment. Many other activities like resampling, rolling and windowing are also directly dependent to these indices. Usually, the rows indices are ordered numbers, date-times or intervals. In many use cases one must make sure that the dataset has no different rows with the same index values.

Most of data processing algorithms only work with numeric features, so it is of big importance to convert any other feature types like text strings, categories, etc. to numeric values. These conversions must be done very carefully, because they are severely prone to errors. For instance,

when converting strings to numbers, a same string for different observations could be registered distinctly with lowercase or uppercase characters, while they must be transformed to the same numeric value. These data type conversions many times may generate NaN (Not-a-Number) values, which must be dropped or replaced with other numeric values. The missing values can be replaced with a constant; or be calculated through a group-wise or overall averaging, or even some kind of interpolations.

Reshaping, sorting, transposing, filtering, manipulating, function applications, and feature aggregations are other frequently used operations in data cleaning and preparation phase. Data windowing, window-based calculations (data rolling), and resampling are also very common, especially in time series. When all these operations are done the data is saved in more efficient data files. Such data formats are designed and optimized for storing a large amount of numeric data frames with fast read/write facilities for data intensive projects. Some examples of such file extensions are Hierarchical Data Format (HDF) [20] or Python numeric zipped archives (.npz files).

2.2.2 Data Analysis

When the dataset is prepared, it gets the time to explore the data to learn some of its important characteristics. This exploration more likely aims to remove redundant and non-relevant parts of the dataset, which consequently results a more performant model and helps to better understand the existing patterns in the data. Calculating statistical parameters of the features, the correlations between them and the targets, and some primary visualizations are usually the first things to do. Such simple steps provide a good familiarity with the dataset and could help to find some obvious patterns in the data.

As discussed in the previous sections, the final goal of many data processing projects is to make models which can perform predictions based on what they have learnt from the dataset. Before starting to work on any model, the dataset must be split into two parts: train and test. Train set usually has about 70-80 percent of the whole data and is used to train different models. The test set contains 20-30 percent of the samples and is completely hidden to all models and analyses while training. The test set is used to measure the generalization capability of a model in predicting the targets of new observations that the model has never seen before. If a trained model works fine on the test set, it shows that the model has found the main governing patterns in the data during the training process. On the other hand, if the trained model does not produce good results on the test set, the model has been overfitted to the training data and has learnt some random noise structures rather than the useful patterns.

The next step is to make sure that all features are considered equally in the analysis. Different features represent distinct attributes and are measured in different scales and units. Therefore, very probably the models will tend to learn more from the features with bigger value ranges. To avoid this, all features must be transformed to the same scale. Scaling usually is done through:

- Min-Max Scaler which scales all data between a minimum and a maximum value, or
- Standard-Scaler which forces the data to have zero mean and unique standard deviation.

It is important to remember that the test dataset represents the new data samples that we have no access to while doing all analyses, so the scaler parameters must be determined by the train set, and then be used to transform both train and test sets.

Feature selection is a principal analysis to do before any further modelling. In some contexts, it may also be called as feature importance analysis, or dimensionality reduction. There are various criteria and methods to use which are listed as follows [21]:

- amount of variation
- pairwise correlation
- multi-colinearity
- principal component analysis (PCA)
- cluster analysis
- forward selection
- backward elimination
- stepwise selection
- LASSO
- Tree-based selection

Choosing the right dimensionality selection criteria is a case-dependent procedure, and usually must be done by testing different criteria on the dataset. The detailed explanation of each criteria is out of the scope of this thesis and could be find in the reference [21].

2.2.3 Data Modelling: Machine Learning

Machine Learning (ML) is defined as the science of programming computers so that they can learn from data [22]. In other words, a computer program that can recognize hidden patterns in the data, and makes predictions using those patterns is a machine learning model. The difference between ML and usual computer programs, is that the first one recognizes existing patterns by experiencing various data instances, while the latter one makes the decisions based on a list of deterministic predefined rules. Therefore, ML models theoretically get more and more performant having more experiences, but normal computer programs keep the same performance while they use the same list of rules. Moreover, when the input data changes over the time, ML models can adopt themselves to get the new patterns.

ML has infinite applications, but some of the most important ones are [22]:

- Problems where non-ML solutions need a long list of rules. Such traditional programs are extremely difficult to maintain.
- Complex problems where traditional approaches have no solution for.
- Fluctuating environments and noisy datasets.
- Problems where the aim is to recognize patterns and find an insight about a large amount of data.

2.2.4 Data Modelling: Deep Learning

Deep Learning (DL) is a subset of machine learning and is all about Artificial Neural Networks (ANN). There are some differences between DL and other traditional ML methods. ML models usually try to recognize the patterns based on some priori logic/assumptions and they mainly work more efficient with small to medium size datasets. Therefore, if a model shows low performance, the data scientist can modify the priori logic or choose different assumptions and try again to train the model. On the other hand, ANNs learn from the data by heart, i.e. there is no priori logic to the model, and model only learns by experiencing a large amount of data samples. DL basically needs much bigger datasets for training.

The building blocks in ANNs are layers of several types and use cases. The simplest and the most usual layer is a fully connected or a dense layer, which is a collection of neurons (units) which perform a linear combination of the inputs followed by a non-linear transformation called the activation function. Figure 2.1 shows a schematic view of a network with an input layer and three following dense layers. As it is shown in a dense layer, every neuron is connected to all neurons of the previous layer.

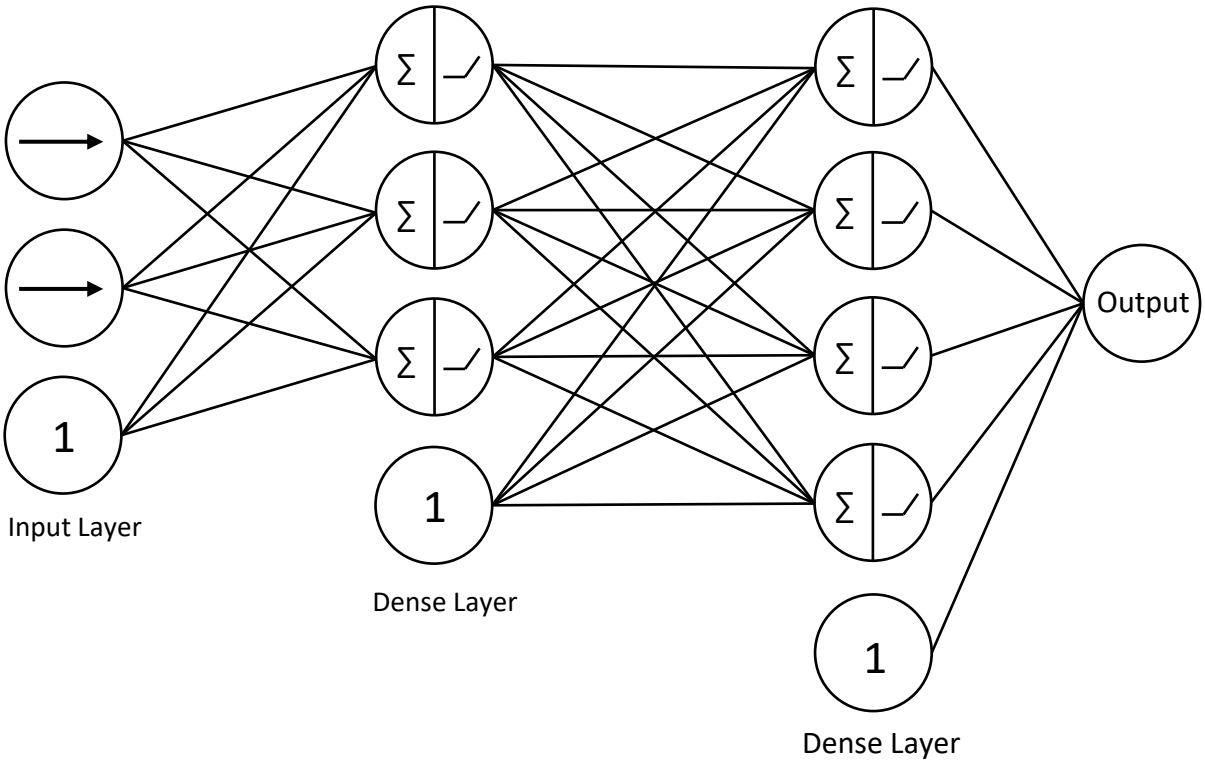


Figure 2.1. Artificial neural network structure

Equation 2.1 explains the transformation a dense layer does on its input.

$$h_i = \varphi\left(h_{i-1}^T w_i + b_i\right) \qquad \text{Eq. 2.1}$$

where h_i is the output of the layer i, φ is the activation function, and w_i and b_i are the corresponding weights and the bias, respectively.

In the simplest case, ANNs start from an input layer, and after few hidden (intermediate) layers, they finish to an output layer. However, nowadays multi-input, multi-output networks are also quite usual in special problems. The features of one data instance enter to the units of the input layer, and the data flows forward through the layers, and each layer transforms it regarding its weights, bias, and the activation function, and the network finally generates a value in the output layer. This value is compared to the target (label) of the same data instance via some comparison functions (loss functions) which are different according to the nature of the label in distinct problems. The comparison results an error, and this error flows backward in the network and modifies the weights of all layers. This backward flow of the error is called “back propagation”.

In the training process the same procedure repeats again and again, and in each iteration usually the error value decreases, as the weights get updated. Through the training iterations each layer learns and understands some parts of the existing patterns in the data, and because this learning is performed only by experiencing a lot of data instances rather than entering a priori logic to the model, it is called learning by heart. Usually the first layers (shallower ones) learn very basic and low-level patterns in the data. Deeper layers combine these low-level patterns to find more complicated and higher-level ones. That is why the ANNs with several consecutive layers are called Deep Neural Networks (DNS).

2.3 Supervised vs. Unsupervised Learning

As discussed in the previous sections, datasets may or may not have target values. In some cases, for each data instance it is only determined if the equipment has a normal or an anormal state, but no more information about the probable fault type or other fault characteristics is provided. Usually, in these cases the number of normal instances is much more than anormal samples. Anomaly Detection methods could help in these situations. In other cases, we may have access to the fault type (as a categorical variable) or some fault parameters like length, intensity, etc. (as continuous numeric values) and the purpose is to predict the class labels (classification), or the target values (regression) for new data samples. All these situations make an important group of methods called supervised learning.

On the other hand, in several problems the data has no determined target, and only the features are available. Another group of ML models called unsupervised models can help to extract important patterns in the data. Such patterns may be used to show the dependence between instances, to divide the data samples in different clusters, or to generate (simulate) new data with the same patterns. Unsupervised methods are usually very sophisticated and advanced algorithms with growing applications in different areas.

The combination of these two groups are known as semi-supervised methods, and usually are used when the final purpose is to perform classification or regression, but only a few amounts of data has corresponding target values. In these situations, we usually assign each sample to a group, using a clustering method, and then according to the few labeled samples, we can generate the labels artificially. Semi-supervised models can also be very beneficial in

Representation Learning where the main goal is the extraction of the most important features. Such features could also be used for further performant classification or regression tasks.

Chapter 3

Methodology

To develop a graphical programming framework which enables users to develop deep learning models to solve reliability and prognostic maintenance problems, and to reach the objectives of the thesis the following methodological steps are necessary:

1. **Literature Review**
   The proposed platform offers a wide range of neural network layers to implement advanced ANN architectures. Hence, before the development step a complete literature review of the common deep learning applications in physical asset management and reliability engineering is needed. Moreover, the proposed platform is designed mainly on top of flow-based programming and visual programming concepts. As a result, a good level of familiarity with these disciplines is a must.

2. **Technology Review**
   The implementation of the proposed platform uses a complex combination of several technologies from back-end and cloud services to front-end and graphical developments. Knowing the necessary technologies for the project is not the whole thing. By growing the open source development culture, one would have the option of choosing between tens of different frameworks or libraries to implement each part of the product. Selecting the optimum option for each part is not a trivial task and needs an intensive review of the documentation of each framework, comparing their pros and cons, and even estimating/evaluating the stability and active maintenance and development of each option in the future.

3. **Platform Development**
   The development of Deep Learning Hub is divided into four main sections:
   - Back-end, which is done based on Google Firebase and is responsible for hosting, authentication, and storing the users' projects on the cloud.
   - Front-end, which is developed based on common web technologies and contains the general graphical user interface and manages all the workflow and navigations in the application.
   - Local Python connectivity, which is implemented based on Jupyter, an open source project originally called IPython. Nowadays, Jupyter is considered as one of the standard tools in any data science project and provides a lot of useful libraries for the developers as well. In DLHub we have adapted Jupyter Kernel Gateway and after some minor modifications we rebranded it as DLHub-Gateway to make sure that the users install and use the modified version.
   - Custom graphical parts including the graphical programming module and the report diagrams are implemented using D3JS (Data Driven Documents), which uses SVG (Scalable Vector Graphics) technology for producing low weight, high quality graphics.

4. **Testing Framework on Public Datasets**
   When the proposed platform is ready, it must be evaluated using well known public datasets. Chapters 4 of the thesis explains the functionality of the app by implementing a vanilla neural network for classification of the Fashion MNIST public dataset. Chapter 5 provides the implementation of more advanced architectures like convolutional neural networks, autoencoders, variational autoencoders, and generative adversarial networks, all using the same dataset. In each example, the automatically generated Python codes have been revised precisely to evaluate the correctness of the implementations in the platform.

5. **Testing Framework on Real Datasets**
   When we make sure that all parts of the platform work fine, it is time to evaluate its usefulness in a real industrial health management problem. The case study is done on two vapor recovery units (VRU) that work in parallel in a natural gas offshore platform. The VRUs are monitored for a period more than a year. For each unit, 172 sensors recorded different parameters of the system every 15 seconds. During the whole operation period, the working state of the equipment has been supervised and registered by maintenance engineers. The dataset contains about 2,400,000 registries of all 172 features. The anomaly detection study performed on the dataset clearly shows the simplicity of the implementation of such complex algorithms in DLHub and proves that the proposed platform helps the engineers focus on their main business rather than programming challenges.

6. **Discussing the Results and Conclusions**
   Finally, the results are discussed, the objectives of the thesis are evaluated, the important points and results of the thesis are concluded, and some future works are suggested.

Chapter 4

Proposed Framework: Deep Learning Hub

This chapter presents the proposed solution for the challenges mentioned in chapter 1. The justification for the proposed framework and its implementation are discussed in the following sections. The rest of this chapter explains the details of the framework through a classification problem via a vanilla artificial neural network.

4.1 Deep Learning Frameworks

In the recent years, many frameworks have been developed to make use of machine learning and deep learning as easy as possible. They provide high-level APIs for users to dive into the most recent and advanced algorithms without having a very solid theoretical knowledge or concerning about the implementation complexities. The good news is that most of these frameworks are developed as open source projects where a big community can contribute in the development, and they often have some main supports from several huge tech-companies. One of the mostly accepted and commonly used frameworks is Tensorflow [15] which originally is developed by Google.

As figure 4.1 shows, Tensorflow has a layered architecture and consists of a stack of APIs built on the top of an execution engine. In the stack, higher-level APIs are more like the human languages, and are very easy and straight forward to use. On the other hand, low-level APIs are the parts where the user can work directly to the hardware. They are more complicated to employ by many users, and usually are useful in implementing advanced algorithms. Although the development of integrated and mature frameworks like Tensorflow made the implementation and deployment of DL based fault diagnostics and prognostics models less challenging than ever, still “coding” remains the main barrier for many non-computer-scientists in the reliability and maintenance community.

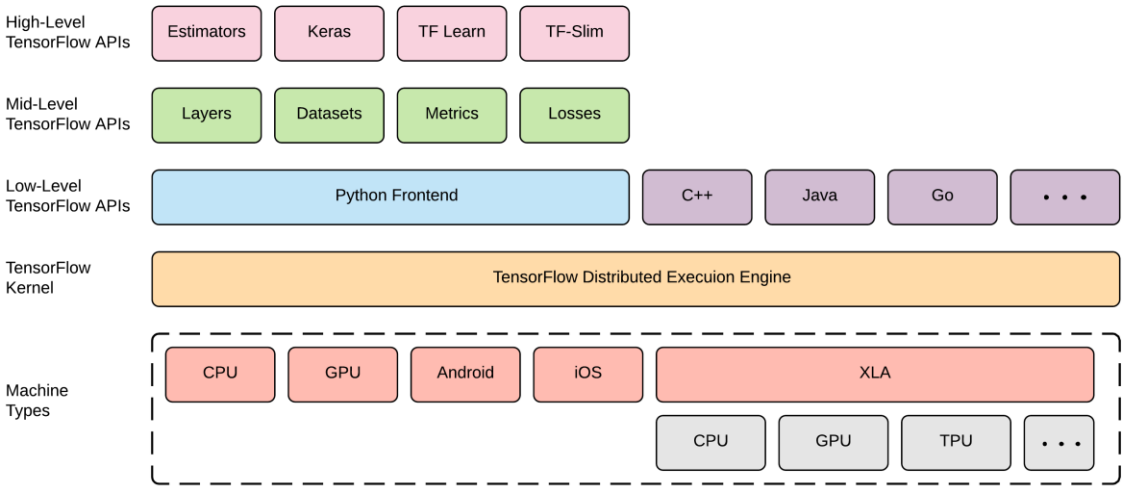


Figure 4.1. Tensorflow architecture [23]

4.2 The Proposed Framework

In order to tackle the challenges mentioned in chapter 1, here we introduce Deep Learning Hub (DLHub), an open access and web-based graphical programming environment for developing deep learning solutions in general, and for reliability and maintenance areas in particular. DLHub adds a new level of visual programming APIs on top of Tensorflow, where implementing DL models only needs a general knowledge on deep learning and its applications. While the user does not need to write a single line of code for implementing DL models, the automatically generated code is accessible for debugging or learning purposes. Figure 4.2 presents the general workflow in the application.

The general architecture of DLHub is like Jupyter notebooks [24]. It consists of two separated components that work together: a web application (https://dlhub.app) and a Jupyter Kernel Gateway. The web app makes a visual environment with facilities and tools that offer an intuitive way to implement and develop a deep learning project. More details on this subject are discussed in the following sections. The kernel gateway, on the other hand, is the official product of the Jupyter open source project with minor changes. It has been rebranded as “dlhub-gateway” to be distinct from the official version and can be installed simply via standard Python package management system. This gateway is responsible to run the codes on the client’s machine.

The front-end is implemented as a web application, and its first page is shown in figure 4.3. Using web technologies brings many advantages by itself. First, a web application works cross-platform on many operating systems (Windows, Mac and Linux) that have a browser installed. Theoretically, using DLHub on any modern browser should make the same experience for the users, but Google Chrome is the preferred one to be used, and many of the debugging processes in the current version have been performed on that. Second, upgrading the web application on the server brings the last features for every user without any further installations. Third, being a Single Page Application (SPA) helps the app to work completely inside the browser and without any further communications with the server. It makes the app more stable on weak internet connections, and even lets the application to work offline while it has been loaded completely. Finally, it is possible to open DLHub on various tabs and to work simultaneously on different projects. This could help the user to compare various models on the same dataset simultaneously.

All the communications between the web app and the kernel gateway are done via HTTP and WebSocket protocols. HTTP is a request-respond protocol which means the web app sends a request to the gateway for doing something, and the gateway sends back a response. DLHub uses HTTP to perform some operations like starting a new kernel or interrupting or shutting down the existing kernels, as well as sending the automatically generated code to the gateway to be ran. On the other hand, WebSocket is a protocol that provides full-duplex communication channels over a single connection. Here, it is mainly used to transfer the execution results in real time. These connections offer an important advantage in DLHub. The users develop DL projects via an online tool and run them on their local machines. That way, they get all the benefits of an online tool, while keeping 100% privacy of the data.

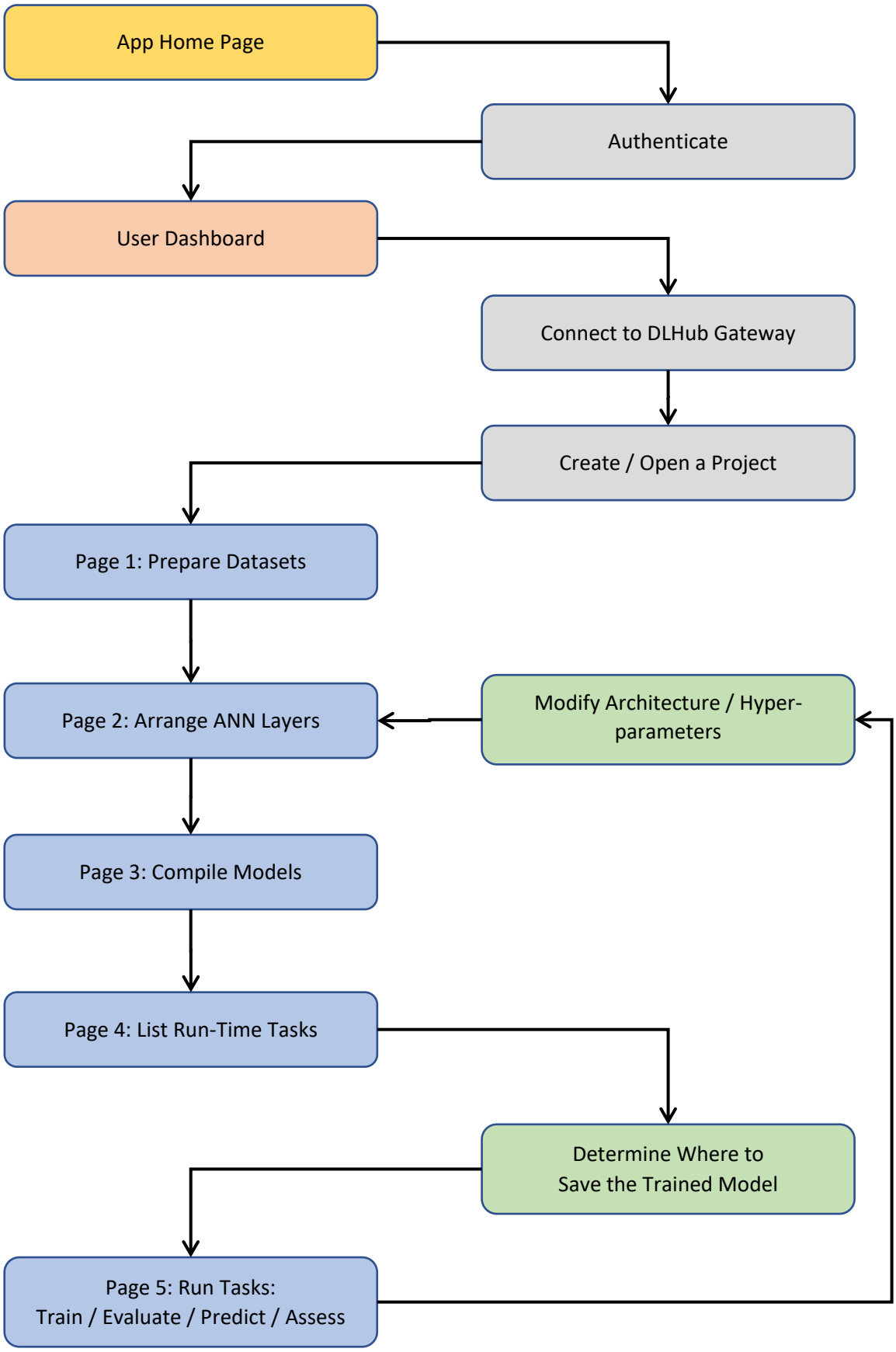


Figure 4.2. General workflow in DLHub

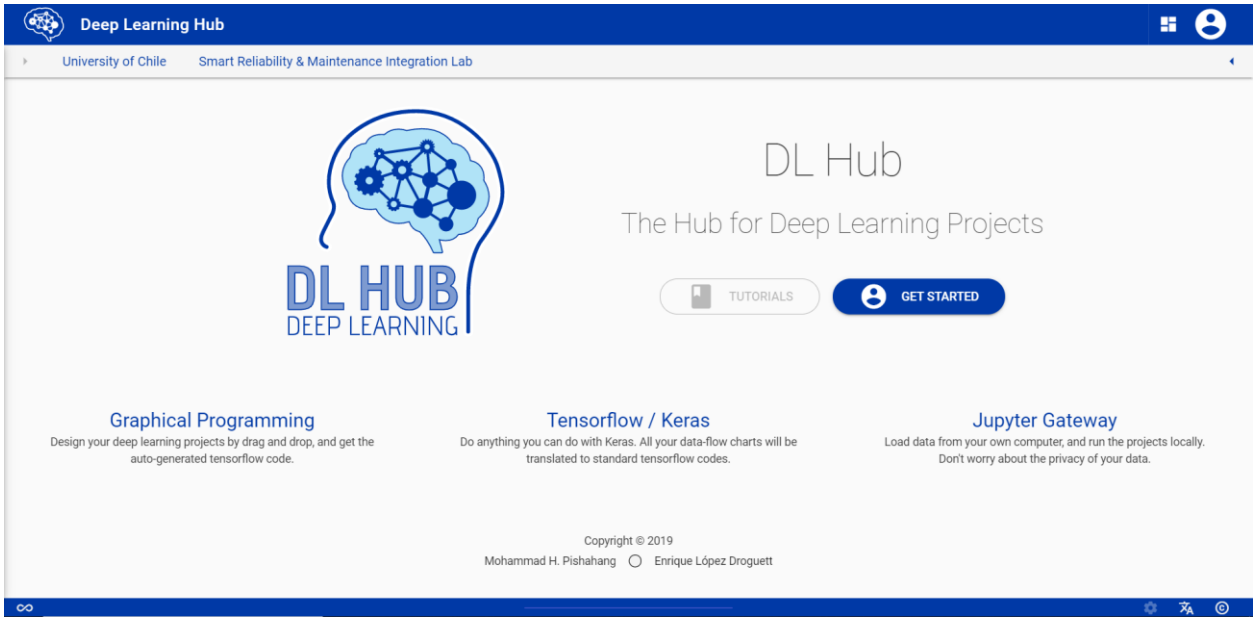


Figure 4.3. DLHub home page

4.3 Requirements

DLHub is built on top of Tensorflow Python API and runs everything locally. Therefore, the client needs to have Python and a few libraries installed on the computer. Tensorflow 2 has some requirements that are listed in table 4.1. DLHub does not have any additional requirements, and only needs some more libraries being installed on Python environment. These libraries are among the most common ones in any data science project, and many usual Python distributions have them installed by default.

Table 4.1. Tensorflow requirements
[adapted from https://www.tensorflow.org/install]

Python	Versions 3.5 - 3.8
Operating System	64-bit OS
Windows	Windows 7 or later with C++ redistributable [https://support.microsoft.com/en-us/help/2977003/the-latest-supported-visual-c-downloads]
Linux	Ubuntu 16.04 or later
Mac	macOS 10.12.6 (Sierra) or later

One can install and configure Python manually, but it is preferable to use Anaconda to install and configure it automatically. Once Python is installed, the user can install DLHub Gateway by running the following snippet in the OS command line (console):

```
pip install dlhub-gateway
```

Then running the following snippets makes sure that Tensorflow 2.0, and the latest version of the other required libraries are installed respectively:

```
pip install tensorflow==2.0
```

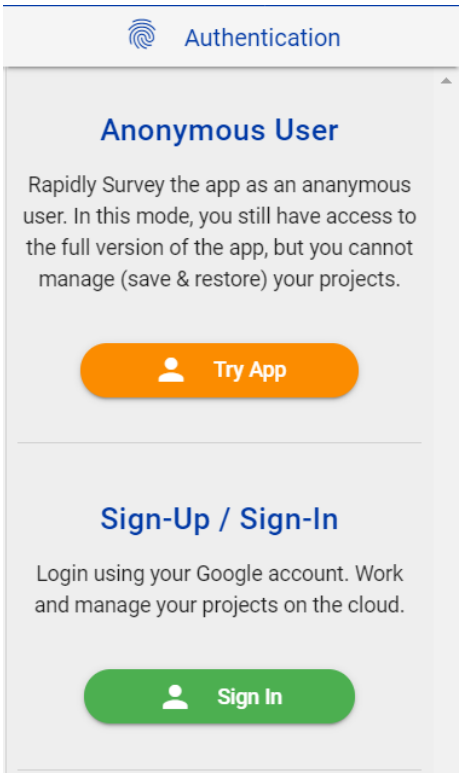
```
pip install --upgrade numpy pandas scikit-learn xlrd
```

DLHub has four main Python libraries as dependencies. Tensorflow is used to implement any ANN architectures and algorithms. Like any other open source project, Tensorflow has experienced a lot of changes in its evolution path, which sometimes are not completely compatible with the previous versions. Currently Tensorflow 2.0 is the supported version in Deep Learning Hub. Scikit-Learn is the most common and sophisticated ML library in Python and here a few parts of it are used mainly for dataset preparation (e.g. train-test split, scaling, …). Pandas provides the safest way to read data files in different formats, and for reading MS Excel files it would need "xlrd" library to be installed. Finally, Numpy is the most standard Python library to work with numeric matrices. It is highly optimized in memory usage and the most parts of its foundation are written in C++, which makes it extremely fast. Many data analytics libraries, including Tensorflow and Scikit-Learn, are written on top on Numpy.

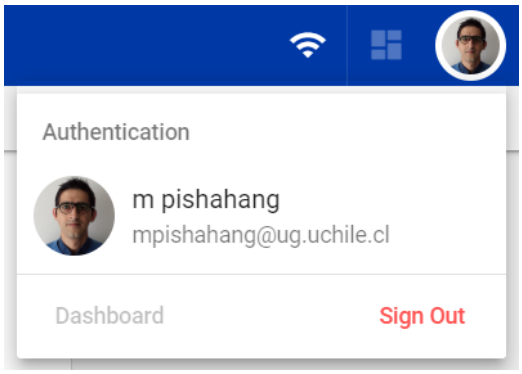
4.4 Authentication

DLHub lets the application to be used anonymously. In this mode still the full functionality of the app is accessible, and everything runs on the client's machine, but no activity trace will be saved. However, signing-in to the app makes it possible to save all the projects in the user's account and to work on them later, on any other computer. For the sake of simplicity and security, the users are permitted to sign-in to the app by any Google accounts (Gmail, or any other kind of Google accounts). It is important to mention that after signing-in, only the projects, which are the graphical programming flowcharts, are getting saved on the cloud, but the data completely lives in the client side.

Figure 4.4 shows the details of the authentication process. By clicking on the sign-in button a new window opens and as a Google service, lets the user to safely authenticate with their user account. The first time the user signs in, it would work like a sign-up step, and the app makes an empty profile in the backend. Every user has a dedicated database on the server-side to store the projects. When this process is completed the app redirects the user to the dashboard page, as the main hub to manage the projects. The corresponding menu in the application bar shows the details about the current user and provides a button to logout.



(a)



(b)

Figure 4.4. Authentication procedure.
(a) The user can choose to sign in and to save the projects in the user's account or to trying the app anonymously.
(b) The authentication menu shows the user's info and the sign-out functionality is available.

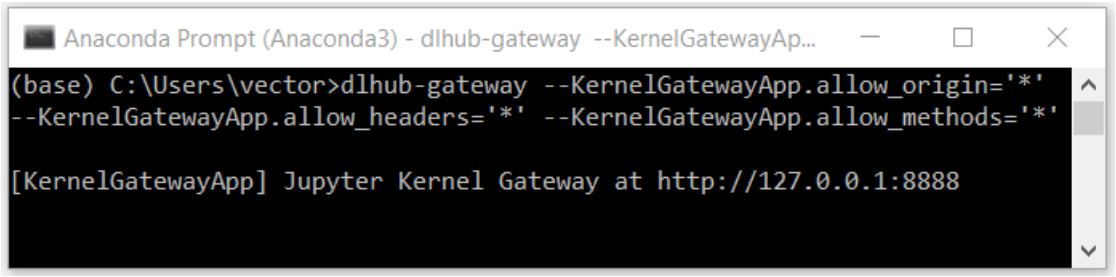
4.5 Gateway Management

Navigating to the app's dashboard, three main panes appear. In the left the gateway manager pane provides a brief instruction for getting connected to the Python ecosystem. The user can decide if she/he wants to get connected to a kernel gateway or not. If the app does not get connected to Python, still it can be used to edit existing projects or to work on a new project. But in this case, the app does not run any code, and it works only as a code generator. However, the best experience achieves by using the full functionality through the communications between app and the gateway.

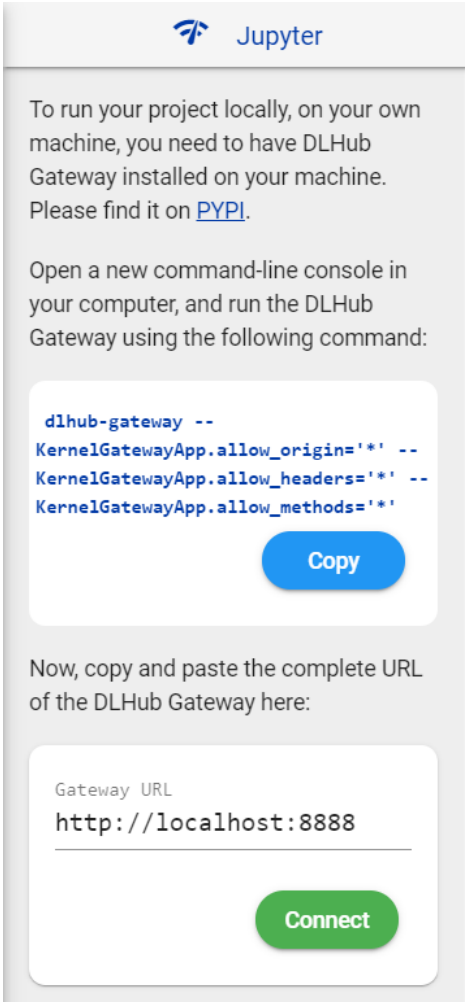
As shown in figure 4.5 Copying and pasting the following snippet in a command prompt runs a new kernel gateway on the client's computer. If the Python is installed correctly and is added to the operating system environment variables, any available command prompt can be used to run a kernel. By the way, if the Python is installed via Anaconda, it is preferable to run such snippets in the Anaconda Prompt.

```
dlhub-gateway --KernelGatewayApp.allow_origin='*'
--KernelGatewayApp.allow_headers='*' --KernelGatewayApp.allow_methods='*'
```

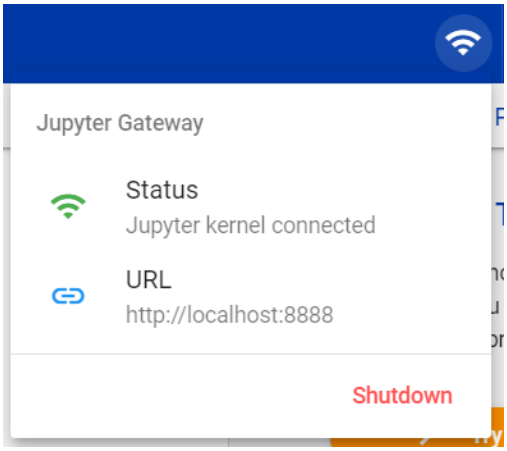
Kernel gateway by default starts on the client's localhost (http://127.0.0.1) and on 8888 port. If this port is occupied by other applications like Jupyter notebooks, another port number will be selected for the kernel gateway. Setting this port number in the Jupyter manager pane, provides the exact address of the gateway and by clicking the "connect" button a new kernel starts on the gateway.



(a)



(b)

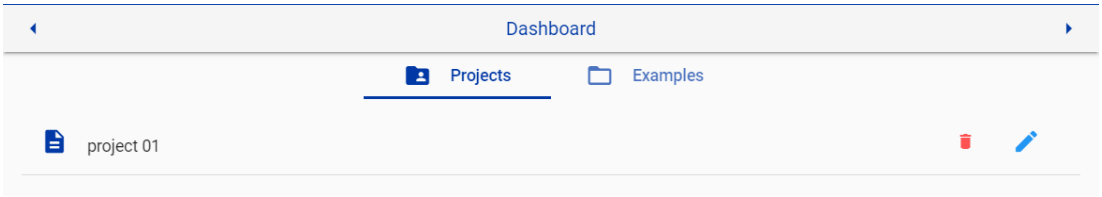


(c)

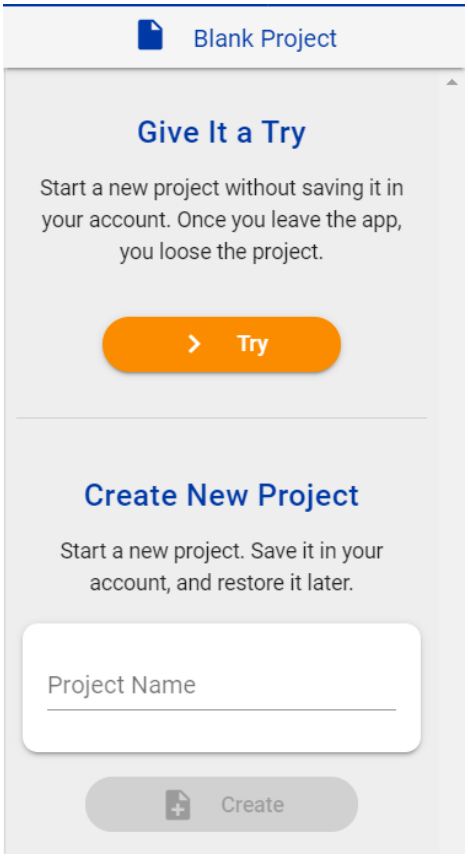
Figure 4.5. Gateway connection management. (a) Running a gateway in command prompt. (b) DLHub gateway manager pane where the connection is configured. (c) The gateway menu shows the details about the current connection and provides a button to shut down the kernel.

4.6 Projects

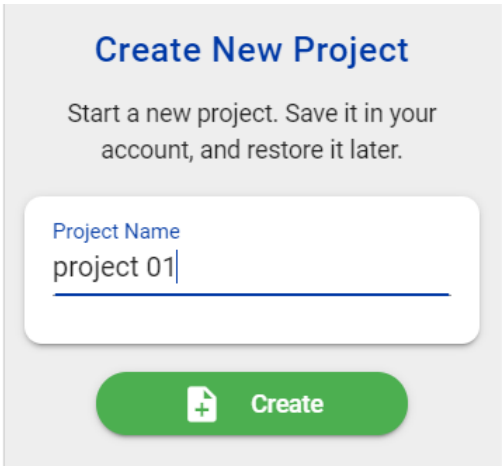
The project pane (figure 4.6) is in the right part of the dashboard and provides two options to start a new project. The user can try the app by working on a project without saving it into the account. It is particularly useful when the aim is only getting familiar with the project structure and the available options. Once the user leaves the app such projects will be lost. On the other hand, creating a new project makes an empty project on the cloud and keeps saving every step of the project. Such projects are listed in the central part of the dashboard, and they could be reused, modified, or deleted on any other computer. Any project in DLHub has five main sections. By starting a project, the user automatically navigates to the first section. These steps are explained in the following sections.



(a)



(b)



(c)

Figure 4.6. Project management.
(a) The existing projects listed for further use or modifications.
(b) The project management pane.
(c) Creating a new project and saving it in the user's account.

4.7 Preparing Datasets

The first step in any project is the preparation of the datasets to be used in the deep learning models. The user works in a graphical programming module to construct a flux-gram which typically starts with reading a file containing data and after few very common preprocessing blocks, ends up with one or more datasets. It is important to mention that the preprocessing blocks that already exist in this step are limited to some simple data structure manipulations, few scaling blocks, and a couple of data label preprocessing. More sophisticated preprocessing steps usually must be performed outside the app (typically in a Jupyter notebook). Figure 4.7 shows available categories of blocks to prepare datasets, and table 4.2 lists all existing blocks in each category. Regarding the point that this app is not aimed to focus on data cleaning and preprocessing, we tried to define as less number of necessary blocks as possible in this step, and it implicitly forces the users to follow some disciplines in saving their data files, which could be considered as a good by-product.

Moreover, the project progress menu in the application bar shows a complete list of steps one must pass in a project (figure 4.8). This menu especially has been added to help the beginners understand their whole data modelling path in a typical deep learning project. It also offers a "project details" button that gives the opportunity to see a summary of all prepared datasets, added input/output layers, and defined models in a separate pane. It shows some important detailed information about each part. It contains the dimension of the datasets, a tree structure representation of the models, the determined loss functions for each output layer, and finally the optimizer chosen for each model.

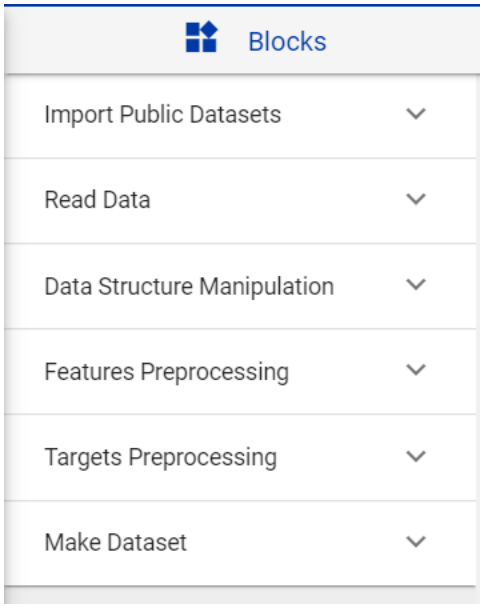


Figure 4.7. Available dataset preparation block categories

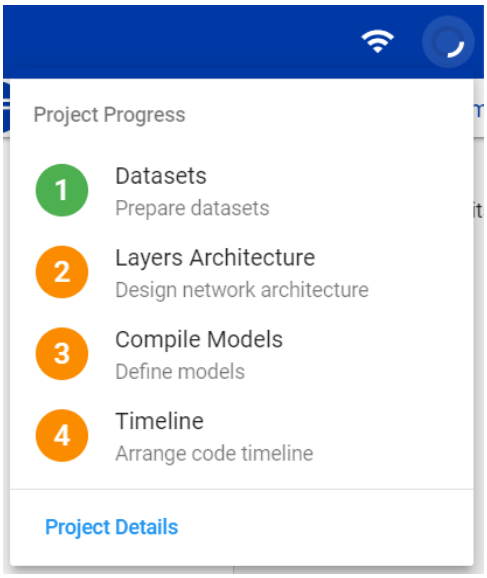


Figure 4.8. Project progress menu showing the status of the project

Table 4.2. List of all available preprocessing and dataset preparation blocks

Import Public Datasets		
Boston Housing CIFAR 10 CIFAR 100	Fashion MNIST IMDB	MNIST Reuters
Read Data		
Read Text File	Read Excel File	Read Pickled File
Data Structure Manipulation		
Extract From Table	Extract From Dictionary	Train Test Split
Features Preprocessing		
Min Max Scaling	Manual Scaling	Normalize
Targets Preprocessing		
Label Encoding	Label Binarizing	
Make Dataset		
Make Dataset		

A typical data preparation flux-gram is shown in figure 4.9. It starts with a block to read data from a local file. The file can be a Comma Separated Values (CSV), Excel, or a Python pickled file. Usually pickled files are preferred as they are optimized to save numeric values, with fast read/write operations. Pickled files return a dictionary of matrices, that can be chosen by “extract from dictionary” blocks. Here, features and targets are stored in two matrices. As explained in Chapter 2, train-test split shuffles the data and generates two sets corresponding to train and test sets respectively. The features need to be scaled using a scaler block. One must keep in mind that the train features are used to set the scaler parameters, and then both the train and the test feature sets are transformed using the configured scaler. The target values do not need any scaling, but in some cases, they also need some additional preprocessing.

In DLHub, a dataset is a block containing two matrices: the first one is the features, and the second one is the targets (labels). A dataset does not have necessarily both features and targets. They also may have only features or only targets in more complicated problems. While working on a

supervised learning problem, two separate datasets corresponding to train and test sets must be created. All the training procedure will be done using the first one, and the latter one is used to measure the generalization capability of the models when they encounter new data instances. In unsupervised learning problems (including generative models) only one dataset is needed to train the models.

Selecting each block opens an editor in the right side of the page where its parameters can be set or modified. Figure 4.10 shows how the “min-max scaler” block is configured. The block editor consists of a series of cards, each one responsible for a group of parameters. For this specific block transform sockets may be added

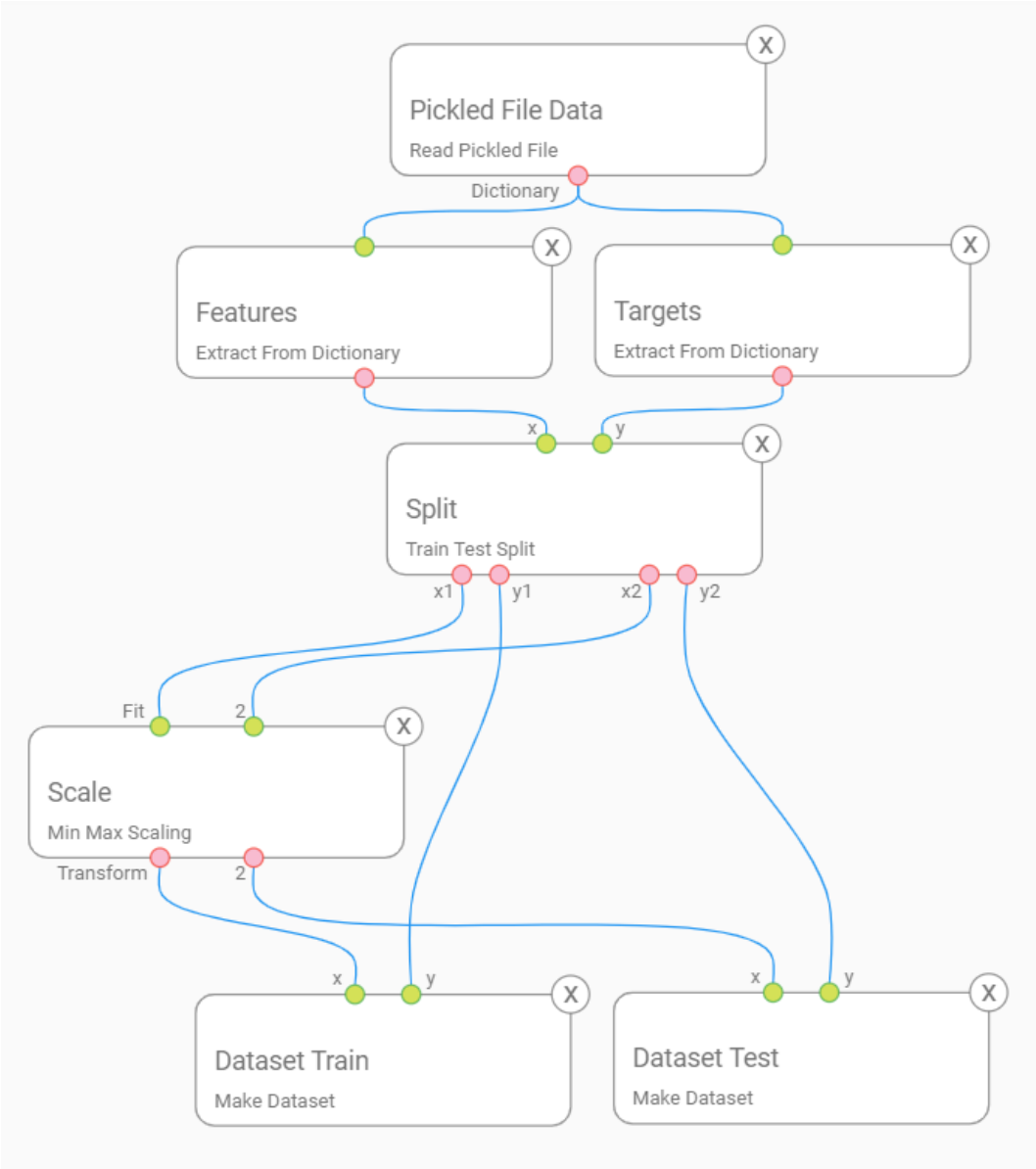
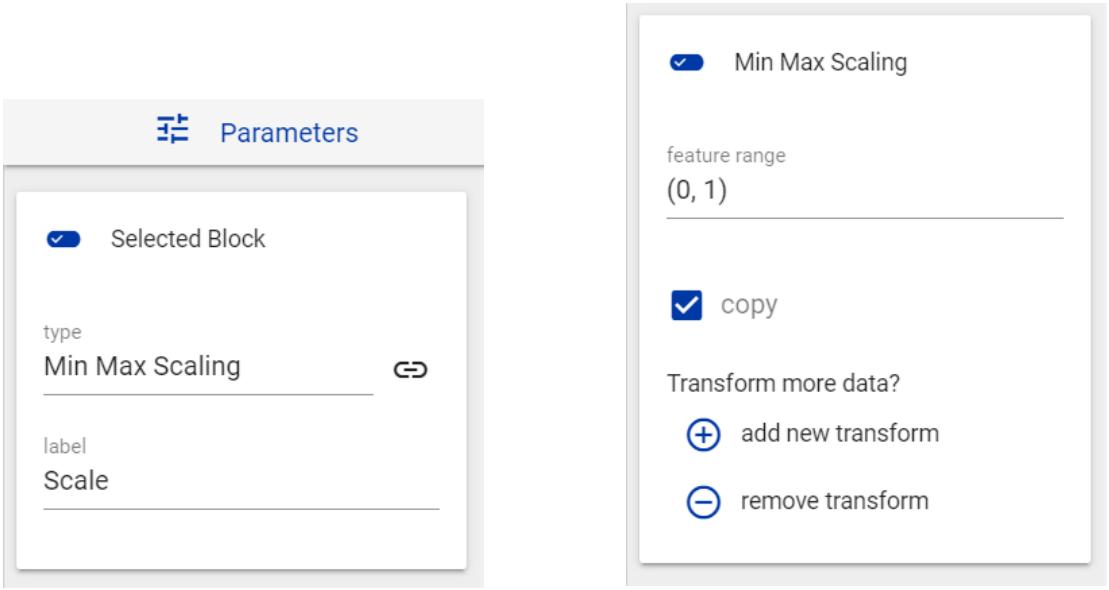


Figure 4.9. Typical dataset preparation flux-gram



(a) (b)

Figure 4.10. Setting parameters of the scaler block.
(a) The general block parameters containing the type, a link to the official documentation, and a label in the graphical programming module. (b) Scaler-specific parameters containing the feature range, and buttons to add/remove transform sockets.

Explaining the whole project is simpler through a real example. Many references use some publicly available datasets in the examples. This way, the readers can easily repeat the same algorithm on their own and compare the results with the original text. One of the most favorite datasets in deep learning community is MNIST which contains thousands of grey-scale 28 by 28 images of handwritten digits [25]. The home page of the MNIST dataset mentions that "it is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting". Moreover, running different algorithms on the same dataset makes them comparable and brings the opportunity to compare them with state-of-the-art benchmarks.

In this chapter Fashion MNIST dataset [26] is used. It has the exact format as the MNIST dataset, but it contains grey-scale 28 by 28 images of fashion articles. The dataset has 60,000 train and 10,000 test images. The images are labeled in 10 classes (T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot). Figure 4.11 shows some examples of the 10 classes. The creators of this dataset have had serious conversations in the community to replace the original MNIST with that. They supported this suggestion with few reasons [26]. First, they believe that MNIST dataset is too easy. Convolutional nets can achieve 99.7% on MNIST. Classic machine learning algorithms can also achieve 97% easily. Second, MNIST is overused in artificial intelligence bibliography. Finally, MNIST cannot represent modern computer vision tasks.

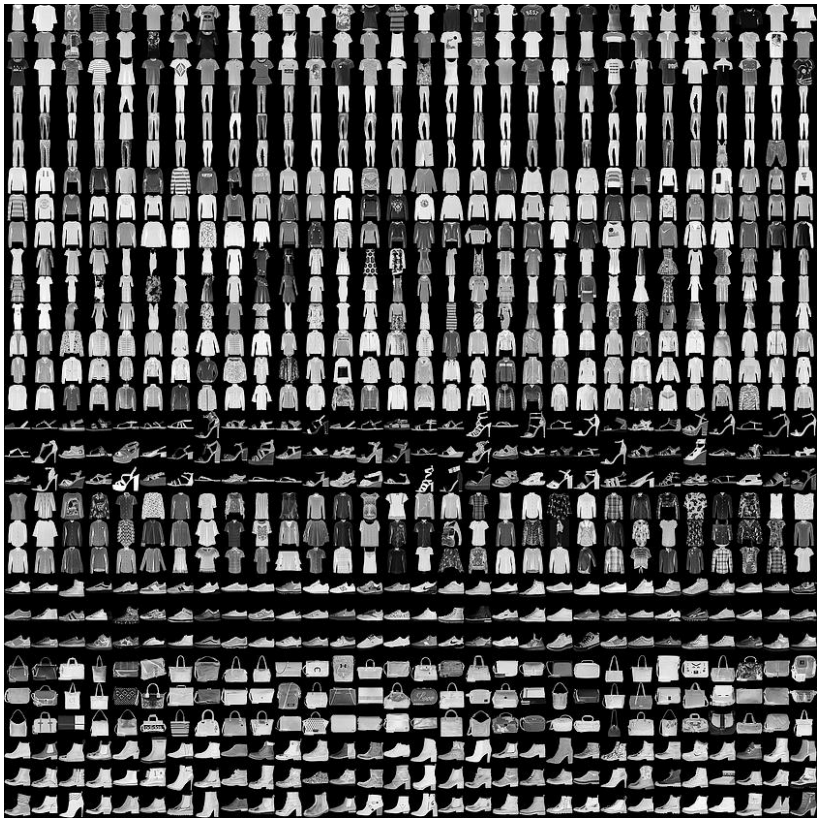


Figure 4.11. Fashion MNIST dataset [26]

Figure 4.12 shows the dataset flux-gram for a classification problem using Fashion MNIST. DLHub offers ready-to-use blocks for some standard datasets launching with Tensorflow. The data already has separate train and test sets. As each pixel in the images is a value between 0 (black) and 255 (white), they just need to be divided by 255 to scale down all values between 0 and 1. “Manual scaling” block is used for such simple scaling process. Finally, two datasets are created by assigning the features and the corresponding labels. In DLHub every block dragged into the window gets a unique id number, so generally the blocks necessarily do not need to have different labels. By the way, for datasets we also need to put a unique label for each block as they are going to be reused in the following steps, and the easiest way to recognize them for the user is through their labels.

Whenever a block added/removed to the working window, any of their parameters change, or the links between the blocks gets modified, DLHub automatically generates a very standard Python/Keras code equivalent to the flux-gram. Figure 4.13 shows the code for the dataset preparation step. In this stage it also adds some lines of code to import all the necessary libraries and defines some utility functions/classes which are used in the following steps. This code can be used for debugging purposes or it could be reused in code editors for deeper learning or more advanced applications.

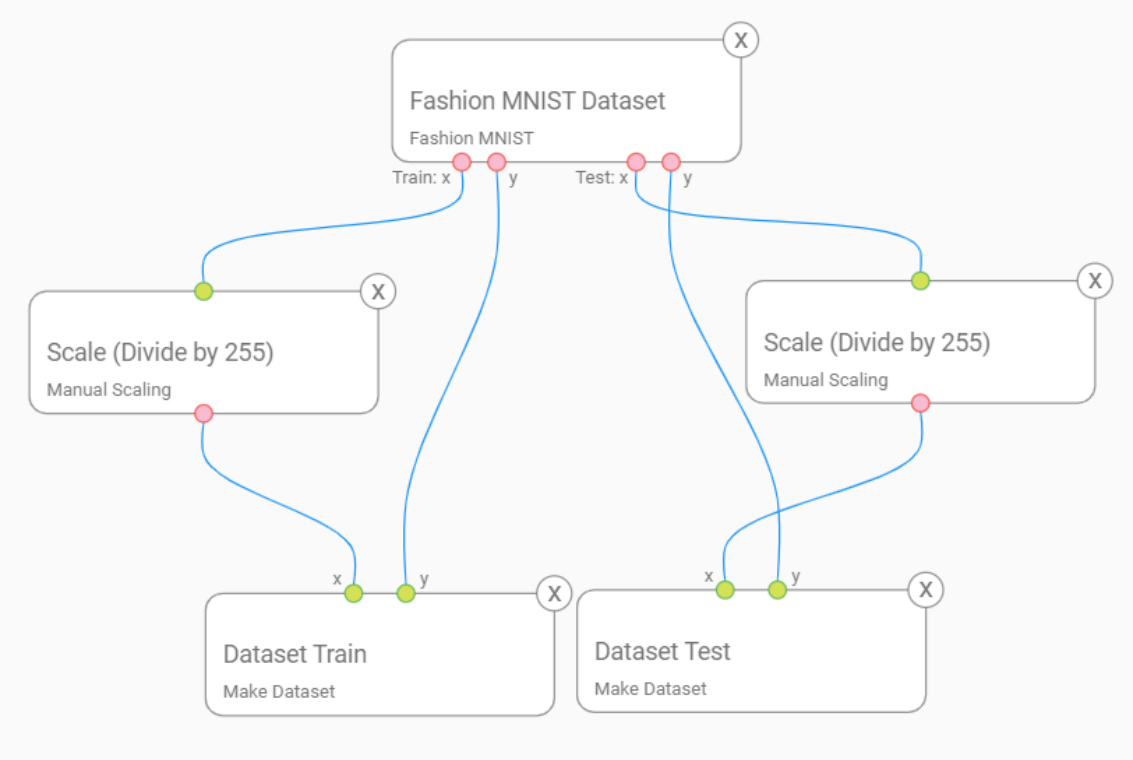


Figure 4.12. Fashion MNIST dataset preparation for classification

Generated Code

```
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)

import tensorflow as tf
import numpy as np
import pandas as pd
import sklearn
import sklearn.preprocessing
import sklearn.model_selection
import sklearn.metrics

def kl_divergence(p, p_hat):
        return (p * tf.keras.backend.log(p / p_hat)) + ((1 - p) * tf.keras.backend.log((1 - p) / (1
- p_hat)))

class KLDivergenceRegularizer(tf.keras.regularizers.Regularizer):
        def __init__(self, weight, target=0.1):
                self.weight = weight
                self.target = target
        def __call__(self, inputs):
                mean_activities = tf.keras.backend.mean(inputs, axis=0)
```

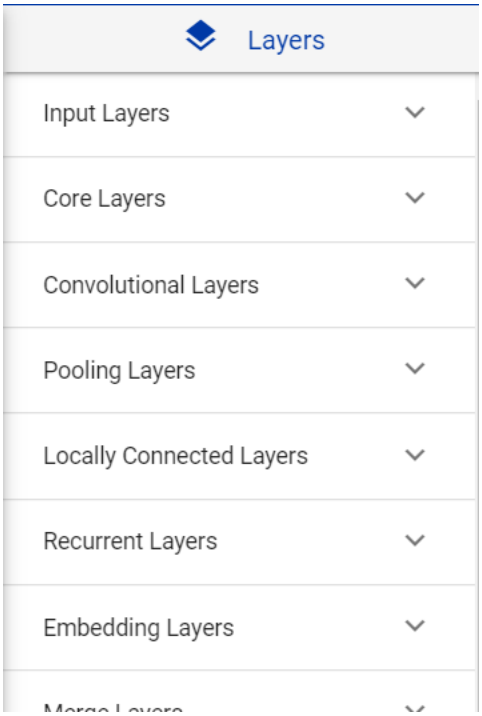
OK

Figure 4.13. Automatically generated Python code for data preparation

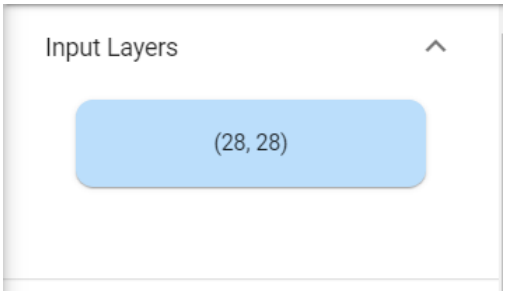
4.8 Layers Architecture

Navigating to the next step, the application runs the generated code and if there are no errors it advances to the second phase. In case of any errors the application makes an alert containing some information about the incorrect parts and lets the user to modify the flux-gram. Usually running this step for the first time take a little bit longer as DLHub tries to import all libraries, and to read data from local files. In the case of public datasets like Fashion MNIST it also may try to download the dataset in the first try which takes a bit longer. Now, it is time to make the artificial neural network architecture again via a visual programming module similar to what we used in the first step. All the official Keras layers have been created in DLHub, and the user can construct whatever network that can be done through coding. Figure 4.14 shows some of the block categories provided here, and table 4.3 lists all the existing blocks.

The architecture starts with one or more input layers and can end up with one or more outputs. DLHub sets automatically the dimension of input layers based on the datasets defined in the previous step. Figure 4.14 shows how DLHub added an input layer of 28 by 28, as the Fashion MNIST dataset contains images of this dimension. It is also possible to have multiple separated architectures. More sophisticated architectures like networks with shared layers, Multi-input/output networks, and “time distributed” layer wrappers are also available in DLHub. Moreover, by selecting a layer, all its parameters can be modified in a very user-friendly manner. After each single modification on the blocks in this step, the application automatically generates a standard Tensorflow code using Keras functional API.



(a)



(b)

Figure 4.14. Blocks to make the architecture.
(a) Some of the block categories.
(b) Customized input layers based on defined datasets. In this case 28 by 28 for Fashion MNIST images.

Table 4.3. List of all available ANN blocks

Input Layers			
Depending on the Datasets			
Core Layers			
Dense	Reshape	Masking	
Activation	Permute	Spatial Dropout 1D	
Dropout	Repeat Vector	Spatial Dropout 2D	
Flatten	Lambda	Spatial Dropout 3D	
Input	Activation Regularization		
Convolutional Layers			
Conv 1D	Conv 3D	Up Sampling 2D	
Conv 2D	Conv 3D Transpose	Up Sampling 3D	
Separable Conv 1D	Cropping 1D	Zero Padding 1D	
Separable Conv 2D	Cropping 2D	Zero Padding 2D	
Depth-Wise Conv 2D	Cropping 3D	Zero Padding 3D	
Conv 2D Transpose	Up Sampling 1D		
Pooling Layers			
Max Pooling 1D	Average Pooling 2D	Global Max Pooling 2D	
Max Pooling 2D	Average Pooling 3D	Global Average Pooling 2D	
Max Pooling 3D	Global Max Pooling 1D	Global Max Pooling 3D	
Average Pooling 1D	Global Average Pooling 1D	Global Average Pooling 3D	
Locally Connected Layers			
Locally Connected 1D		Locally Connected 2D	
Recurrent Layers			
Simple RNN	GRU	LSTM	Conv LSTM 2D
Embedding Layers			
Embedding			

Table continues in the next page.

Table 4.3. List of all available ANN blocks

Merge Layers			
Add	Multiply	Maximum	Dot
Subtract	Average	Concatenate	
Advanced Activation Layers			
Leaky ReLU	ELU	Softmax	
PReLU	Thresholded ReLU	ReLU	
Normalization Layers			
Batch Normalization			
Noise Layers			
Gaussian Noise	Gaussian Dropout	Alpha Dropout	
VAE Layers			
Sampling			

Figure 4.15 shows a vanilla ANN defined to classify MNIST data. The network starts with an input layer which automatically had been customized to the dataset, following by a flatten, a hidden layer, and an output one. The flatten only reorders the input pixels to a one-dimensional tensor. The hidden layer is simply a dense layer with 128 units and ReLU activation function. Finally, the output is another dense layer with 10 units (corresponding to 10 classes) with Softmax activation which is the typical case for classification problems. All other parameters of the layers have been put as default.

As the figure shows, input layers do not have input sockets, so they cannot be connected to some previous layers. They also have an arrow symbol in the upper left side to emphasize the starting point of the network. On the other hand, any other layer type could be defined as the output as it is necessary in different arrangements. Marking a layer as output in the block's editor pane, adds an arrow to its bottom right side to demonstrate it clearer. In this step one may add tens of layers in different sequences, while in defining the models only input/output layers are needed to be selected. Marking the output layers has been added here to let the application to make a short list of IO layers in the following step.

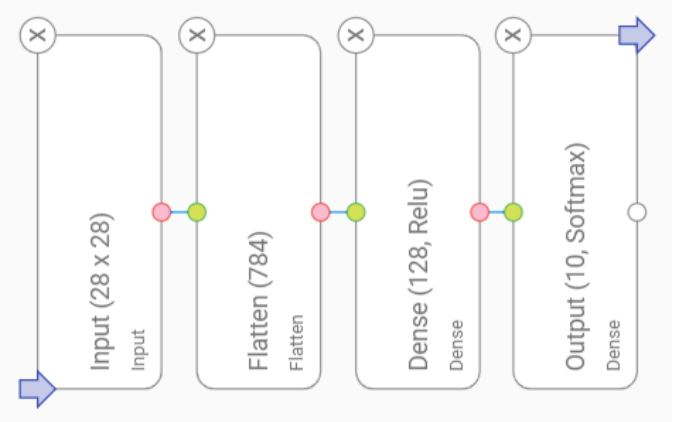


Figure 4.15. A vanilla classifier architecture for Fashion MNIST dataset

4.9 Compiling Models

The third step provides the user with tools that help forming a list of models based on the previous architectures. Figure 4.16 shows the details of different parts of model configuration. Each model gets a default name which can be modified easily. Again, one must make sure that all models have unique names. DLHub uses Keras functional API, so the models can be defined based on their inputs/outputs or in more complicated cases they can be a stack of previously defined models. When we define several models, we may need to compile few of them. That is why the compile option could be turned on or off for any model. If the model is chosen to be compiled, the user has options to select loss function and metrics for each output, as well as the optimizer for the model. In the case of stacked models, each part can be determined to be trainable or not. In the next chapter it will be explained through some examples.

The model inputs can be added simply by selecting from a list of all available input layers in the previous step. In this example, we have a vanilla sequence of layers with only one input. The model outputs will be added in the same manner, but each output needs a loss function, loss weight, and a list of metrics we are planning to compute during training. The loss function depends on the problem's logic and could be selected from a list of standard Keras functions. The loss weight determines the importance of one output in comparison with the others in the same multi-output model. Finally, metrics are some functions that aim to be calculated in each iteration and could be helpful for the user while deciding about the goodness of the model. In usual classification problems "accuracy" is the most common metric.

Lastly, each compiled model needs an optimizer that plays a crucial role in the training process. DLHub has a list of famous optimizers, and each one is fully customizable by its parameters in a graphical way. In many basic or intermediate cases, the optimizer type and its learning rate are the most important parameters to set. By default, Adam optimizer [27] with learning rate 0.001 is selected for the model. In many problems it is a good choice to start with, but when the user tries to find the best hyper-parameters during the fine-tuning of the model it would be a good idea to check at least various optimizers with several learning rates.

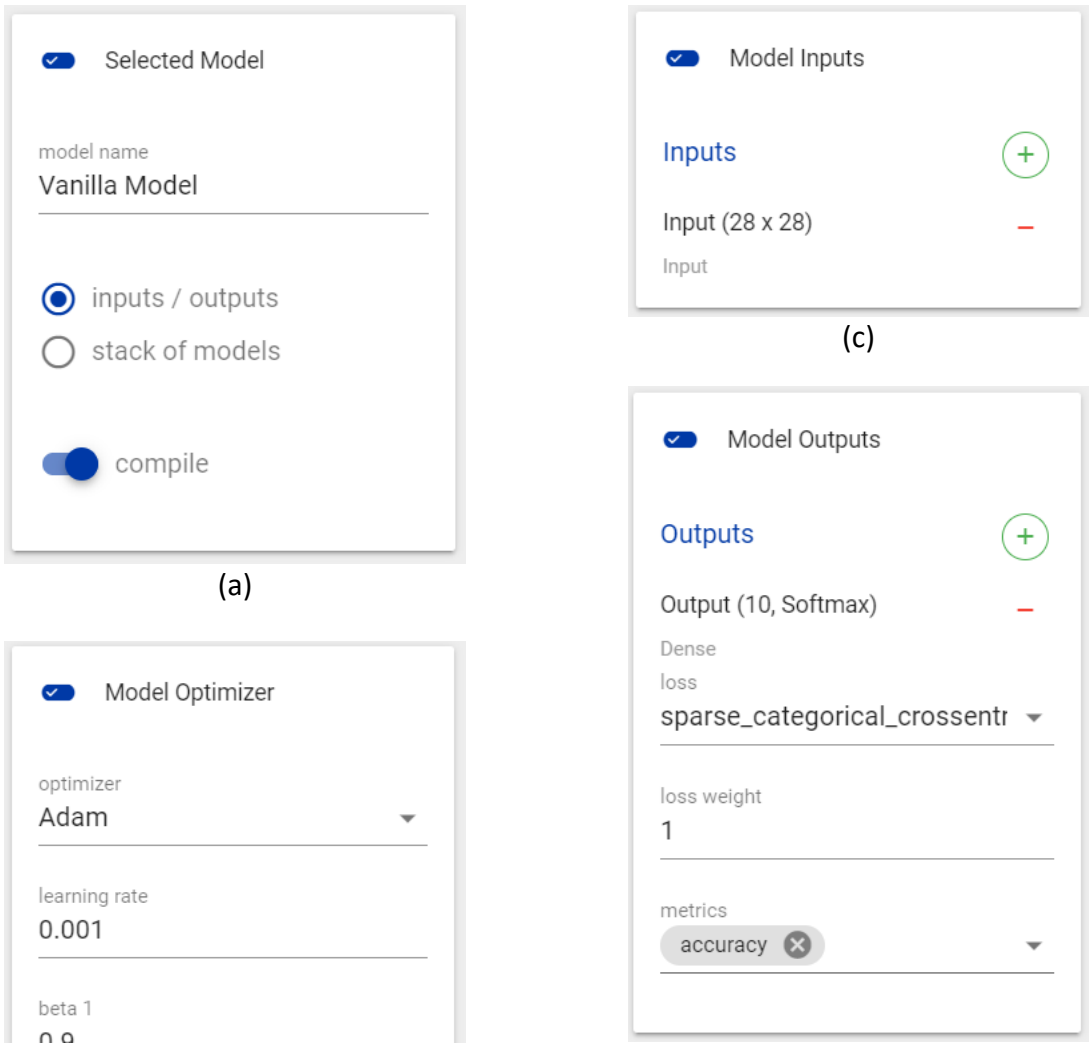


Figure 4.16. Defining models.
(a) Model structure. (b) Model optimizer. (c) Model inputs. (d) Model output configuration.

4.10 Organizing Runtime Tasks

The fourth step is to make a list of tasks to run. As Figure 4.17 shows, there are four main tasks for each model: fit (train) the model, evaluate the trained model on test data, make predictions using the trained model, and finally, assess the quality of the predictions. The tasks can be added, removed or modified based the special needs of every problem. It is important to keep in mind that here we are not running the tasks, but we arrange them and customize all the details, and put them ready to be ran in the final step. This is the way Tensorflow works for running heavy computing problems.

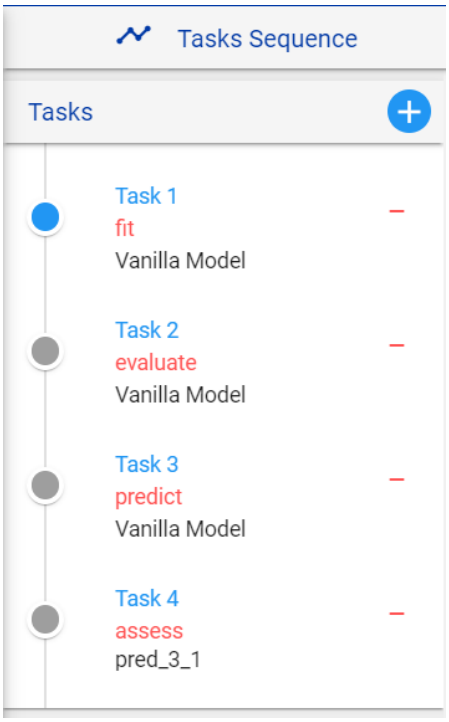


Figure 4.17. Runtime tasks sequence.
In a usual problem, fit, evaluate, predict, and assess are used in the mentioned order.

After adding a new task, an empty pane appears to select the task function. “fit”, “evaluate”, and “predict” tasks directly work with a model. The corresponding model can be chosen from a list of models defined in the previous step. After selecting the function and the model, a new pane appears that provides the user with some options to customize the function’s parameters. Figure 4.18 shows this pane for the “fit” function. Bach size and the number of epochs can be changed as important hyper-parameters. One can choose if it is necessary to shuffle the dataset after each epoch. Finally, if we already have a dataset for validation, it could be selected from a list. Otherwise, a slider component shows the portion of the data we want to be used as the validation dataset. In this example we choose 10 percent of the train dataset to be used for validation. In the runtime, Tensorflow randomly extracts this 10 percent.

Callbacks could be very helpful during training a model, especially if training takes a long time to finish. Tensorflow offers several callback functions, and DLHub adopts two of the most important ones. Figure 4.19 shows that one may turn these callbacks on or off. Early Stopping [28] callback gives the opportunity to choose a big number of epochs for training, and then monitoring the process to see if training gets better results in each epoch. A metric is chosen to be monitored and if, based on some definitions, the selected metric does not get better after some consequent epochs thee training process will be stopped automatically. In some references using Early Stopping callback is mentioned as one of the best experiences to follow in real problems [22].

Model Checkpoint callback is another important implemented option in DLHub. Determining a local place to save the model, and a metric to monitor during training, we can decide to save the best model on local machine. It is very useful as the models can suffer overfitting to the training data during long iterations. The saved model can be used in other programs, and especially for deployment purposes. Usually, validation loss is the best metric to monitor while determining the best model in the process.

Figure 4.18. Customizing the “fit” function

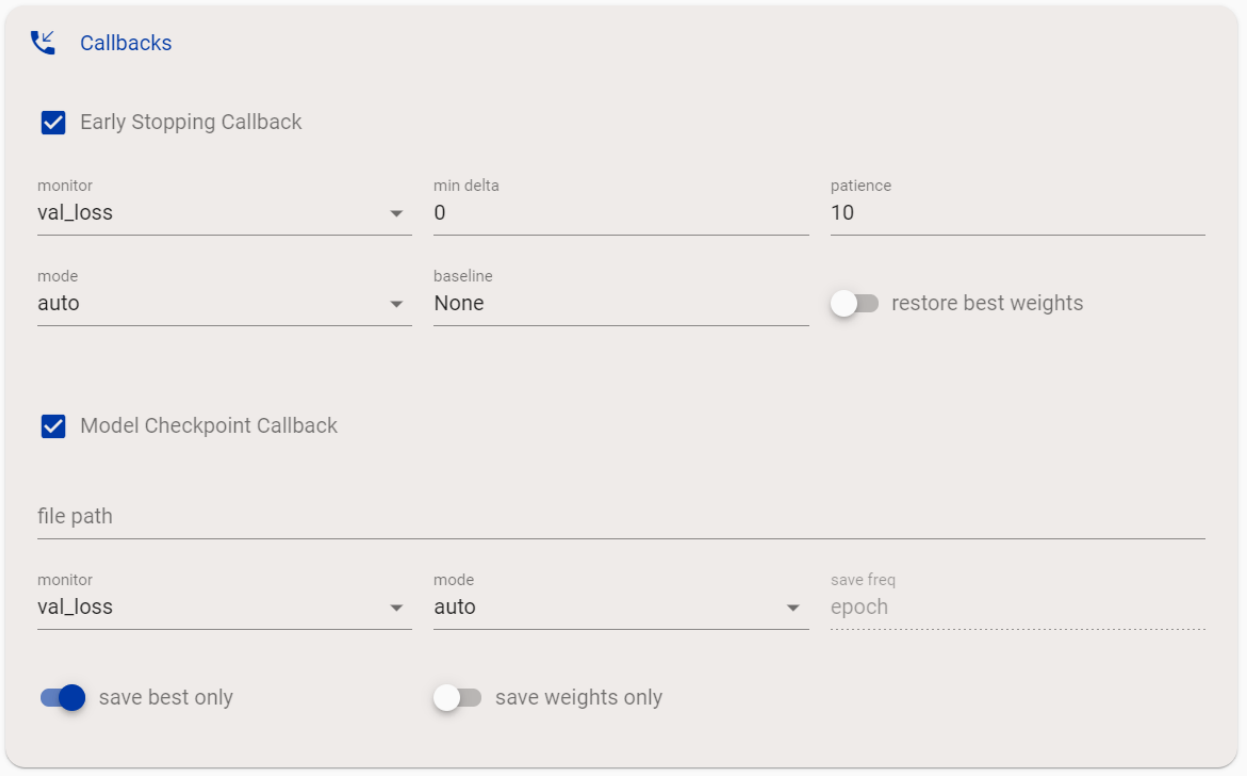


Figure 4.19. Adding callbacks to “fit” functions

In each task, the user must select the model, and its inputs/outputs from some lists which are populated based on the previous steps, but depending on the tsk function, there may be additional parameters to set. Figures 4.20 and 4.21 show it for “fit” and “predict” functions. In this example, the vanilla model is a single input single output model. “fit” function receives the train dataset in both parts, and as it is shown the dimension of the features and the labels are

demonstrated correspondingly. In the output section, and specifically for classification problems, one may modify the weight of each class. Here, we keep all the classes with the same weight, as the Fashion MNIST dataset has the same number of instances for each class, but in unbalanced datasets it could make improvements in the results.

On the other hand, the “predict” function only receives test dataset in the inputs, and the model will predict the corresponding labels for these features. Here, the features consist of 10,000 instances of 28 by 28 images. The predicted labels must be saved in a new dataset, which in this case DLHub has put the name “pred_3_1” which means “predictions of first output in the third task”. This name has been generated automatically, and it could be modified. The other important configuration here is the argmax operation. In classification problems, the Softmax activation function generates probabilities for all classes, but the labels must contain the class number. That is the case where “use argmax” option must be selected. On the other hand, in regression tasks the model directly predicts the output value and this option must be kept deselected.

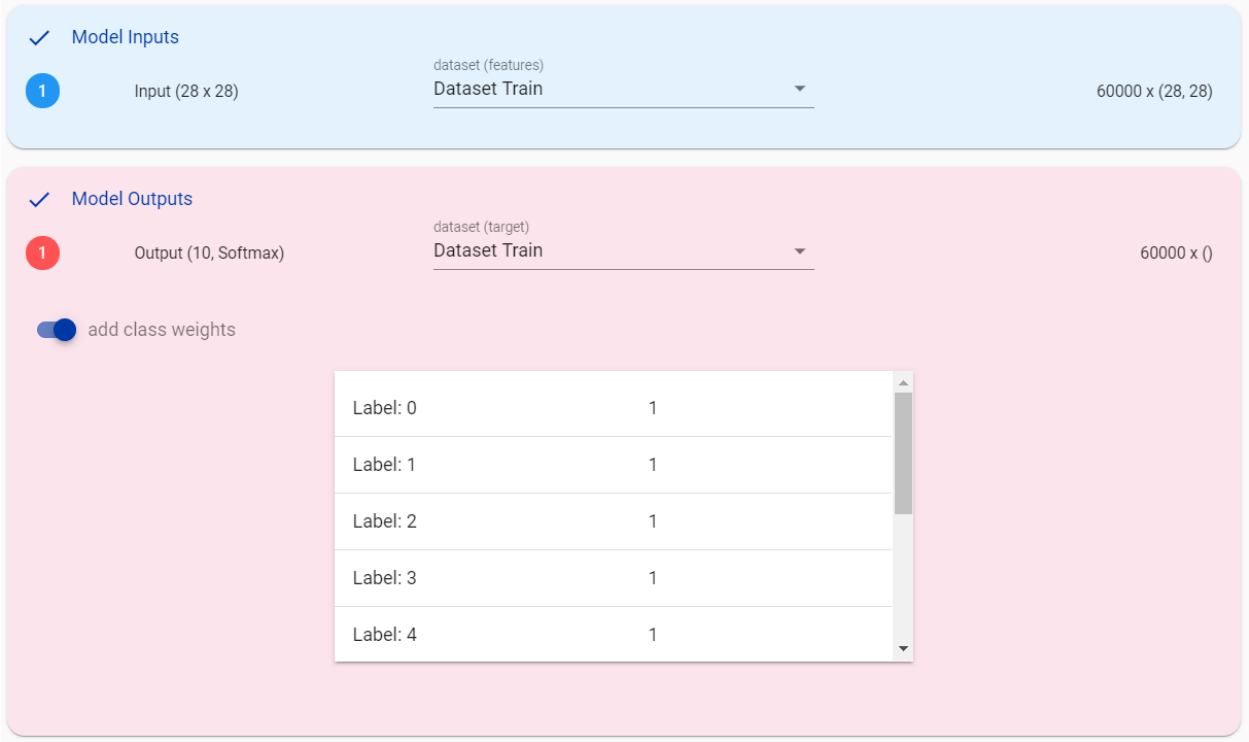


Figure 4.20. Inputs/Outputs of the “fit” function, and the class weights for classification

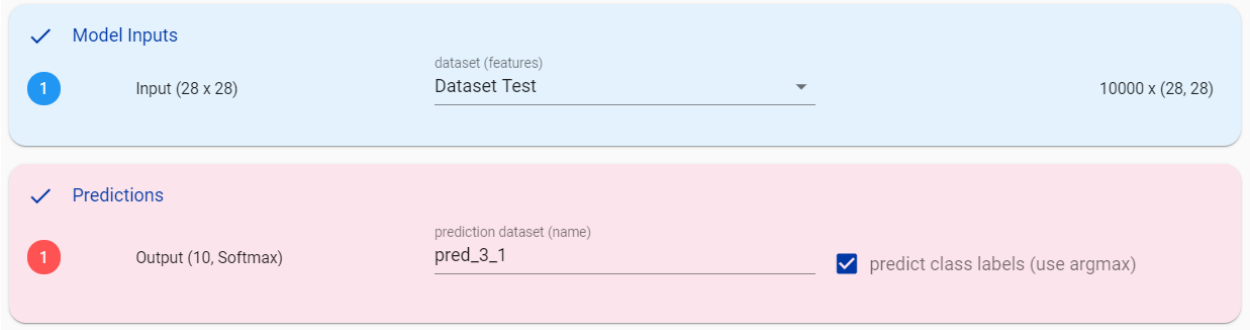


Figure 4.21. Inputs/Outputs for the “predict” function, and the argmax operation for classification

The “assess” function aims to evaluate the quality of the predictions, and this evaluation is done through a comparison between the real labels and the predicted ones for the same set of data instances. Therefore, configuring “assess” function is quite simpler than other tasks. As figure 4.22 shows, it is only necessary to define the problem type as well as two datasets to be compared. The problem type generally could be “classification” or “regression”, and it tells DLHub how to compare the datasets, and how to present the comparison. In the case of classification, a Confusion Matrix would be a good representation for the quality of the predictions, and regressions a scatter plot beside some calculations would be a good idea. Here we must choose two datasets. The first one contains the real labels, which in this example it is the test dataset, and the second one contains the predictions, which is the dataset created by the “predict” function. In this example it calls “pred_3_1”. If the model has multiple outputs, the “predict” function will generate multiple prediction datasets, and we will need to add an “assess” task for each output. By the way, they may be compared differently, i.e. one output may be classification, while the other one is a regression.

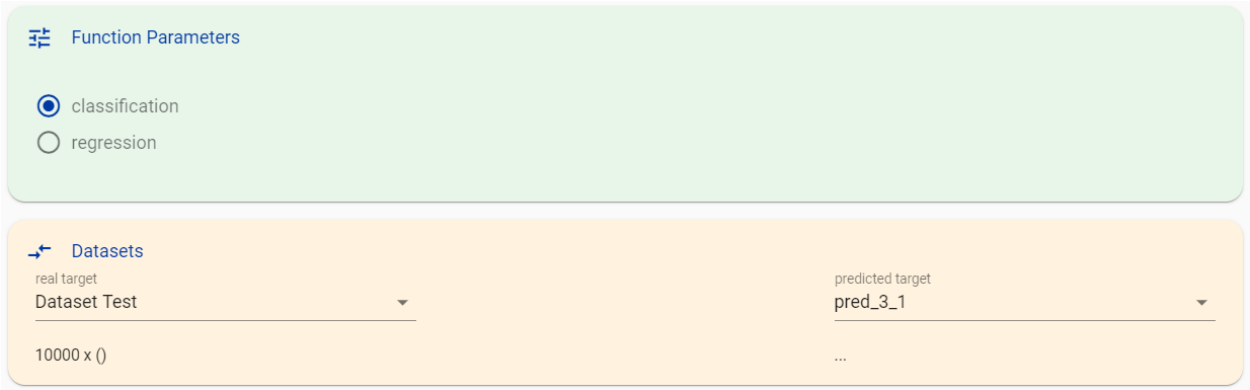
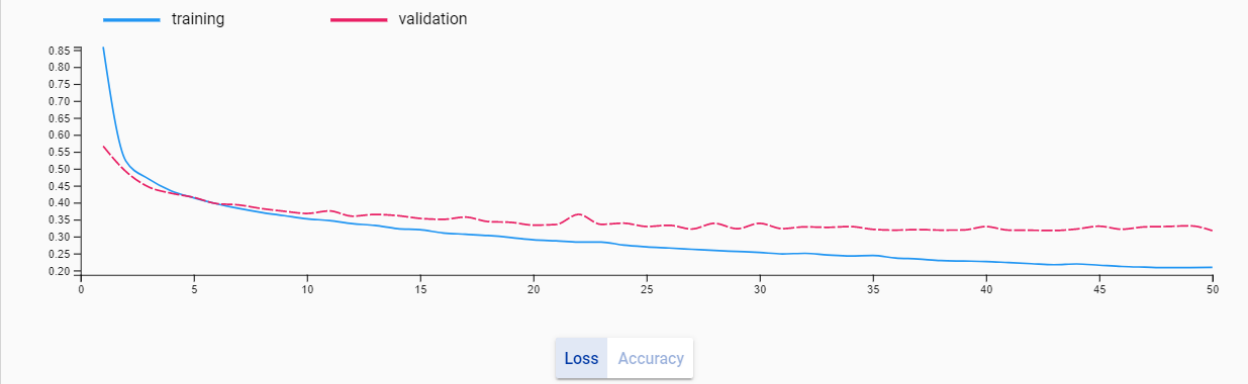


Figure 4.22. Configuration of Assess tasks

4.11 Running Tasks

Finally, by navigating to the last step, the user can run each task and analyze the results. Each task, based on its characteristics, generates a visual and user-friendly way to represent the results. The “fit” task shows a dynamic graph which plots the loss as well as all the other metrics in real time while the model is training. Some interactive buttons under the graph make it possible to lively change the metric to show in the plot. In addition, an interactive table shows the numerical values for all metrics. If any validation dataset has been added in the train task, or if the validation split value is greater than zero, all metrics are calculated for both train and validation datasets. Figures 4.23 and 4.24 show how DLHub presents the results for the “fit” and “evaluate” functions. Figure 4.23 demonstrates that the model has been overfitted to the train dataset.



epoch ↓	loss	val_loss	accuracy	val_accuracy
50	0.20906416133598046	0.3169977565606435	0.92535186	0.89416665
49	0.2083439959420098	0.3314799865086873	0.92685187	0.88516665
48	0.20791963708621483	0.329448069135348	0.9262593	0.8831667
47	0.20951513587324708	0.32819289962450665	0.9253889	0.8865
46	0.21145988172954983	0.3212357411781947	0.9248704	0.888

Rows per page: 5 1-5 of 50

Figure 4.23. Training the model in DLHub

loss	accuracy
0.3284935712814331	0.8868

Figure 4.24. Model evaluation in DLHub

Figure 4.25 shows the results of the “prediction” task. In a classification problem, the class label as well as Softmax probabilities are shown in the figure. Because we have used an argmax operation in the prediction task, the class label is the zero-based index of the maximum of Softmax probabilities. All the results can also be saved in a Python pickled file in the client’s machine, and they could be used in Python codes for further analyses.

Figure 4.26 shows the interactive confusion matrix created for the “assess” task. It can be chosen to show “counts” or “percentages” in the graph. This figure clearly shows that our vanilla model trained on the Fashion MNIST dataset has the best accuracy in classification of class 1 (Trouser). In 97.3% of the images containing a trouser, the model classifies the image correctly. In very few cases the trousers have been confused with T-shirts, pullovers, dresses, coats, or shirts. On the other hand, class 6 (Shirt) had the worst accuracy. In many cases the model could not distinguish a shirt from T-shirts, pullovers, dresses, and coats, which seems very logical to a human mind.

Figure 4.26 also shows another quality assessment of the predictions. DLHub generates a table containing the real and predicted labels for all test data instances. In the case of correct prediction, the cell is colored green, and for incorrect predictions, it has painted as red. This way, users who may need to know which exact data instances had an incorrect prediction, could explore the results simply. It may help the investigator to improve the architecture of the model, or to change some hyper-parameters.

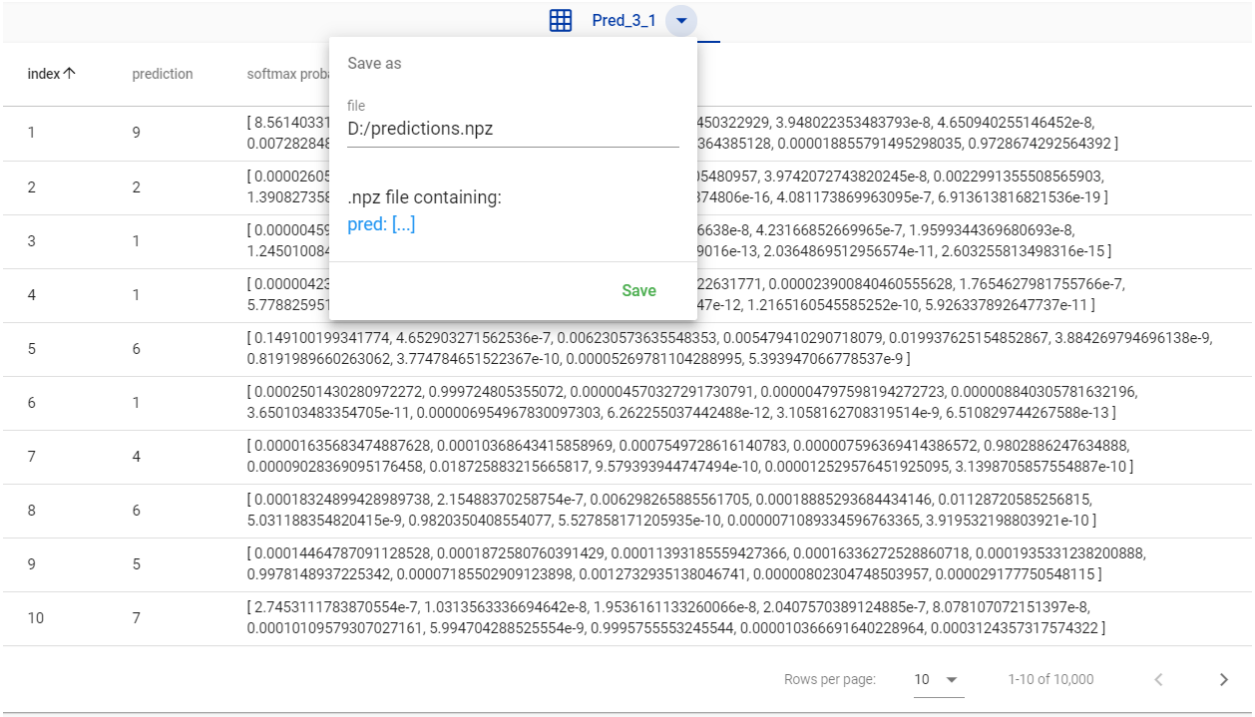
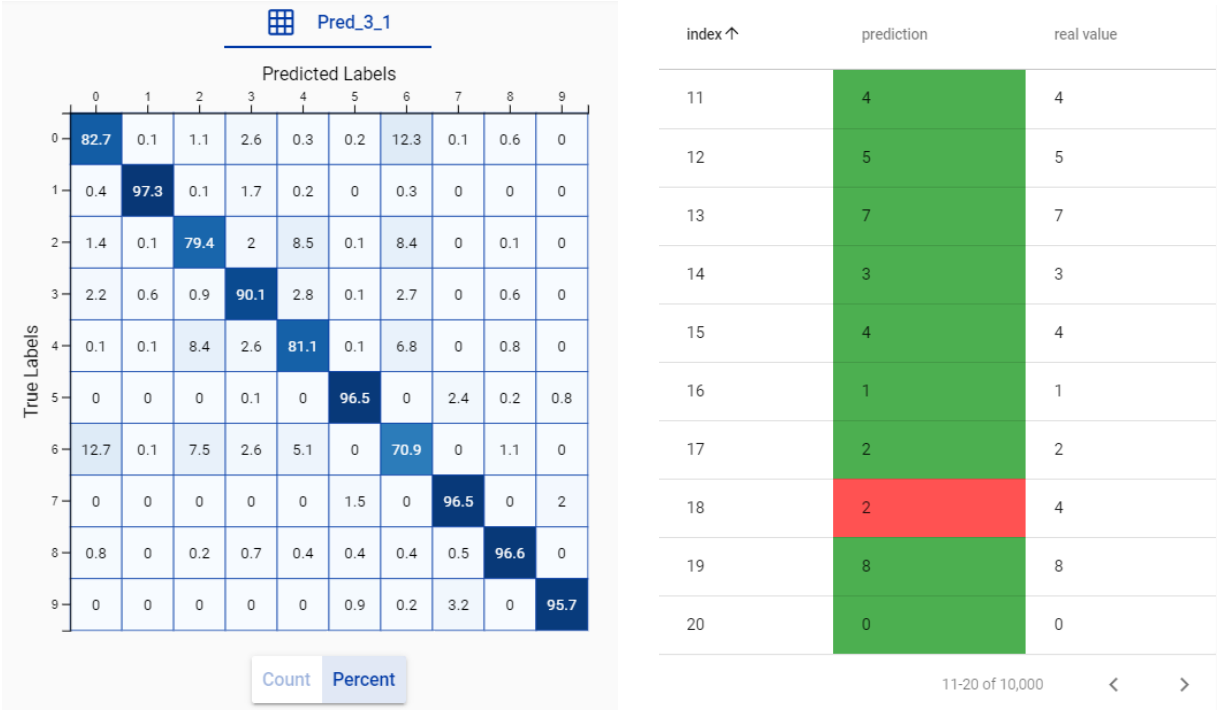


Figure 4.25. Model predictions in DLHub



(a) (b)

Figure 4.26. Quality assessment of the predictions.
(a) The confusion matrix presentation. (b) Comparison of the real and predicted labels.

Chapter 5

Advanced ANN Architectures

In the previous chapter our methodology for implementing deep learning algorithms has been explained through a vanilla neural network classification on the famous Fashion MNIST dataset. This chapter briefly adds few more examples to show how to implement more advanced ANN architectures in DLHub. Here, we are not going to explain the complete path in each problem and only the steps which are different with a usual case are shown. All these examples use the same dataset to emphasize the fact that the same problem may be solved via different architectures with different points of view. All neural networks explained in this chapter are extremely well studied with proved efficiency and applications in the literature, and here we are focusing on their implementation in our proposed graphical programming environment. Therefore, here we are not going deep to their concepts.

5.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks are among the most useful ones in image processing, and nowadays, they form the skeleton of computer vision. Convolutional and Pooling layers are the basic building blocks in CNNs. Figure 5.1 shows how to implement a simple CNN classifier on our dataset. Our inputs have two dimensions as they are grey-scale images, but convolutional layers expect to receive a three-dimensional input, because a usual RGB image consists of three channels of the same size, corresponding to red, green, and blue components. Therefore, in this example we need to use a Reshape layer to add a third dimension with unit value to the images. Then, a sequence of Convolutional or Pooling layers are employed to extract important features from the input. One needs to try several arrangements of these layers to find an optimum one. This feature extraction part is followed by a sequence of fully connected layers for classification. Here, we just use one Dense layer followed by the output layer. Again, the Dense layer expects one-dimensional inputs while the CNN feature extraction part gives three-dimensional outputs. As a result, a Flatten layer in needed in between.

Table 5.1 briefly shows classification results after 50 training epochs. The general classification accuracy for the training, validation, and test datasets are 0.96, 0.91, and 0.9 respectively. Again, it is obvious that the model is suffering overfitting to the train data. By the way, comparing these numbers with the classification results for the vanilla ANN architecture discussed in the previous chapter, shows a little improvement in the accuracy. This improvement was quite expectable as CNNs first extract spatial features from the input images by applying several filters, and the classification process is being done on these extracted and more informative features, while vanilla ANNs perform everything directly on the raw images. It is worth mentioning that the convolutional network used here is not the best and the most optimum one, and we tried to use a CNN as simplest as possible just to explain the main idea. Moreover, one may use some regularization methods or add a Dropout layer in the classification part to prevent the model overfitted to the train data.

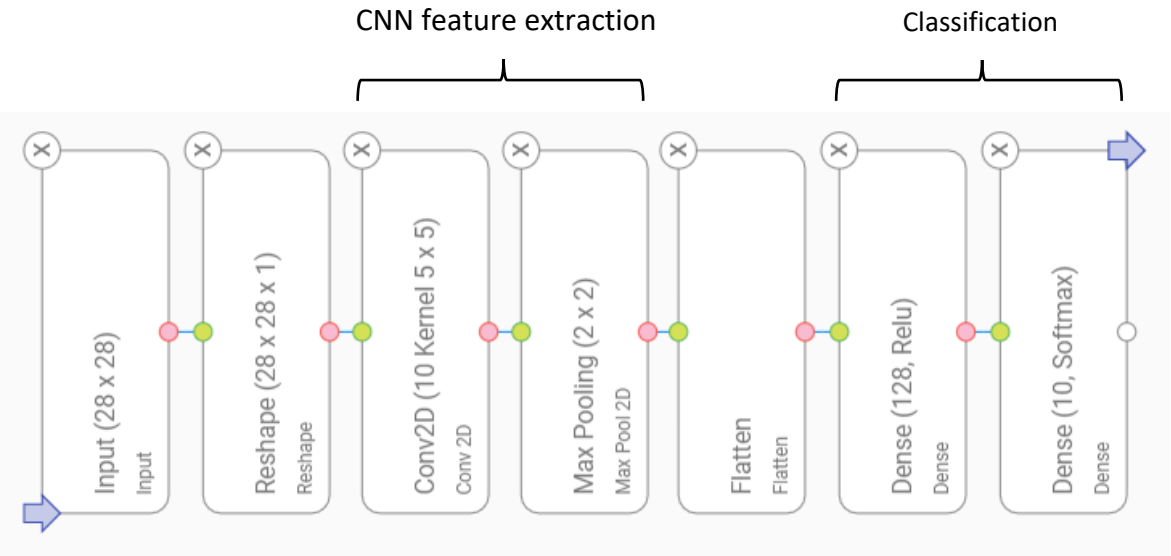


Figure 5.1. CNN classifier architecture on Fashion MNIST dataset

Table 5.1. CNN classification Results on Fashion MNIST after 50 iterations

Training				Evaluation	
loss	val_loss	accuracy	val_accuracy	loss	accuracy
0.1298698	0.2942695	0.96	0.91	0.2929258	0.90

5.2 Autoencoders (AEs)

Autoencoders are among the most important and useful techniques in representation learning, also called as feature learning. An autoencoder is structured as the stack of two more basic neural networks: Encoder and Decoder. In the most usual situation these two basic networks are mirror-flipped of each other, which means AEs often have symmetrical architectures. Encoder is the network that starts with the input layer and contains some hidden layers such that their number of neurons is monotonically decreasing. The output of encoder is called the latent layer. Decoder starts from the latent layer and passing the same series of hidden layers finishes to the output layer with the exact size as the encoder's input.

During the training process, data features are fed into both input and output layers. It means that an autoencoder receives a set of features, and after few consecutive transformations tries to reconstruct the same features. The key point is the decreasing number of neuros in consecutive layers in the encoder network. This way, the latent layer will be an encoded representation of the input with smaller dimension. The decoder network actually receives this encoded version and tries to reconstruct the original data. That is why autoencoders are very useful in dimensionality

reduction or feature selection. Here, a classification example on Fashion MNIST dataset using AE dimensionality reduction explained.

In DLHub, dataset preparation for training autoencoders is a little bit different from previous examples, as an AE receives the features in both input and output layers. Figure 5.2 demonstrates how to make the datasets for this problem. Here, we need to make four datasets: two for autoencoder, and two for classification. As shown in the figure, classification datasets receive the scaled features and the labels in their socket “x” and “y” respectively. On the other hand, AE datasets get scaled features in both sockets.

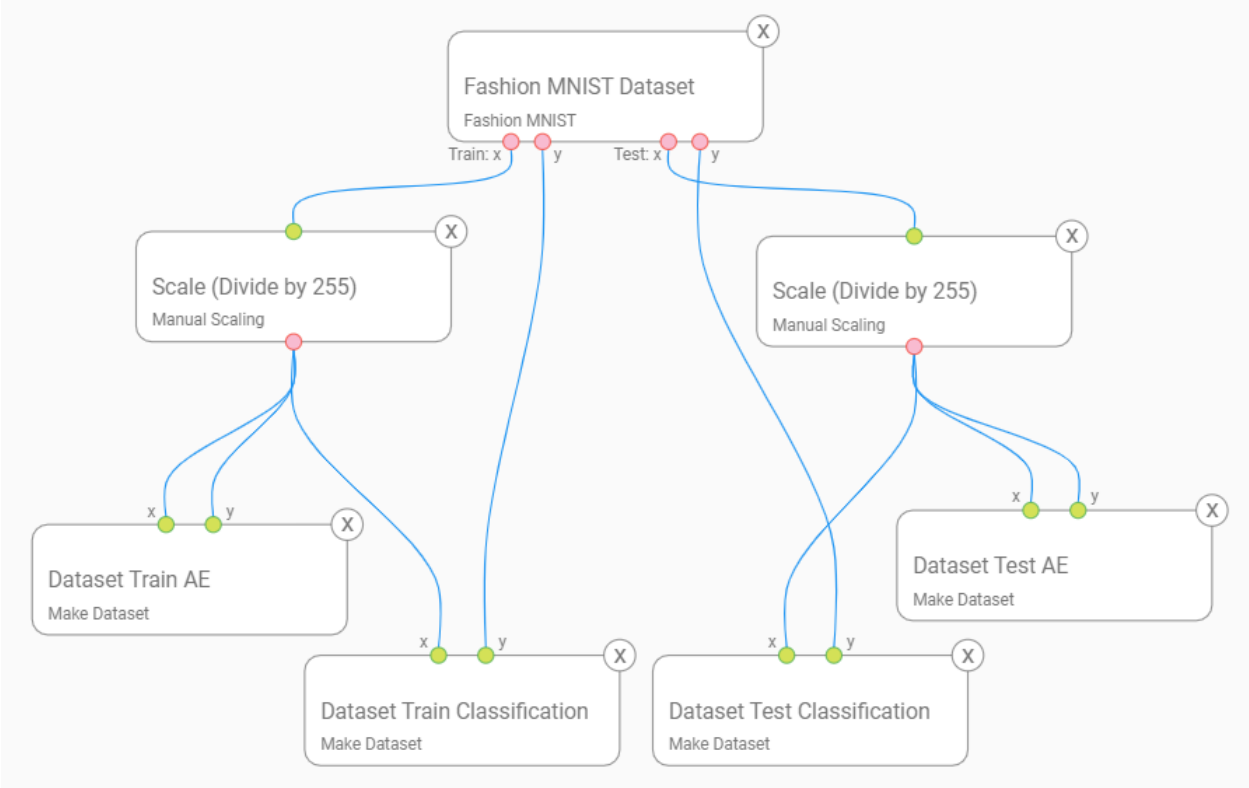


Figure 5.2. Dataset preparation for classification using autoencoders

Now, it is time to implement the layers architecture. Figure 5.3 shows that in this example we need to define three separated sequences of layers for encoder, decoder, and classifier. The encoder starts with a two-dimensional image followed by a Flatten layer to re-arrange it into one dimension, a hidden Dense layer, and another Dense layer as the latent space. We choose ReLU activation function for both Dense layers. Implementation of the decoder is quite simple. It is just needed to flip the layers in the encoder. Decoder will receive the latent layer, so it will start with an Input layer with the same dimension as the latent layer, followed by two Dense layers with dimensions equal to the hidden layer and the Flatten layer in the encoder. Finally, a Reshape layer re-arranges the data into two-dimensional reconstruction of the original input. One important technical detail here is that the last Dense layer in the encoder must have Sigmoid activation function to force the network to generate all the reconstructions with values between 0 and 1.

This is quite important, because the original input of the AE will be a scaled image with values in this range.

Finally, the classification part must start with an input equal to the latent layer. It is followed by a hidden and the output layers. The idea here is that we first must train the autoencoder such that the encoder learns to extract the most important features (with lower dimensionality), and the decoder learns how to reconstruct the original features (image) from this low dimensional representation. Then the encoder can be used to transform all images into their lower dimensional representations and the classifier tries to perform the classification process on this new data. In many cases, classifier would perform better as usually the encoded data is a more informative representation than the original one.

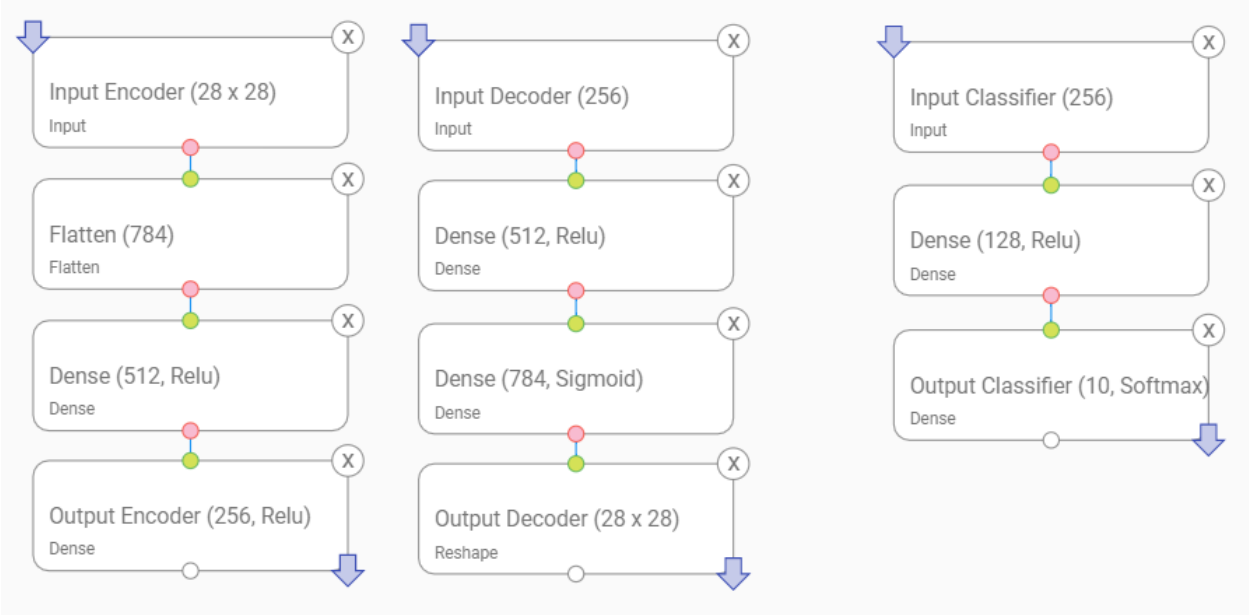
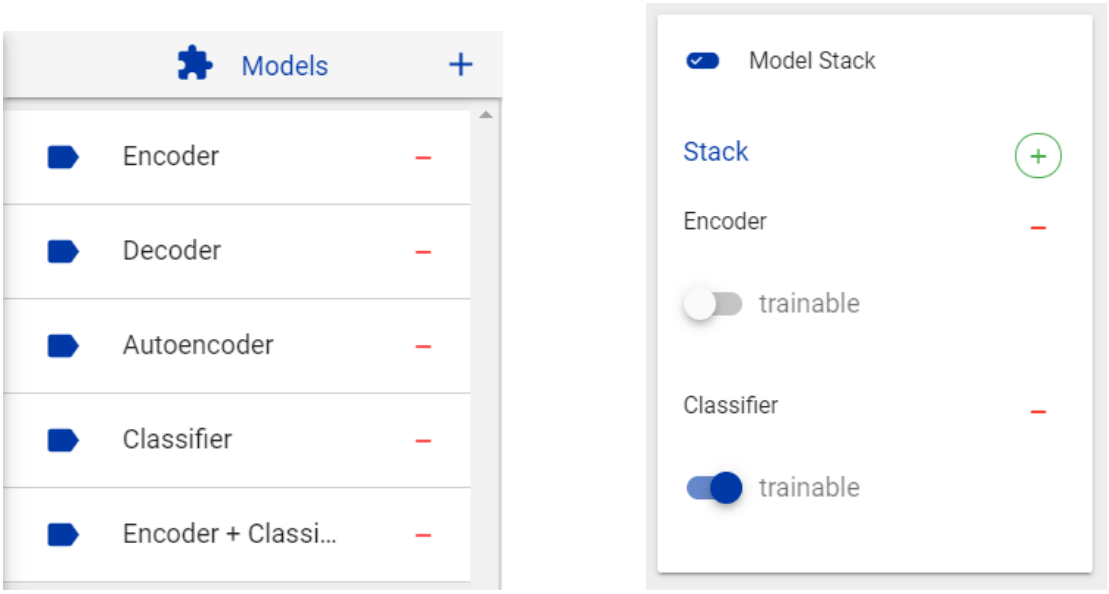


Figure 5.3. Autoencoder classification architecture on Fashion MNIST dataset

Figure 5.4 and table 5.2 show the models and their configurations in this problem. The encoder, decoder, and classifier models could be defined with their input/output, but they will not be trained individually, so it is not needed to define loss functions or optimizers for them. In other words, these models are not get compiled individually. Two more complicated models are needed as stacks of these basic models. The autoencoder is the stack of encoder and decoder, while both are trainable. Here, our AE needs “mean squared error” as the reconstruction loss function. Besides, we need to stack the encoder and the classifier, while only the classifier is trainable. This model needs “sparse categorical cross entropy” as the loss function for the classification task.



(a) (b)

Figure 5.4. Models in a classification problem using autoencoders.
(a) List of all added models. (b) Freezing a part of the stacked classifier.

Table 5.2. Details of the models in a classification problem using autoencoders

Model	Type	Compiled	Loss Function
Encoder	Input / Output	No	-
Decoder	Input / Output	No	-
Autoencoder	Stack • Encoder (trainable) • Decoder (trainable)	Yes	mean squared error
Classifier	Input / Output	No	-
Encoder + Classifier	Stack • Encoder (freezed) • Classifier (trainable)	Yes	sparse categorical crossentropy

As discussed earlier, we first need to train the autoencoder using the dataset specifically prepared for that, and then train the stacked "encoder + classifier" on the other dataset. Table 5.3 demonstrates the classification results in this example. Although we did not get better results even from the vanilla architecture, very probably more sophisticated convolutional autoencoders would give better results.

Table 5.3. Classification Results on Fashion MNIST using autoencoders, after 50 iterations

Training				Evaluation	
loss	val_loss	accuracy	val_accuracy	loss	accuracy
0.2905934	0.3484798	0.89	0.87	0.3647707	0.87

5.3 Variational Autoencoders (VAEs)

Variational autoencoders was introduced in 2013 by Diederik Kingma and Max Welling [29], and nowadays, they are among the most common generative models in deep learning. Generally, VAEs are very similar to vanilla AEs that start with a data instance and after reducing the dimensionality in consecutive layers, they make a latent representation of the data, and finally through a reverse transformation they try to reconstruct the input data. The main magic in VAEs happens in the latent layer. In VAEs it is assumed that the input data has some unknown statistical distribution, and the encoder learns to transform this arbitrary distribution into some pre-defined distribution, usually a multivariate Gaussian one. Then, the decoder can reconstruct the input data via the reverse transformation. The interesting point is that, if the decoder back-transforms any random sample from the latent Gaussian distribution, it will not reconstruct the exact input data, but it will generate some new data with exactly the same unknown distribution as the input dataset. Therefore, a VAE trained on a specific dataset can generate countless new instances that share the same statistics with the original dataset. That is why they are called as generative models.

According to the importance of VAEs in deep learning, DLHub also has built-in functionalities to implement variational autoencoders and run generative models. Figure 5.5 shows the dataset preparation step to generative new Fashion MNIST-like data using VAEs. In such generative problems, only the features of the train dataset are needed. As always, the features are scaled by dividing to 255 and then, a dataset for training an autoencoder is prepared where both the inputs and outputs contain the scaled features. Figure 5.6 shows the implementation of a very simple VAE architecture in DLHub. Here again, two separated architectures are implemented to the encoder and the decoder. The encoder starts with the two-dimensional image data followed by a Flatten layer and a couple of Dense layers. The latent layer here consists of two parallel Dense layers corresponding to the “mean” and the “variance logarithm” of the latent multivariate Gaussian distribution. DLHub has added a customized Sampling layer which does not exist in Keras standard API to simplify the sampling process from the latent normal distribution. This layer specifically can be used in VAEs and expects to receive the two latent layers discussed above. On the other side, the decoder simply starts with an Input layer with the same dimension as the latent sample, and back-transforms everything toward the input image. The last layer of the decoder network is a Reshape layer to change the dimensions to 28 by 28 as the Fashion MNIST images. Moreover, the Sigmoid activation function on the last Dense layer, ensures the network to generate values between zero and one.

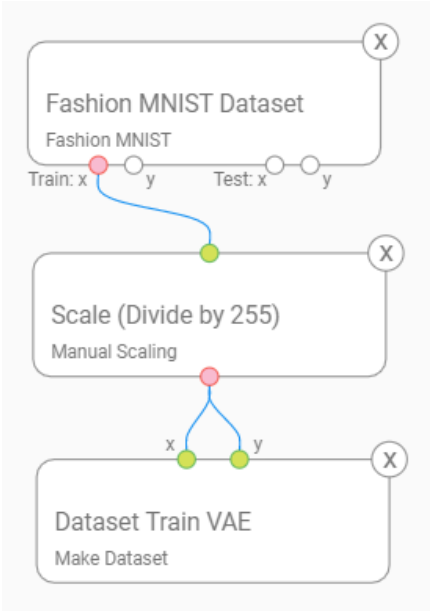


Figure 5.5. Dataset preparation for a generative problem using VAE

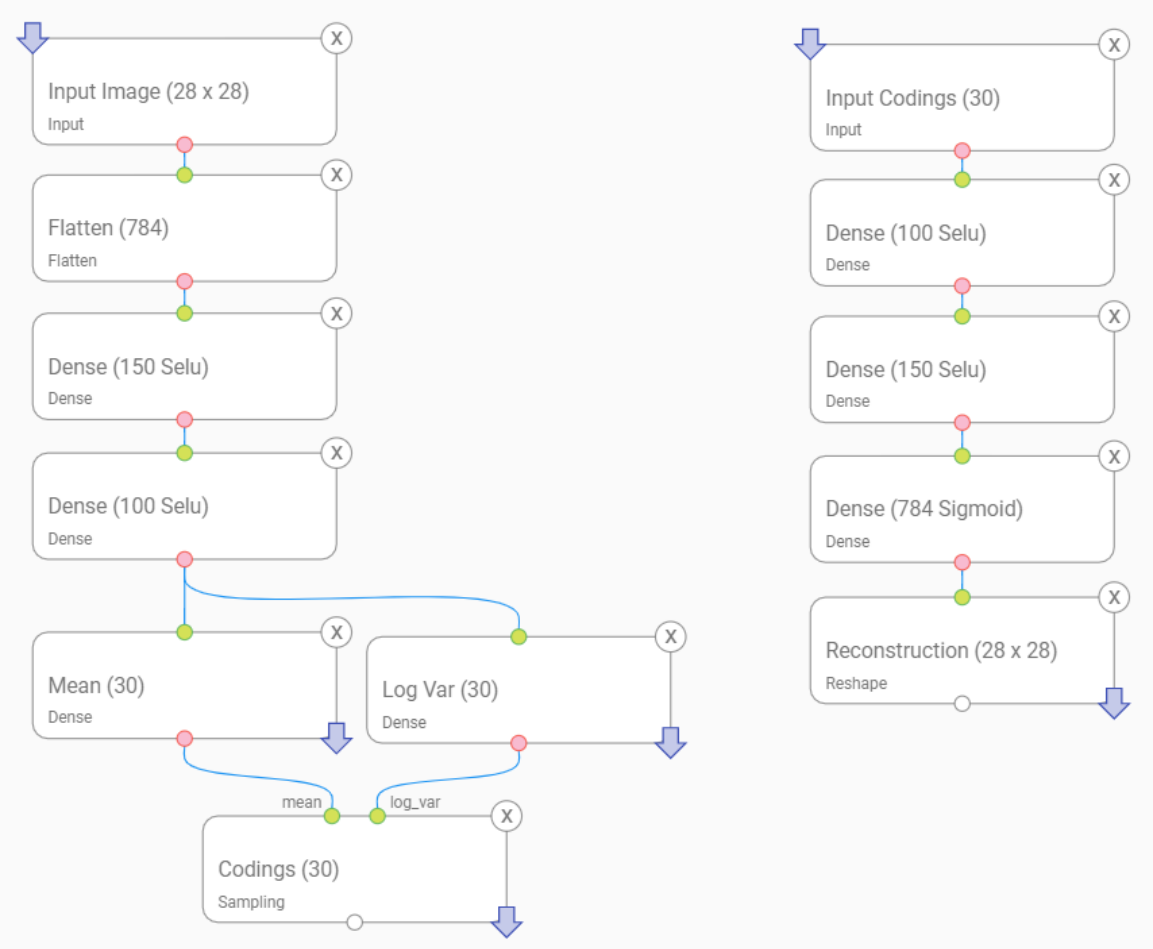
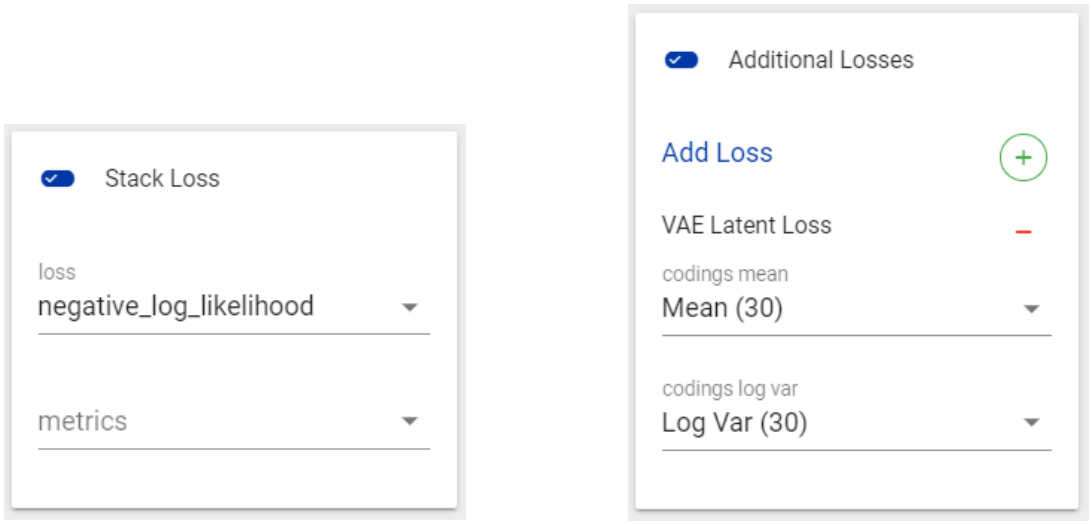


Figure 5.6. VAE architecture on Fashion MNIST dataset

When compiling the model for variational autoencoders, few details must be considered. In VAEs the model's loss function consists of two parts. One part is the reconstruction loss, but this time as the reconstructions are not the exact input values, instead of mean squared error, usually negative log likelihood is used. The second part of the loss function is the latent loss, which is the KL divergence between the target distribution (i.e. Gaussian) and the actual distribution of the codings. Figure 5.7 shows how to add the two loss function parts into the model definition in DLHub. VAE latent loss is an additional built-in option in the software which lets implementing VAEs as simple as possible.



(a) (b)

Figure 5.7. Loss functions for VAE.
(a) Reconstruction loss function. (b) VAE latent loss function.

In a generative model problem, two main runtime tasks are needed. First, the model must be trained using the prepared train dataset. Second, the "predict" function of the decoder model must be used to generate new data samples, but this time the inputs must be samples from the Gaussian distribution corresponding the codings (latent). Figure 5.8 shows that in DLHub for the inputs of the "predict" function the user instead of selecting a prepared dataset, can use "random normal" option which automatically samples from a multidimensional Gaussian distribution. The application automatically uses the latent size, but the number of new generations must be determined by the user. Figure 5.9 provides some examples of the artificially generated data based on the Fashion MNIST dataset. It is shown that the model has generated successfully some pullovers, trousers, sneakers, and ankle boots. However, in some cases the generated image would be something in between, and this is an interesting property of VAEs. The figure brings two examples of such cases, where the generated image in one image is something between a T-shirt and a trouser, and in the other one is something between a sneaker, a trouser, and a T-shirt. After generating a lot of images with the same model, we got that such a simple model does not generate good sandals, and regarding the complexity of the sandal images in comparison to the other classes, it was not a surprising result.

Figure 5.8. Inputs/Outputs for the “predict” function of a generative model

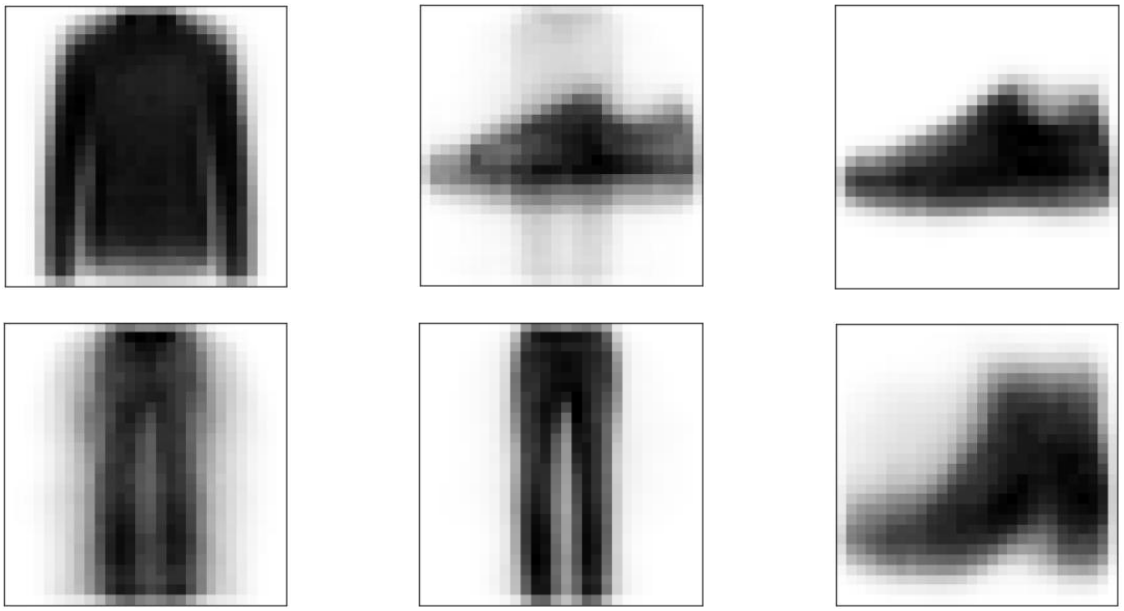


Figure 5.9. Generated images of VAE on Fashion MNIST dataset

5.4 Generative Adversarial Networks (GANs)

GANs are among the most sophisticated and advanced models in deep learning introduced in 2014 by Ian Goodfellow et al. [30], and currently are quite stable in AI industries. Like autoencoders, GANs also consist of two networks, but this time these networks instead of a direct collaboration, are competing against each other. Generator is a network that receives some noise (i.e. random numbers) and tries to generate some fake data as similar as possible to the data instances of a dataset. On the other hand, Discriminator is a network that receives a combination of real and fake data and tries to classify them correctly. In the first iterations, the generator is not expert in generating fake data and mostly generates some data that simply get classified by the discriminator. During the competition the generator learns more and more about the

important patterns in the dataset such that if the training process converges, the discriminator would not be able to detect fake data from the real one.

Dataset preparation for GANs is somehow similar to VAEs. As shown in the figure 5.10 GANs only need the features of the train dataset, and the only difference is that here in the dataset, the labels are not important, as finally all these data are considered as the real data instances. Figure 5.11 presents a simple implementation of the generator and discriminator networks. The generator starts with a noise with some predefined dimension, and after few hidden layers, ends up to a Reshape layer to generate 28 by 28-pixel images. Like in autoencoders, here also the Dense layer before the Reshape layer must have Sigmoid activation to force the network to generate values between zero and one. The discriminator network starts with a 28 by 28 layer, and after a few hidden layers it ends up to a one-neuron output layer. The output layer would have a value between zero and one corresponding to fake and real data respectively.

Table 5.4 explains the details in implementation of the necessary models. Generator is a model which would not be trained separately, so it does not need to be compiled. The discriminator will be trained periodically as a classifier in the training process. Therefore, it needs binary cross entropy as the loss function. The GAN model is the stack of the generator and the discriminator, while only the generator part is trainable. Both these models use RMSprop optimizer. The training process in GANs is a bit different with the other models discussed before. This time the discriminator and the GAN models are trained consequently on each batch of data, and the whole process repeats for each epoch. That is why training generative adversarial networks is much more complicated in comparison with other architectures.

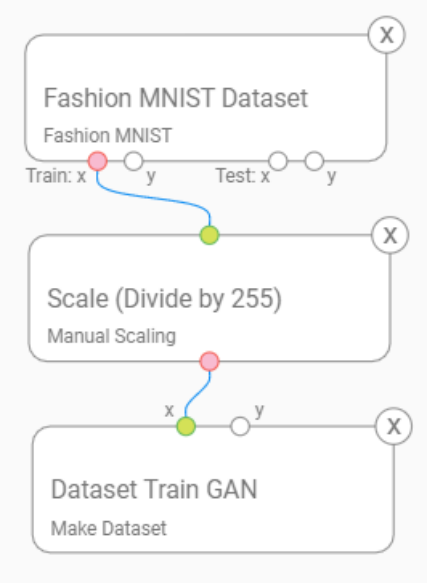


Figure 5.10. Dataset preparation for a generative problem using GANs

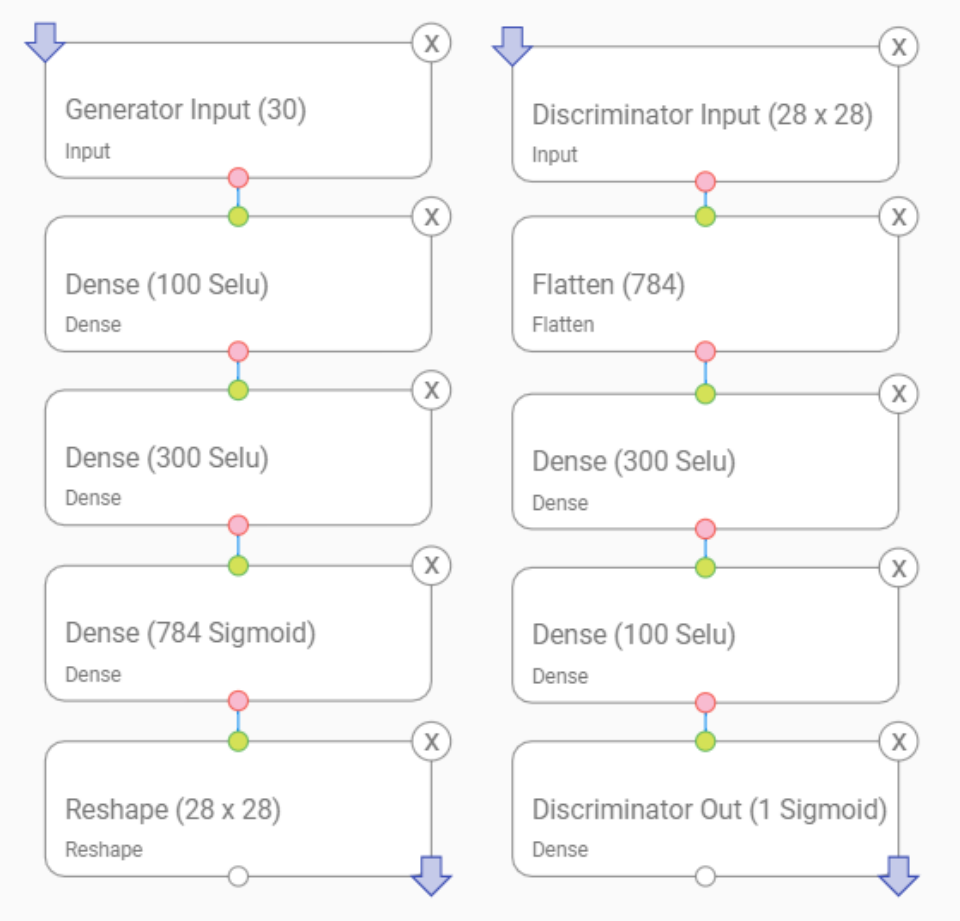


Figure 5.11. GAN architecture on Fashion MNIST dataset

Table 5.4. Details of the models in GANs

Model	Type	Compiled	Loss Function
Generator	Input / Output	No	-
Discriminator	Input / Output	Yes	binary crossentropy
GAN	Stack • Generator (trainable) • Discriminator (freezed)	Yes	binary crossentropy

In the runtime tasks, a “train GAN” task type has been added to DLHub, to simply let non-coder users working with their generative models. Figure 5.12 shows that this task needs to know the dataset to works on, the generator, discriminator, and GAN models, and finally the batch size and epoch numbers. In this example we just train the models for two epochs on batches of 16 images. Because the training process takes place for each batch and produces some training logs, and because DLHub keeps all the logs to show as tables or diagrams, for GANs it is very probable that

the big amount of training logs overflows the computer memory. That is why for this task type the user can also determine the maximum number of training logs to keep in memory.

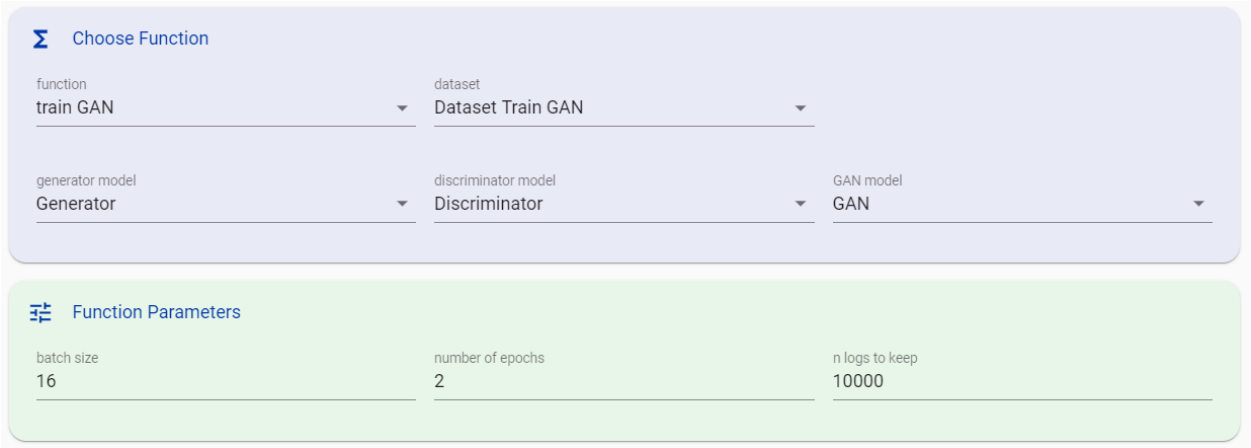
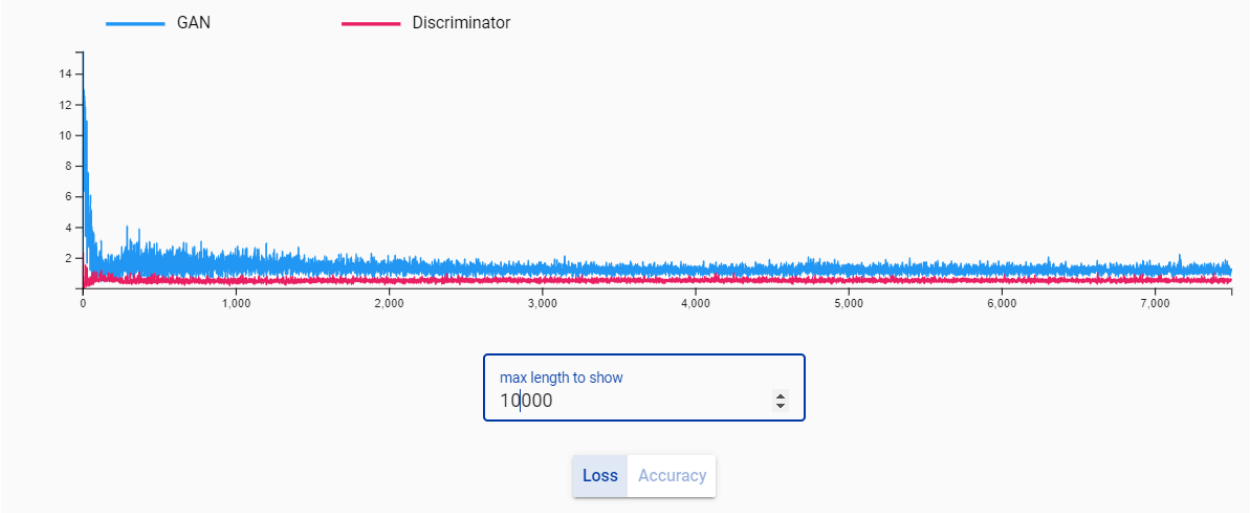


Figure 5.12. Configuration of a GAN task

Figure 5.13 shows the training process in action. In this case instead of showing two curves for the train and the validation datasets, DLHub presents the loss values for the GAN and the discriminator models. This figure somehow explains the fact that training GANs is generally much more difficult than other deep models, and any GAN does not necessarily converge during training. Figure 5.14 presents few examples of the generated fake data based on the Fashion MNIST dataset. It seems that the generator after only two epochs has learnt very good to imitate coats, T-shirts, and bags. The generated images in the figure are highly pixelized, and one important reason is that the model needs much more training epochs to learn the patterns.

Again, we must pay attention to the point that in this example we just have used Dense layers in the architecture of the generator network. In most of the computer vision problems convolutional neural networks produce much better results as they can understand existing two-dimensional patterns in the images. Although generative adversarial networks are still active research topics, and their dynamics are not completely understood yet, in the recent years very successful architectures have been proposed and their applications in real-world problems is already completely proved [31]–[33].



step ↓	epoch	gan_loss	dis_loss	gan_accuracy	dis_accuracy
7500	2	1.2892565	0.5324708	0.0625	0.71875

Figure 5.13. Training a GAN on Fashion MNIST dataset

Figure 5.14. Generated images of GAN on Fashion MNIST dataset

Chapter 6

Case Study

This chapter presents a case study on a real industrial problem. It consists of two Vapor Recovery Units (VRU) in an offshore petroleum platform. The main objective of VRUs is collecting hydrocarbons from low volume gases during the petroleum production operations. The changes in the temperature of the crude oil reserved in tanks causes a part of the oil to be vaporized. This vapor contains valuable hydrocarbons like propane, butane, and gasoline. Recovering this vapor not only increases the revenue of the company by rising the production level, but also has many benefits for the community. Preventing additional air pollutions, reducing the fire hazards, and preventing the emission of poisonous gases are some examples of such benefits. A Vapor Recovery Unit consists of different components like suction scrubber, gas bypass system, liquid transfer pump, lubrication system, and a compressor. As the compressor is the most important component in VRUs, usually its performance is continuously being monitored via a substantial sensor system installed in different parts of the unit.

6.1 Problem Description

Here, two VRUs that work in parallel in an offshore natural gas platform are monitored for a period of more than a year. For each unit, 172 sensors recorded different parameters of the system every 15 seconds. During the whole operation period, the working state of the equipment has been supervised and registered by maintenance engineers. Finally, for each unit the data is organized in two major parts. The first part contains about 2,400,000 registries of all 172 features, and the second part contains the operation time periods with the qualitative observations and descriptions about the working conditions. The features gathered in the dataset are of different types, each measuring conditions in separate components of the units. Table 6.1 lists some examples of the features recorded by the sensors in this case study.

All the events happened during this time period that cause a problem in the normal functioning of the unit are registered by the engineers. The events are categorized as normal stops, external causes, instrument faults, mechanical faults, electrical faults, scrubber high level, high suction pressure, and not identified causes. Finally, the maintenance engineers have divided all these events in three main categories: stand-by, external causes, and faults. Here, these three categories are used as the working state of VRUs with a small change. We have added electrical faults to the external causes, as any problem in the electrical system is not directly related to the VRU itself. Table 6.2 shows the distribution of the working intervals in the three determined main categories.

In this step, all the labels and counts are defined for time intervals and not for each observation. Therefore, after any normally working time interval, there must be an interval corresponding to an event such as “stand-by”, “fault” or “external cause”. Table 6.2 also demonstrates it clearly, as

there are 127 intervals where the equipment worked normally, and on the other hand, there are 107 intervals where the unit was paused, and 20 intervals where some kind of faults are detected in the vapor recovery unit. The objective in this case study is to develop deep learning models to be used in diagnosis of the faults in new unseen data. The next section starts the path with data cleaning, preparation, and preprocessing.

Table 6.1. Examples of the recorded parameters in VRUs

Parameter Types & Measurement Units	Examples
Pressure (kPa)	1st & 2nd stage compressor suction v-cone differential pressures, Compressor mechanical seal differential pressure, Seal oil supply differential pressure, Oil filter differential pressure, Cooling water supply pressure 1st stage gas cooler
Gauge Pressure (kPag)	1st & 2nd stage compressor suction pressure, 1st & 2nd stage compressor discharge pressure, Lube oil pumps outlet supply pressure, Lube oil supply pressure
Revolutions per Minute (rmp)	1st & 2nd stage compressor male rotor, 1st & 2nd stage compressor female rotor
Percentage (%)	1st & 2nd stage scrubber level, Oil reservoir level, 1st & 2nd stage scrubber drain level control valve, 1st & 2nd stage compressor spillback valve, 1st & 2nd stage cooling water return control valve
Acceleration (g)	1st & 2nd stage compressor casing acceleration,
Velocity (mm/s)	Compressor casing vibrations
Displacement (µm)	1st & 2nd stage compressor rotor suction side X displacements, 1st & 2nd stage compressor rotor discharge side X displacements
Temperature (ºC)	Temperature of compressor in different stages & locations, Temperature of motor, Oil reservoir temperature, Lube oil supply temperature pre-cooler

Table 6.2. Cross-Table of the working intervals and the labels in the first VRU

	Normal	Pause		Faults
		Stand-by	External Causes	
Normal	127			
Normal Stop		13		
External Cause			90	
Electrical Fault			4	
Instrument Fault				6
Mechanical Fault				3
Scrubber High Level				6
High Suction Level				1
Not Identified Causes				4
Total	127	107		20

6.2 Data Cleaning & Preprocessing

Although one might consider this dataset as big machinery data that provides an excellent opportunity to explore the behavior of VRUs in different working states, It cannot be used directly to develop deep learning models. A significant effort is needed to prepare a dataset ready to be used by DL models. With the problem at hand, the data is stored in various Microsoft Excel files with some differences in the format for both VRUs. Therefore, it is needed to concatenate all the files, check the order of the columns, and save them in some optimized file format for storing big chunks of data. Here, we store all the data in two HDF files, each one corresponding to one of the VRUs. HDF files are one of the best options in such situations as they offer very fast read/write operations for big amounts of data.

Exploring data columns one-by-one is a necessary step to start with. It is very important for data analysts to know what is the parameter presented by each column; what data type could be included (e.g., integers, numbers, categorical, strings, and date-times); which values must be considered as NaN (not-a-number) values; and finding out if a column contains useful information for the current data modelling objectives. In this case study, after a few meetings with the plant

engineers, some columns of the data have been detected to contain wrong or non-representative information for the fault diagnosis purposes. These columns have been removed from the data. On the other hand, the columns with percentage unit are likely related to the usual operations of the vapor recovery unit than the working states. Some examples could be valve opening rates, or some liquids level measurements in the system. Therefore, these columns also have been deleted.

Here, all the preparation and preprocessing steps will be explained for the first VRU, as the second unit goes through exactly the same steps. Any observation is recognized by its index which is of type date-time. At first, the indices are saved as text strings, but they must be converted to Python date-time variables. It is also important to make sure that all the observations have valid and unique index values, and to sort all the observations based on their indices. At this point, the dataset contains 2,384,640 rows and 155 columns.

The final version of the dataset to be used by deep learning models must contain only numeric values. As a result, the next step is to convert all columns to numbers. Some columns may contain numbers stored as strings, and some columns may include categorical values. After checking and performing all the conversions, any value in the dataset is a number or NaN (not-a-number). Figure 6.1 shows the distribution of NaN values in the columns. It shows that after conversions, 26 columns have more than 70% of the values as NaNs. All the other columns contain less than 2% NaN values. These 26 columns must also be deleted from the dataset as non-informative features. Before removing these columns, the dataset has 61,011,703 NaN values. This amount is reduced to 390,943 after this elimination, and the resulting dataset has 129 informative columns. The further handling of NaNs will be postponed after few additional steps in between.

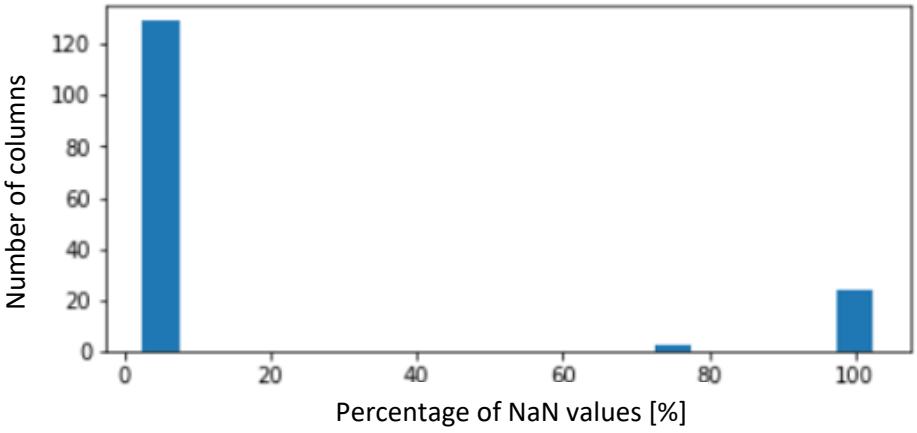


Figure 6.1. Distribution of NaN values in columns for the first VRU

As mentioned before, at this moment each row of the dataset does not have a label. A second Excel file is available in which the normal working intervals of the equipment and the reason each interval has been finished are presented. Using this file and the observations' indices each row

will be assigned a label. As explained in the previous section, each observation has one of the "normal", "pause", and "fault" labels. The data provider informed us that the first four months of operation were mainly dedicated to install and test the equipment, so very likely the sensor recordings may not be completely valid or synchronized. therefore, all the data rows in this period have been removed from the dataset. Figure 6.2 shows the percentage of each working state in the dataset. Observations with "pause" label correspond to the recordings when the equipment was stopped because of any external causes. Thus, these observations are not considered in fault diagnosis process. Also note that the dataset is highly unbalanced, as the amount of "normal" data is about 31 times more than the amount of "fault" rows.

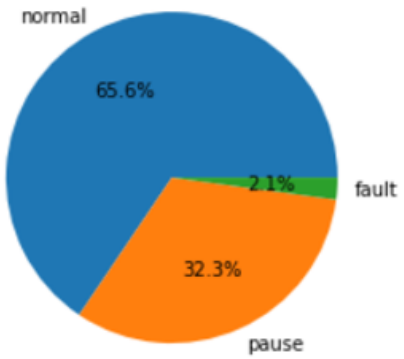


Figure 6.2. Percentage of different working states (labels) for the first VRU

Statistical analysis also plays a significant role in data preprocessing. Such analyses mainly depend on the type of problem one is going to perform, i.e., either fault classification or fault prognosis (regression). In this case study, the dataset is highly unbalanced, and classification methods would not work well for fault diagnosis purposes. One good alternate option would be to develop an anomaly detection. The implementation details are discussed in the next section. However, note that in an anomaly detection a model is trained only on "normal" data to learn the patterns corresponding to the normal operating state. Then, this trained model sees new data samples and predicts whether the observation corresponds to a normal state or not. That is why here, the statistical preprocessing must be done only based on the rows with "normal" label.

One statistical preprocessing approach is to calculate the amount of information a column conveys. There are many methods that can be used for this purpose, but one of the simplest and most effective ones is the ratio of the standard deviation and the mean of the column. The standard deviation is a parameter showing the amount of dispersion of a variable. Although this parameter could be used directly, it is recommended to scale it with the mean value of the variable. This is important because, for instance, 1 unit of dispersity for a variable with mean 1 is completely different with a variable with mean 100. In the former case, the dispersion is 100% while in the latter one is only 1%. Here, all the columns with dispersion ratio less than 5% are removed, which resulted in 58 columns with sensor data without relevant variation. These sensors have had almost constant values for the time period with normal data. At this point, the dataset has now 71 remaining features.

Table 6.1 provides a list of few examples of the sensors recorded signal in this case study. There are many sensors recording the same physical variable but installed in different locations of the equipment. Some of these sensors may have highly correlated information. If such correlated variables are used in deep learning models, they will have an artificially higher influence than other variables, and thus the model will tend to learn more the patterns in such variables. That is why it is important to check the correlation among features and, if necessary, one has to remove redundant information from the dataset. Therefore, for each column, the correlation with all other columns is calculated, and all the features with more than 95% correlation with the current column are removed. This step removes 11 more columns, and the resulting dataset has 60 final columns.

The previous steps analyzed and cleaned the columns (features), but still many observations contain NaN values. In time-series an efficient way to reduce the number of NaNs is resampling. Here, each sample represents 15 seconds of equipment operation, but for our objective we can use larger periods as well. A resampling to one minute combines every four data samples and calculates the mean value for each feature. By doing so, a significant percentage of the NaN values are replaced effectively with numeric values. In the final step of the preprocessing, all the remaining NaN values are removed via a linear interpolation. The interpolation has been done separately for normal and faulty data portions, thus it ensures us that the dataset now contains only valid numeric types.

6.3 Dataset Preparation

As mentioned in the previous section, the nature of the existing data makes anomaly detection an appropriate option for fault diagnosis. The implementation of anomaly detection in deep learning needs a dataset containing only normal data to train the model, which learns the common patterns in normal data samples. Another dataset is required to evaluate the capacity of the trained model to discriminate normal operating health state from abnormal health state. To this aim, a second dataset containing both normal and anormal data is needed. The difference between these two datasets is that the former one does not need labels, as it only contains data from one class (i.e., health state). The latter one does need bi-class labels, usually with 0 corresponding to the normal class, and 1 for anomalies. Figure 6.3 shows the procedure to make the datasets just after preprocessing steps. In this figure, the yellow blocks are the preprocessed normal and faulty data rows. The amount of normal data is about 31 times more frequent than faulty samples. To make the dataset containing both classes balanced, the same number of samples as the fault data are selected randomly from the normal observations.

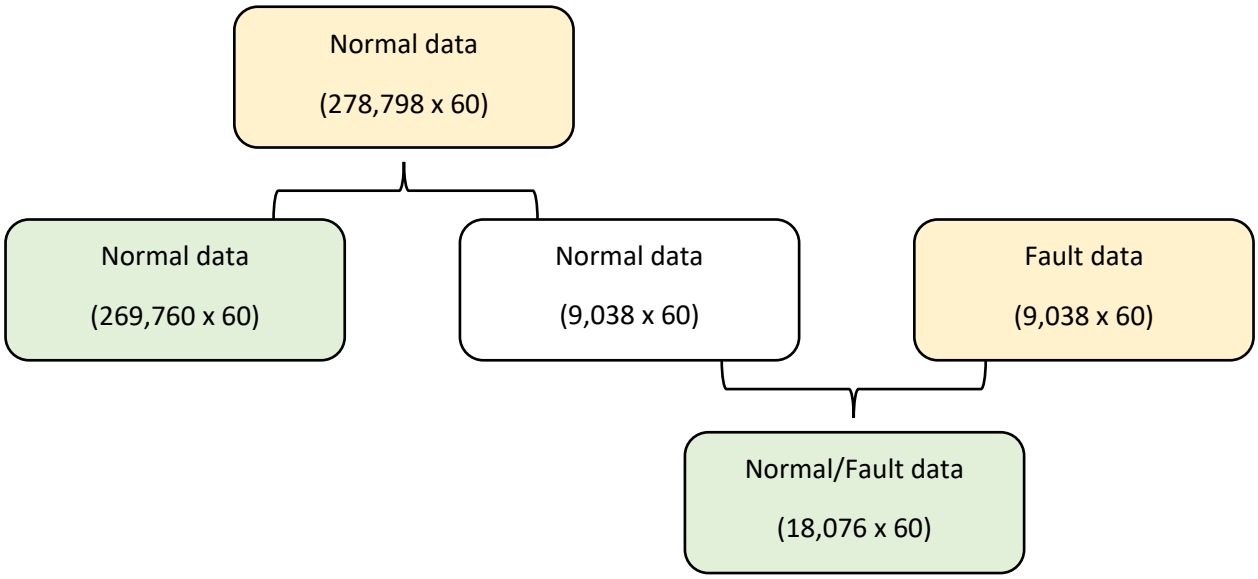


Figure 6.3. Dataset structure for anomaly detection.
Yellow blocks are the preprocessed normal and faulty data,
and green blocks are the datasets to be used in anomaly detection.

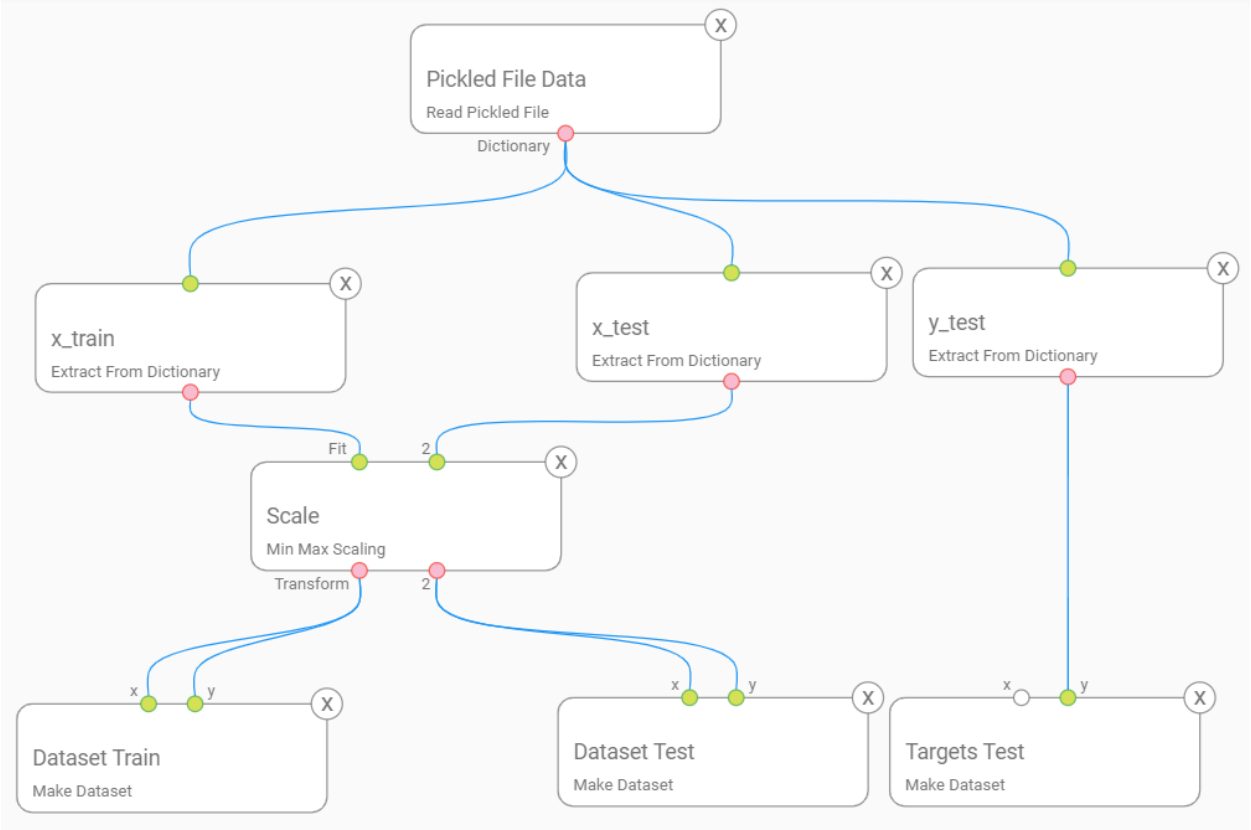


Figure 6.4. Dataset preparation for anomaly detection

The normal and mixed data are stored in a Python ".npz" files. Figure 6.4 demonstrates the dataset preparation in DLHub. The file contains three tables containing features for the train and test data as well as labels for the test dataset. The features are scaled using a Min-Max Scaler. The scaler is fitted on the train data and then transforms both feature sets. Lastly, as the features are going to be used in an autoencoder, see chapter 5, their "x" and "y" sockets must be connected to the scaled features. The target dataset only contains the labels without any features.

6.4 Data Modelling

As we have discussed earlier, the available data is highly unbalanced between the two health states of the equipment. Therefore, simple classification algorithms would not work properly for fault diagnosis purposes. In such cases, anomaly detection is a good choice. Chapter 5 provides an explanation about representation learning algorithms like autoencoders. The main idea in autoencoders is that they start with a data sample, and after few hidden layers which have smaller sizes than the input data, they try to reconstruct the original data. This way, autoencoder identifies the major patterns existing in the data and can extract the most representative features of the data in the latent layer. The key point is that normal data usually have different patterns than the faulty ones. As a result, an autoencoder which is trained only on the normal data, only learns to reconstruct data from the same class, and it produces low quality reconstructions (high reconstruction errors) for the data from other classes (such as faulty health states). This reconstruction error could be used to classify normal and faulty health states. Figure 6.5 shows a simple architecture for an autoencoder to be used in this study. The encoder starts with 60 neurons in the input layer and reduces the dimensionality to 10 in the latent layer. Then, the decoder reconstructs the original data. The Sigmoid activation function in the last layer forces that all reconstruction neurons have values between 0 and 1.

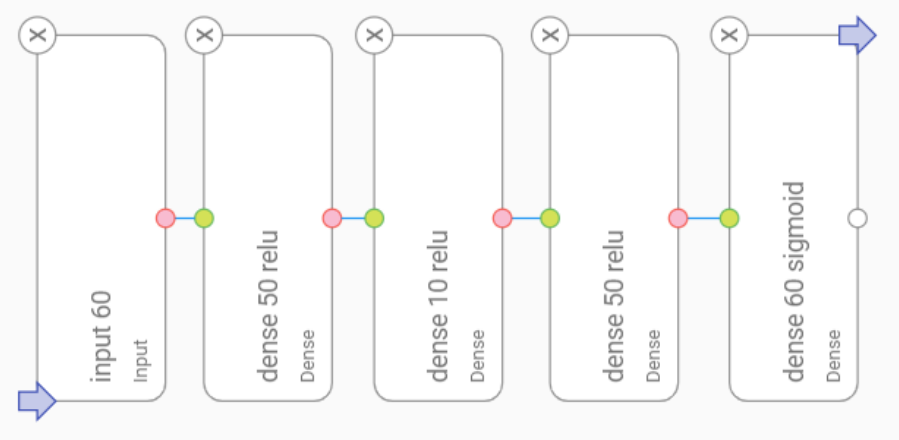
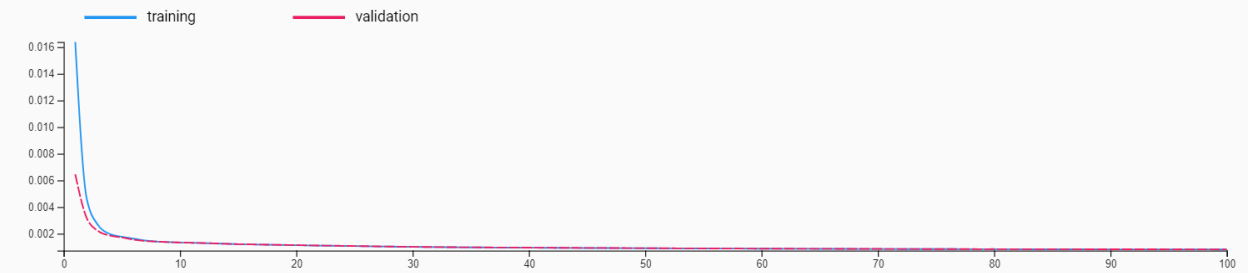


Figure 6.5. Autoencoder architecture for anomaly detection

In this case, we do not need to define encoder and decoder as separated networks as we only would use the whole autoencoder in the analysis. The model has “mean squared error” as its loss function, and Adam optimizer with learning rate of 0.001 has been used. Figure 6.6 shows the autoencoder’s training curve on the normal data. It is obvious that the model has no overfitting to the training set and the value of the reconstruction loss is quite small. Therefore, it could be inferred that the AE had successfully extracted 10 dominant features in the latent layer from the 60 input features.



epoch ↓	loss	val_loss
100	0.0008104253847186883	0.0008148690540310278
99	0.000811422431024876	0.0008141440213550723
98	0.0008123205947639902	0.0008149153944904459

Figure 6.6. Autoencoder training curve on the first VRU dataset

6.5 Anomaly Detection

Anomaly detection is a commonly used algorithm in reliability engineering and fault diagnosis. DLHub has a standard implementation for anomaly detection based on autoencoders. Figure 6.7 shows how to start an anomaly detection task in DLHub. The task needs to know the name of the pretrained AE, as well as the test data features and labels. The test dataset is a balanced mixture of the data from both normal and faulty health states. In the labels, often 0 and 1 are chosen for the normal and faulty classes, respectively. The task also needs to know the type of the reconstruction error. Classic autoencoders mainly use Root Mean Squared Error (RMSE), which is a one-by-one comparison of the neurons in the original and reconstructed data. On the other hand, if in advanced studies, the user wants to employ a variational autoencoder for anomaly detection purposes, such RMSE reconstruction error would not be a helpful criterion, as VAEs try to generate an output with the same statistical distribution as the original data rather than the exact same data reconstruction. In these cases, a “negative log likelihood Bernoulli” reconstruction error is used to compare the distributions.

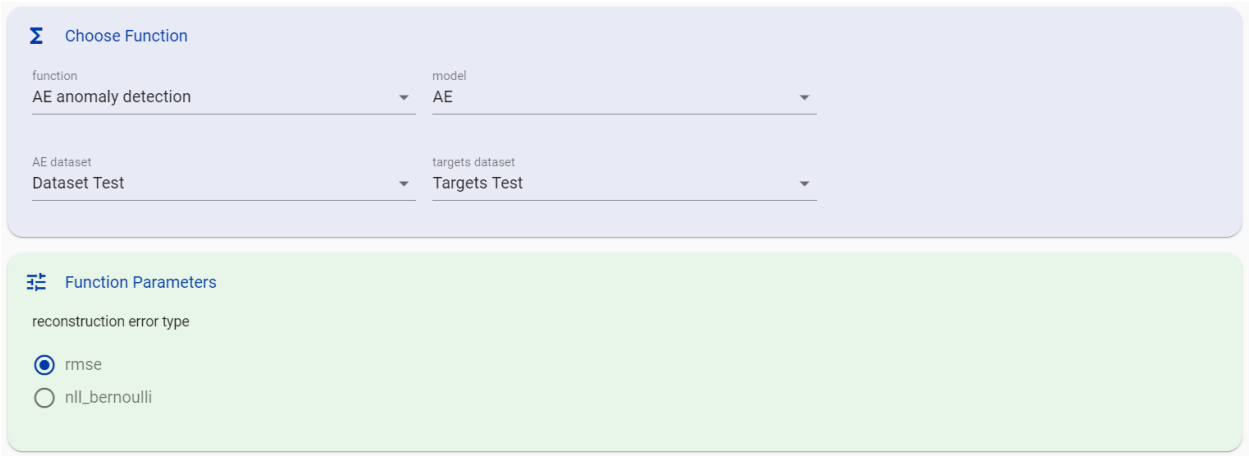


Figure 6.7. Implementation of anomaly detection in DLHub

Anomaly detection task is divided into four steps. The first step is the calculation and plotting the Receiver Operating Characteristic (ROC) curve. In a binary classifier, ROC curve is a plot depending on the positive alarms of the system and is considered to represent the classification ability of the model. True Positive Rate (TPR) is the probability of detection and usually called as recall or sensitivity. On the other hand, False Positive Rate (FPR) is the probability of the false alarms. The ROC curve is the plot of TPR against FPR and shows the efficiency of the binary classifier to discriminate positive and negative classes. To make the ROC curves comparable, some numeric characteristics of the curve must be calculated. The Area Under Curve (AUC) is a value between 0 and 1 and is the most representative parameter to measure the capacity of the classifier. Figure 6.8 shows the ROC curve and the AUC value for the trained AE on the test data. In this case, the AUC value is high showing that the autoencoder can correctly predict almost all labels.

As discussed earlier, the classification is done based on some thresholds on the AE reconstruction error. Choosing the best threshold is highly dependent on the user´s expectation on the false negative rate of the classifier. False negatives happen when the equipment has a fault, but the model does not recognize it, and hence it does not fire any alarm. Such cases could end up in very dangerous and costly situations. Therefore, usually the idea is to minimize the false negatives. in this context, minimizing the false negatives is often equivalent to maximizing the false positives (i.e., false alarms). As a result, depending on the use case, an equilibrium must be maintained between them. In DLHub, the user can determine the maximum percentage/count of the false negative cases, and the application calculates the equivalent threshold. The user can choose between using the same optimal threshold or any other custom value in the classification procedure. As shown in figure 6.9, testing various threshold values to comprehend the behavior of the classifier is quite easy in DLHub. In each experiment, the application presents some statistics of the binary classification task including precision, recall, f1-score, and support for both normal and faulty classes. The figure shows both the best threshold and the corresponding parameters for maximum 1% of false negatives.

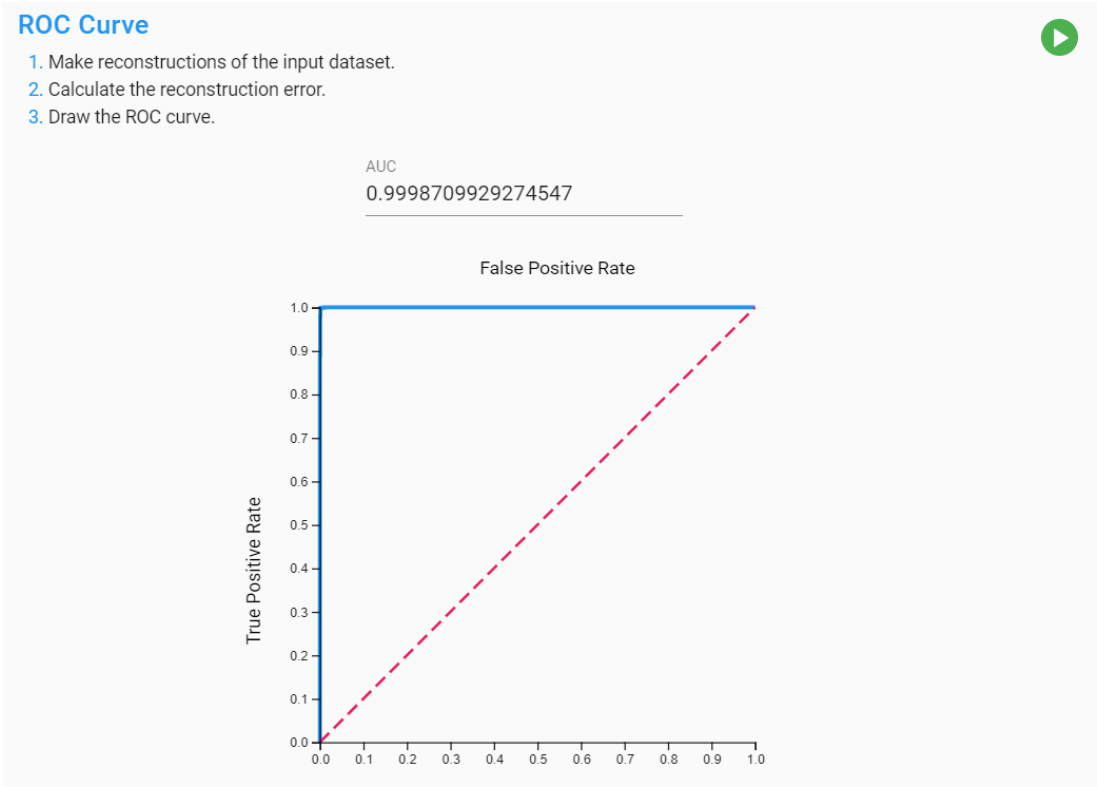


Figure 6.8. Step 1 of anomaly detection: ROC curve

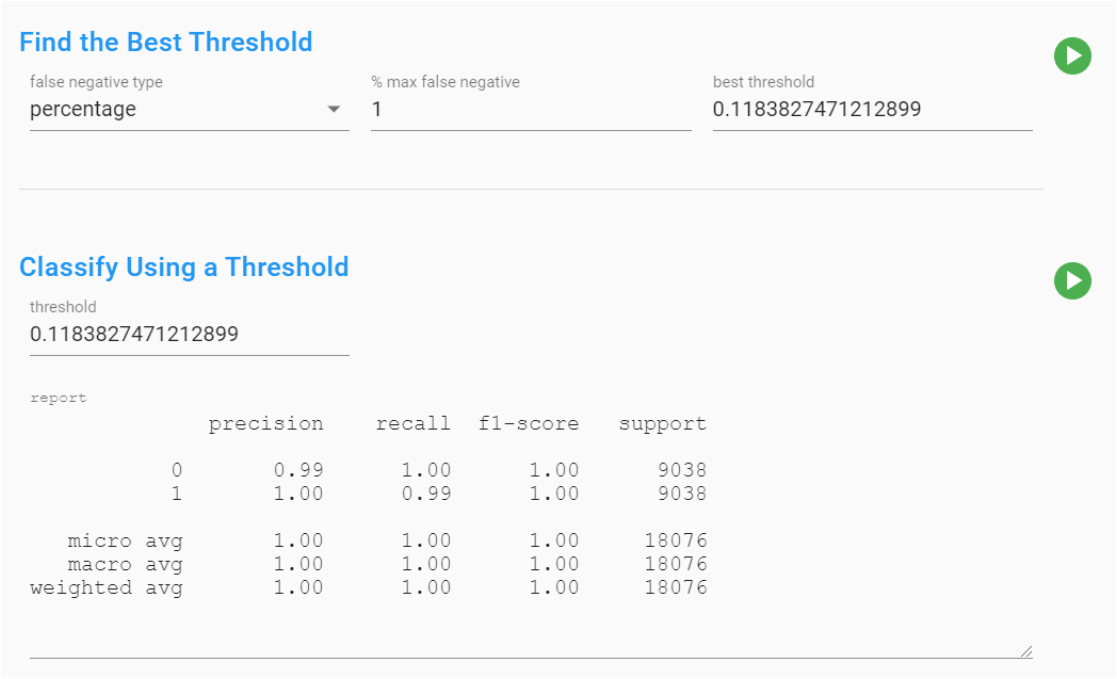


Figure 6.9. Steps 2 & 3 of anomaly detection: finding the best threshold and classification results

The last step in anomaly detection is constructing the confusion matrix. Figure 6.10 presents the graph for the current case study. It shows that while the objective is to have maximum 1% of false negatives, the classifier in 99.9% and 99.2% of cases of normal and faulty data, respectively, correctly predicts the health states (i.e., labels). It means that the reconstruction error has almost perfectly separated the two classes of data.

All the data cleaning, preparation, analysis, and modelling steps performed are repeated on the data from the second vapor recovery unit as well. Although the same features and criteria have been used in the preprocessing sections, a different set of features have been removed as non-informative or redundant columns from the dataset. This seems quite reasonable as the two equipment were operating under similar but not identical working conditions and have functioned under various workloads.

In the anomaly detection task for the second VRU, the same autoencoder architecture has been used. Doing so, the results for both equipment could be compared. As figure 6.11 shows, the same AE works quite well on the second dataset, thus it could be inferred that the preprocessing stages and the anomaly detection task could be repeated for other VRUs with small case-specific tunings. Indeed, for the second VRU the same autoencoder architecture has AUC value of almost 0.94 and the best threshold equilibrium may generate about 10% false negatives.

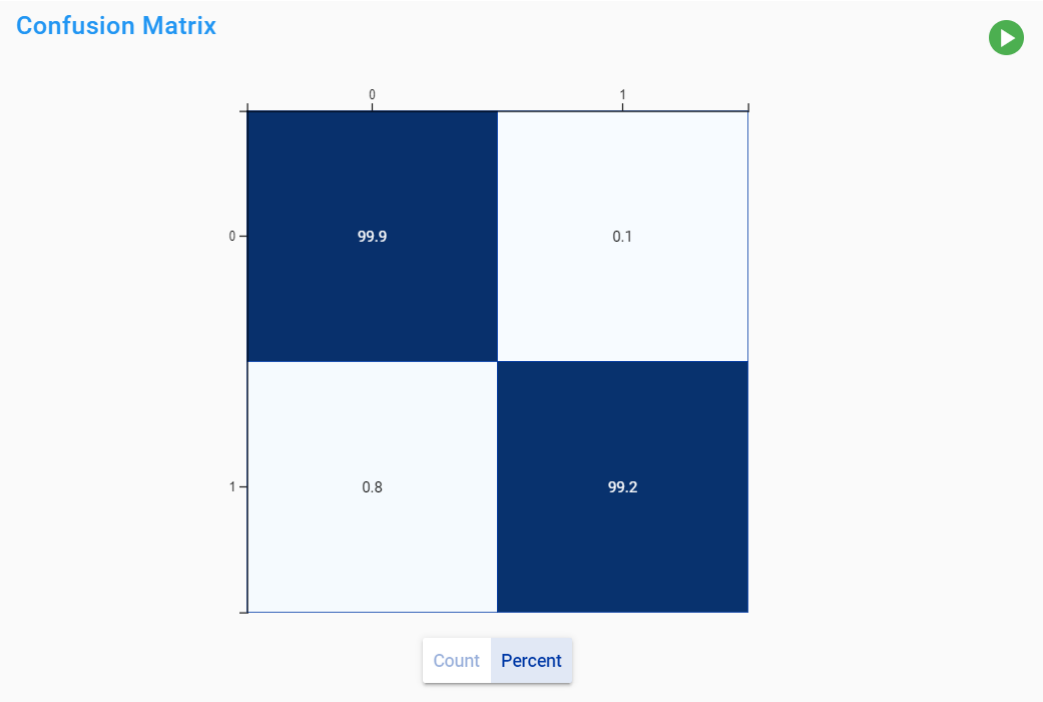
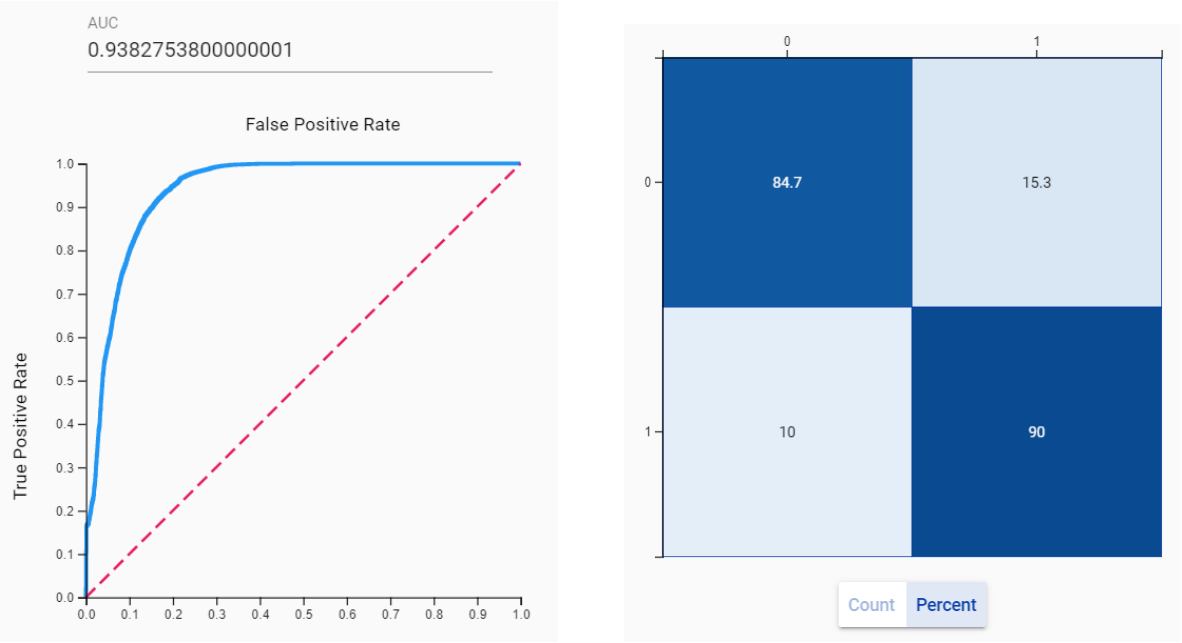


Figure 6.10. Step 4 of anomaly detection: confusion matrix

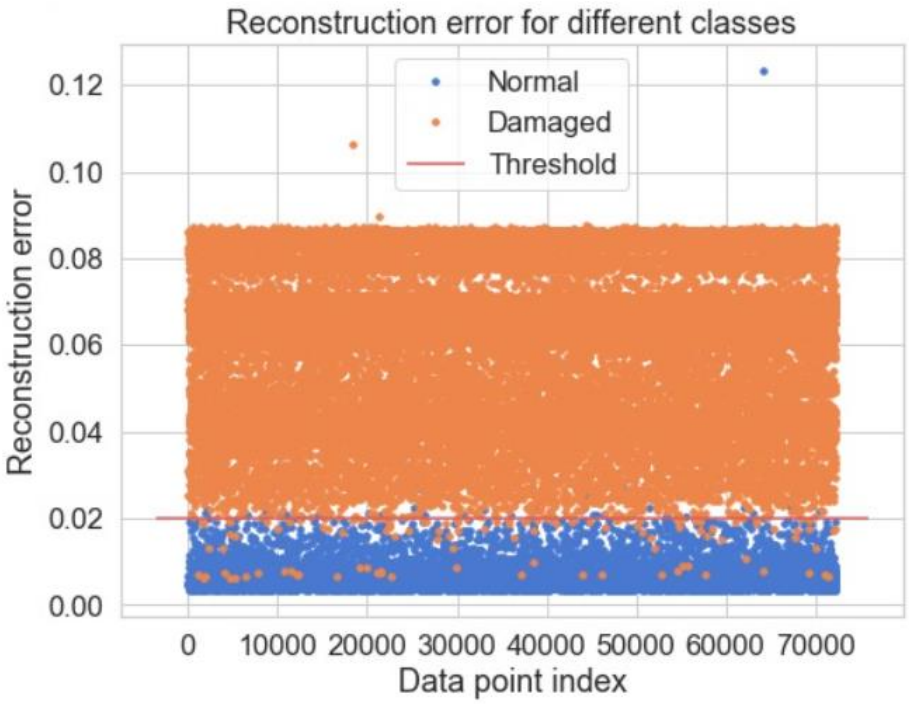


(a) (b)

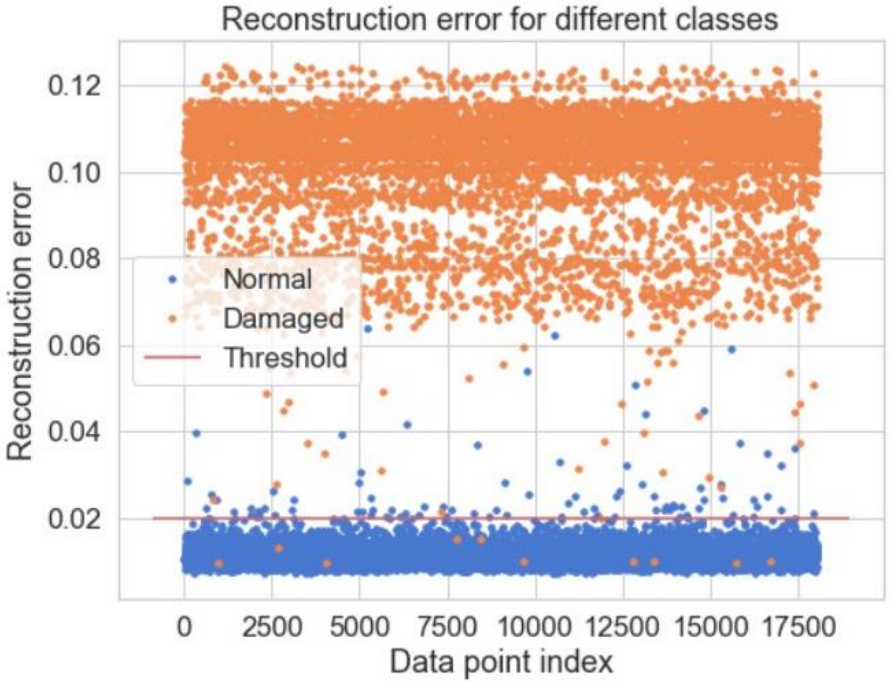
Figure 6.11. Anomaly detection on the second VRU dataset.
(a) ROC curve, (b) Confusion matrix.

Further insights from this fault diagnosis problem can be obtained from figure 6.12. Here, the reconstruction errors for both normal and faulty health states have been plotted for all test dataset for the first vapor recovery unit. For the sake of comparison, the same plot is presented for the raw data without any preprocessing as well as the prepared dataset. Note that the separation of the classes after preprocessing has been increased significantly. It means that if someone tries to develop an autoencoder for the raw data, a significant portion of the faulty data can be recognized correctly, but still there is a considerable overlap between classes (i.e., health states). On the other hand, after applying all the preprocessing steps, the classes are less mixed together and there are very few data in the overlap zone.

As a summary, in this example, data exploration took a significant part of the analysis time, as it required frequent meetings with the field engineers. All data cleaning, preparation, and analysis need a solid understanding of the physics of the problem, the hypotheses, constraints, operating and maintenance conditions and conventions in that specific industry. Moreover, any data modelling has a significant part of model tuning, where different architectures, hyper-parameters, and model settings should be selected through a try-and-error procedure.



(a)



(b)

Figure 6.12. Comparison of the AE reconstruction errors before and after preprocessing. (a) Before preprocessing, (b) After preprocessing.

Chapter 7

Conclusions

7.1 Summary

The main objective of the thesis was to develop a new framework and tools to make the usage of artificial intelligence in general, and deep learning in particular, more accessible for engineers and professional who work in reliability engineering, predictive maintenance, and physical asset management. Deep Learning Hub (DLHub) is presented as the proposed framework to achieve this objective. After a brief review of the background concepts in data processing, DLHub is explained through a simple classification problem. It has also been shown that implementing advanced ANN architectures like convolutional neural networks, variational autoencoders, and generative adversarial networks is possible in a simple and intuitive manner in DLHub. Finally, in the case study, the usefulness of DLHub in real-world problems has been demonstrated through an autoencoder-based anomaly detection on a dataset containing massive and multidimensional data from a highly sensorized compressor in a vapor recovery unit in an offshore petroleum platform. This chapter provides some concluding remarks and suggestions for future works to address some of the limitations here identified.

7.2 Conclusions

Tensorflow, as the most common framework in deep learning, contains a stack of APIs with different levels of abstraction. Low level APIs usually have more access to the hardware and could be used to implement more customized models. The cost one pays to take advantage of such flexibility is the complexity of the syntax of such APIs. On the other hand, high level APIs are much simpler to implement commonly used models rapidly in a language very near to human languages. DLHub adds a new layer of abstraction on top of Tensorflow, in which the user can implement deep learning algorithms in an intuitive drag and drop manner. Certainly, this visual programming layer would not be very useful in pure and fundamental research in data-science, but it provides lots of new opportunities in more applied areas like reliability, predictive maintenance and physical asset management.

Deep Learning Hub not only permits working with many different ANN architectures, but also offers out-of-the-box implementation of some models that are specifically useful in reliability and maintenance. Examples of such models are variational autoencoders and generative adversarial networks that have complex implementations and could be a real challenge for most of practitioners. In DLHub, such models have standard implementations that work quite well as generative models, or in combination with other models for fault diagnosis or prognosis purposes. Another example is autoencoder-based anomaly detection which is a frequently used data processing path in fault diagnosis. DLHub lets the users to combine their completely custom autoencoders to the standard anomaly detection task, and to reach some acceptable results rapidly.

Another contribution of DLHub is the offered conventions provided based on the best practices in deep learning projects. Following these conventions, the users keep the same structure for all their projects, starting from preparing datasets, followed by implementing neural network architectures, compiling custom models, and defining the run-time tasks. Keeping all the projects in a unified structure decreases significantly the probability of occurring errors somewhere in the data processing path. Moreover, it lets the users to focus on the conceptual parts of data modelling instead of organizing and reviewing the codes again and again. The automatically generated Python code for each step would be an excellent starting point for the users who want to dive deeper in the topic.

The graphical programming module in DLHub is developed from the ground in order to allow for the implementation of data-flow algorithms. That is why many rules, behaviors, and constraints of artificial neural network architectures by default exist in DLHub. This module simplifies significantly the implementation of multi-input multi-output models, sharing a layer between various models, changing the number of input/output sockets of some specific blocks based on the use cases. As a results DLHub provides a rapid prototyping environment for engineers. Regarding the point that, in many cases, an optimum deep learning architecture must be found in a try-and-error procedure, a fast and error-free prototyping tool makes a big difference especially in data/model exploration steps.

Data privacy is another interesting topic considered in DLHub. Many data processing tools in the same level force the users to upload the whole dataset to some cloud databases for further analyses. However, data is an important and strategic asset for industries, and they do not want to share the data with others. DLHub is developed on top of Jupyter technology and performs all the processing steps in the clients' local machines. In other words, although DLHub is an online application that only needs a web browser to be used, and also it keeps all the projects in a cloud database, the data always remains in the client's local computer. This way, it takes advantage of both being an online tool that could be used on various computers and operating systems, and a local tool that keeps 100% of the privacy of the data for the users.

The case study of the thesis shows clearly that DLHub works well on real datasets. All the results have been compared with pure Keras codes written in a Jupyter notebook for various model architectures and their correctness is evaluated for different anomaly thresholds. DLHub also permits to save the autoencoder used in this study to a file and to use it as a pre-trained model in future deployments. Moreover, it is important to mention that DLHub must be considered as a helpful tool in specific use cases and does not replace coding in all applications of artificial intelligence to reliability and predictive maintenance.

Finally, although Deep Learning Hub helps the users in many aspects, it has some limitations as well. The training process for a model in DLHub is done through iterations over various epochs. This training strategy is the best one for many use cases, but still some very advanced models, especially in academia and research sectors, need iterations over batches of the data. The other limitation of DLHub is that it assumes that the user has access to the preprocessed data. Data exploration and preprocessing are complex and time-consuming tasks that need a good level in programming and knowledge of specific libraries. Moreover, DLHub does not give the users the opportunity to compare their deep learning models with more traditional machine learning ones like Support Vector Machines, or Random Forests. These traditional algorithms still have a lot of applications in reliability and maintenance. Lastly, Deep Learning Hub does not have built-in

implementations for some novel methodologies like Natural Language Processing. Such novel models and algorithms offer new types of analysis on the data which never have been possible before.

7.3 Future Work

Deep Learning Hub is developed on top of Keras and Tensorflow as very young open source projects. Generally, open source tools are very dynamic in their nature, as the community is always extending them to cover more and more features. Tensorflow had major changes in its fundamental APIs in version 2.0. DLHub adapted the latest features in Tensorflow 2.0 but very soon there were some additional changes in the next versions of Tensorflow. It is of big importance to keep working on DLHub to maintain it compatible with the latest version of Keras and Tensorflow. Here, some future works on DLHub are listed:

- DLHub for the moment has few reliability-specific tasks implemented (e.g. anomaly detection). Regarding the successful experience with such particular tasks, much more diagnosis or prognosis tasks especially for fault detection, fault characterization, and remaining useful life (RUL) analysis must be added to the application.
- DLHub already contains the implementation of some advanced models like VAEs or GANs, but some newer models like Capsule Networks [34] or Graph Networks [35] also have been presented in the AI community which are not still implemented in DLHub. As all these new concepts are introducing new applications in the area of physical asset management, further work is needed to make them accessible in DLHub as well.
- Natural Language Processing (NLP) is getting a lot of attention in the recent years [36]. In reliability and maintenance engineering, a big portion of the data is stored in the form of qualitative and descriptive reports. Although DLHub already contains few layers to work on NLP models, a lot of more effort is needed in this section.
- Reinforcement Learning is another group of AI hot and interesting algorithms which are not covered in DLHub. Regarding their increasing applications in selecting maintenance strategies [10], [11], it would be a good idea to cover them in the application as well.
- Finally, DLHub assumes that the user has a clean and prepared dataset to work on. In most of the real-world problems this is not the case. In the case study of this thesis, we spent about 75% of the whole time on data cleaning and preprocessing steps. It shows that adding newer features for the preprocessing procedures also could be priceless works for the future.

Bibliography

[1] N. A. J. Hastings, “Introduction to Asset Management, in *Physical Asset Management: With an Introduction to ISO55000*”, Cham: Springer International Publishing, 2015, pp. 3–23.

[2] Z. Ding and M. Fei, “An Anomaly Detection Approach Based on Isolation Forest Algorithm for Streaming Data using Sliding Window”, *IFAC Proceedings Volumes*, vol. 46, no. 20, pp. 12–17, 2013, doi: https://doi.org/10.3182/20130902-3-CN-3020.00044.

[3] S. Guha, N. Mishra, G. Roy, and O. Schrijvers, “Robust Random Cut Forest Based Anomaly Detection on Streams”, in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, 2016, pp. 2712–2721.

[4] Y. Bao, Z. Tang, H. Li, and Y. Zhang, “Computer vision and deep learning–based data anomaly detection method for structural health monitoring”, *Structural Health Monitoring*, vol. 18, no. 2, pp. 401–421, Feb. 2018, doi: 10.1177/1475921718757405.

[5] J. Mao, H. Wang, and B. F. Spencer, “Toward data anomaly detection for automated structural health monitoring: Exploiting generative adversarial nets and autoencoders”, *Structural Health Monitoring*, p. 1475921720924601, Jun. 2020, doi: 10.1177/1475921720924601.

[6] G. San Martin, E. Lopez Droguett, V. Meruane, and M. das Chagas Moura, “Deep variational auto-encoders: A promising tool for dimensionality reduction and ball bearing elements fault diagnosis”, *Structural Health Monitoring*, vol. 18, no. 4, pp. 1092–1128, 2019, doi: 10.1177/1475921718788299.

[7] D. B. Verstraete, E. Lopez Droguett, V. Meruane, M. Modarres, and A. Ferrada, “Deep semi-supervised generative adversarial fault diagnostics of rolling element bearings”, *Structural Health Monitoring*, vol. 19, no. 2, pp. 390–411, 2020, doi: 10.1177/1475921719850576.

[8] C. Modarres, N. Astorga, E. Lopez Droguett, and V. Meruane, “Convolutional neural networks for automated damage recognition and damage type identification”, *Structural Control & Health Monitoring*, vol. 25, no. 10, p. e2230, 2018, doi: 10.1002/stc.2230.

[9] S. Cofre-Martel, P. Kobrich, E. Lopez Droguett, and V. Meruane, “Deep Convolutional Neural Network-Based Structural Damage Localization and Quantification Using Transmissibility Data”, *Shock and Vibration*, vol. 2019, p. 9859281, 2019, doi: 10.1155/2019/9859281.

[10] J. Huang, Q. Chang, and J. Arinez, “Deep Reinforcement Learning based Preventive Maintenance Policy for Serial Production Lines”, *Expert Systems with Applications: X*, p. 100034, 2020, doi: https://doi.org/10.1016/j.eswax.2020.100034.

[11] S. Wei, Y. Bao, and H. Li, “Optimal policy for structure maintenance: A deep reinforcement learning framework”, *Structural Safety*, vol. 83, p. 101906, 2020, doi:

https://doi.org/10.1016/j.strusafe.2019.101906.

[12] C. Correa-Jullian, J. M. Cardemil, E. Lopez Droguett, and M. Behzad, “Assessment of Deep Learning techniques for Prognosis of solar thermal systems”, *Renewable Energy*, vol. 145, pp. 2178–2191, 2020, doi: https://doi.org/10.1016/j.renene.2019.07.100.

[13] A. Aria, E. Lopez Droguett, S. Azarm, and M. Modarres, “Estimating damage size and remaining useful life in degraded structures using deep learning-based multi-source data fusion”, *Structural Health Monitoring*, vol. 0, no. 0, p. 1475921719890616, doi: 10.1177/1475921719890616.

[14] A. Ruiz-Tagle Palazuelos, E. Lopez Droguett, and R. Pascual, “A novel deep capsule neural network for remaining useful life estimation”, *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk Reliability*, vol. 234, no. 1, pp. 151–167, 2020, doi: 10.1177/1748006X19866546.

[15] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.” 2015, [Online]. Available: https://www.tensorflow.org/.

[16] F. Chollet, “Keras.” 2015, [Online]. Available: https://keras.io/.

[17] A. Paszke *et al.*, “Automatic differentiation in PyTorch”, 2017.

[18] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[19] H. Abbes and F. Gargouri, “Big Data Integration: A MongoDB Database and Modular Ontologies based Approach”, *Procedia Computer Science*, vol. 96, pp. 446–455, 2016, doi: https://doi.org/10.1016/j.procs.2016.08.099.

[20] M. Folk, R. E. McGrath, and N. Yeager, “HDF: an update and future directions”, in *IEEE 1999 International Geoscience and Remote Sensing Symposium. IGARSS’99 (Cat. No.99CH36293)*, Jun. 1999, vol. 1, pp. 273–275 vol.1, doi: 10.1109/IGARSS.1999.773469.

[21] V. Patel, “A Practical Guide to Dimensionality Reduction Techniques”, in *PyData DC 2016*, 2016.

[22] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd Editio. O’Reilly Media, Inc., 2019.

[23] F. Aguilar, “Getting Started with TensorFlow, The Basics of Tensorflow in Python”, *Medium*, 2020.

[24] T. Kluyver *et al.*, “Jupyter Notebooks – a publishing format for reproducible computational workflows”, in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS Press Ebooks, 2016, pp. 87–90.

[25] Y. LeCun, C. Cortes, and C. J. C. Burges, “THE MNIST DATABASE.” http://yann.lecun.com/exdb/mnist/.

[26] H. Xiao, K. Rasul, and R. Vollgraf, “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.” 2017.

[27] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization.” 2014.

[28] K. Hagiwara, “Regularization learning, early stopping and biased estimator”, *Neurocomputing*, vol. 48, no. 1, pp. 937–955, 2002, doi: https://doi.org/10.1016/S0925-2312(01)00681-6.

[29] D. P. Kingma and M. Welling, “Auto-Encoding Variational Bayes.” 2013.

[30] I. J. Goodfellow *et al.*, “Generative Adversarial Networks.” 2014.

[31] W. Zhang, X. Li, X.-D. Jia, H. Ma, Z. Luo, and X. Li, “Machinery fault diagnosis with imbalanced data using deep generative adversarial networks”, *Measurement*, vol. 152, p. 107377, 2020, doi: https://doi.org/10.1016/j.measurement.2019.107377.

[32] K. Yan, A. Chong, and Y. Mo, “Generative adversarial network for fault detection diagnosis of chillers”, *Building and Environment*, vol. 172, p. 106698, 2020, doi: https://doi.org/10.1016/j.buildenv.2020.106698.

[33] T. Pan, J. Chen, J. Xie, Y. Chang, and Z. Zhou, “Intelligent fault identification for industrial automation system via multi-scale convolutional generative adversarial network with partially labeled samples”, *ISA Transactions*, vol. 101, pp. 379–389, 2020, doi: https://doi.org/10.1016/j.isatra.2020.01.014.

[34] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic Routing Between Capsules.” 2017.

[35] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A Comprehensive Survey on Graph Neural Networks”, *IEEE Trans. Neural Networks Learn. Syst.*, pp. 1–21, 2020, doi: 10.1109/tnnls.2020.2978386.

[36] Y. Mo, D. Zhao, J. Du, M. Syal, A. Aziz, and H. Li, “Automated staff assignment for building maintenance using natural language processing”, *Automation in Construction*, vol. 113, p. 103150, 2020, doi: https://doi.org/10.1016/j.autcon.2020.103150.