National Technical University of Athens
Department of Electrical and Computer Engineering

Implementing Machine Learning Models and Distributed Data Processing on Cloud Computing Infrastructures

Master Thesis

Author: Christos Zacharioudakis (Student ID: 03400132)
Supervisor: Professor Nectarios Koziris

Thesis Committee:

- Professor Nectarios Koziris
- Assistant Professor Ioannis Konstantinou
- Associate Professor Georgios Goumas

May 13, 2023

Περίληψη

Στις μέρες μας, δεδομένα υπάρχουν σε αφθονία, αυξάνονται συνεχώς και βρίσκουν πολλές χρήσεις. Μια πιο πρόσφατη χρήση είναι η εκπαίδευση μοντέλων Μηχανικής Μάθησης, λογισμικού που είναι ικανό να λαμβάνει τις δικές του αποφάσεις. Σε αυτή τη διατριβή, θα εκπαιδεύσουμε και θα αναπτύξουμε τέτοια μοντέλα, με τη μορφή ροών εργασιών Μηχανικής Μάθησης. Μια ροή εργασίας Μηχανικής Μάθησης αποτελείται από έναν πεπερασμένο αριθμό βημάτων, τα οποία εκτελούνται σε κάποιο υπολογιστικό σύστημα.

Ωστόσο, η εκπαίδευση τέτοιων μοντέλων απαιτεί περισσότερα από δεδομένα. Απαιτεί τεράστιους υπολογιστικούς πόρους, τους οποίους το μέσο υπολογιστικό σύστημα δεν διαθέτει. Το υπολογιστικό νέφος καλείται να λύσει αυτή τη δύσκολη θέση. Ο συνδυασμός των πόρων που προσφέρει το νέφος, μαζί με το Kubernetes, έναν ενορχηστρωτή container (πακέτρων), διευκολύνει την εκτέλεση ροών εργασιών Μηχανικής Μάθησης. Για αυτόν τον σκοπό, θα χρησιμοποιήσουμε το Kubeflow, το οποίο υλοποιείται ειδικά για το Kubernetes και είναι αφιερωμένο στην ανάπτυξη των ροών εργασιών Μηχανικής Μάθησης και τις καθιστά απλές, φορητές και επεκτάσιμες.

Εκτός από το Kubernetes και το Kubeflow, τα πλαίσια ανάλυσης Μεγάλων Δεδομένων, όπως το Apache Hadoop και το Apache Spark, αξιοποιούνται για τη φιλοξενία και την προεπεξεργασία των δεδομένων, τα οποία θα χρησιμοποιηθούν για την εκπαίδευση των μοντέλων Μηχανικής Μάθησης.

Λέξεις Κλειδιά: Υπολογιστικό Νέφος, Μεγάλα Δεδομένα, Μηχανική Μάθηση, Kubernetes, Kubeflow, Apache Hadoop, Apache Spark

Abstract

In our days, data exists in abundance, it is ever increasing and it finds numerous uses. A most recent use is the training of Machine Learning models, software capable of making their own decisions. In this thesis, we will train and deploy such models, in the form of Machine Learning workflows. A Machine Learning workflow consists of a finite number of steps, which are executed on some computing system.

However, training such models requires more than data. It requires vast computational resources, that the average computing system does not possess. Cloud computing is called upon to solve this predicament. Combining the resources offered by the cloud, alongside Kubernetes, a container orchestrator, facilitates the execution of Machine Learning workflows. For that purpose, we will utilize the Kubeflow project, which is implemented especially for Kubernetes and is dedicated to making deployments of Machine Learning workflows simple, portable and scalable.

In addition to Kubernetes and Kubeflow, Big Data analytics frameworks, such as Apache Hadoop and Apache Spark, are exploited to host and pre-process the data, that will be used to train our Machine Leaning models.

Keywords: Cloud Computing, Big Data, Machine Learning, Kubernetes, Kubeflow, Apache Hadoop, Apache Spark

Acknowledgments

After two years of arduous and yet productive and interesting studies, the time has finally come to present my thesis, which has benefited greatly from the support of many people, some of whom I would sincerely like to thank here.

To begin with, I would like to thank *Professor Ioannis Konstantinou* for pointing me towards exciting and modern topics and for finding the time to guide me and answer my questions.

Finally, I wish to thank my family for their financial and moral support throughout my studies. I owe my deepest gratitude to *my mother* for her constant and unconditional support, patience and encouragement, even in my most pessimistic moments. Last but not least, I would like to thank *my uncle Andreas* for bringing me in contact with computers when I was quite young, thus igniting my interest in them and inspiring me to pursue this field of study.

Contents

1 Introduction

1.1 Data and its usefulness

In computing, data is information that has been translated into a form that is efficient for movement or processing. Relative to today's computers and transmission media, data is information converted into binary digital form. Raw data is a term used to describe data in its most basic digital format. The concept of data in the context of computing has its roots in the work of Claude Shannon, an American mathematician known as the father of information theory. He ushered in binary digital concepts based on applying two-value Boolean logic to electronic circuits. Binary digit formats underlie the CPUs, semiconductor memories and disk drives, as well as many of the peripheral devices common in computing today, such as hard and solid-state drives.

The era spanning from the beginning of the 20th century to today is known as the Information Age. This is largely due to the fact that the world's technological capacity to store information has grown from 2.6 exabytes in 1986 to 5 zettabytes in 2014. Also, due to the extended use of the Internet (Social Media, Cloud Storage e.t.c.) and with the growth of the Internet of Things (IoT),[1] 2.5 exabytes (2.5 million terabytes) of data is created every day and the pace is ever increasing. In fact, it was estimated in 2013 that 90% of the data in the world was generated in 2011 and 2012. It is also estimated that the volume of the available data is doubled every three years and by 2020, data is expected to double every 73 days.

Early on, the importance of data in business computing became apparent by the popularity of the terms "data processing" and "electronic data processing," which, for a time, came to encompass the full spectrum of what is now known as information technology. Over the history of corporate computing, specialization occurred, and a distinct data profession emerged along with growth of corporate data processing. Data processing refers to the process of collecting and manipulating raw data to yield useful information. In technical terms it is the process of converting raw data to machine-readable form and its subsequent processing such as updating, rearranging, or printing by a computer.

[1] The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. In other words, IoT is composed of connected "smart" devices that interact with each other and us while collecting all kinds of data. The number of these devices is estimated from 2 billion devices in 2006 to a projected 200 billion by 2020.

Data processing has numerous applications in many sectors, such as healthcare, customer oriented service (Netflix, Amazon, e.t.c), telecommunication, marketing, commerce, security and many others. As a result, data is considered extremely valuable and nowadays is one of the most important assets a company has. With the rise of the data economy, companies find enormous value in collecting, sharing and using data. Companies such as Google, Facebook, and Amazon have all built empires atop the data economy.

In the current thesis, we will be concerned about the usage of data in the science of Data analytics and Machine Learning. The two fields are highly correlated.

Data analytics is the process of analyzing raw data in order to draw out meaningful, actionable insights, which are then used to inform and drive smart business decisions. A data analyst will extract raw data, organize it, and then analyze it, transforming it from incomprehensible numbers into coherent, intelligible information. Having interpreted the data, the data analyst will then pass on their findings in the form of suggestions or recommendations about what a company's next steps should be.

Machine learning is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy. Through the use of statistical methods, algorithms are trained to make classifications or predictions, and to uncover key insights in data mining projects. These insights subsequently drive decision making within applications and businesses, ideally impacting key growth metrics.

1.2 Our task

Analyzing vast volumes of data or training large Machine Learning models are examples of tasks that require vast computational resources, that the average person do not have at their disposal. The best and most common way to overcome this obstacle is to use the Cloud. Cloud computing is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing. Instead of buying, owning, and maintaining physical data centers and servers, you can access technology services, such as computing power, storage, and databases, on an as-needed basis from a cloud provider. The most popular cloud providers are Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft

Azure.

The processes that we will execute during this thesis, will take the form of containerized applications. Therefore, we are in need of a container orchestrator. In our case, we will use Kubernetes [1], also known as K8s. To execute our Machine Learning workflow, we will install Kubeflow [2], a Machine Learning framework, specifically built for Kubernetes. These workflows take the form of Directed Acyclic Graphs. Therefore, they are easy to understand and to debug. Last but not least, we will deploy an independent Apache Hadoop [3] cluster to store our data, and will use Apache Spark [4] to process them.

1.3 Thesis Organization

In this section we jointly outline the organization of this thesis:

- **Chapter 2**: We place the theoretical foundations required for a reader to fully comprehend the subjects mentioned in this thesis.
- **Chapter 3**: We describe in detail the steps we followed in order to install the required software and configure the cluster.
- **Chapter 4**: We analyze the various experiments we performed on our system.
- **Chapter 5**: The conclusion to this thesis and some ideas on how our work could be expanded.

2 Theoretical background

2.1 Containers

2.1.1 Overview

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. This application is known as a containerized application. Before we had containers, applications ran on physical servers or in virtual machines. Containers are just the next iteration of how we package and run apps. As such, they're faster, more lightweight, and more suited to modern business requirements than servers and virtual machines.

A container *image* is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, run-time, system tools, system libraries and settings. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging. Using application containerization is used to deploy and run distributed applications without launching an entire virtual machine for each app. Multiple isolated applications or services run on a single host and access the same OS kernel. Containers work on bare-metal systems, cloud instances and virtual machines, across Linux and select Windows and Mac OS operating systems.

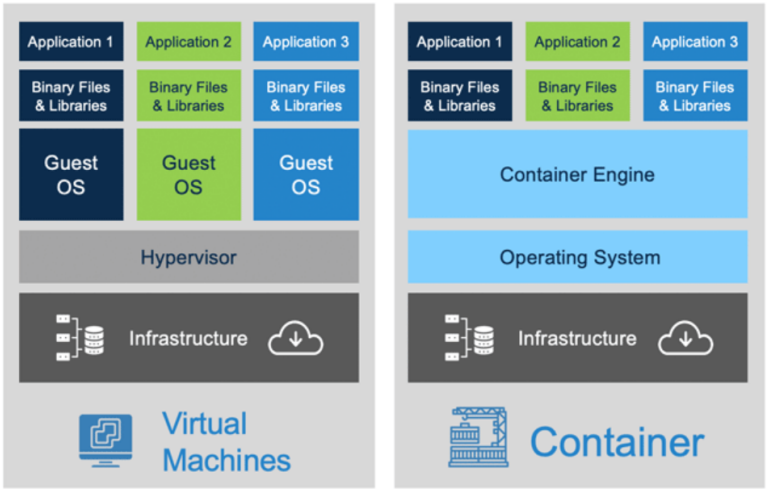


Figure 1: Containers and Virtual Machines

It is important to understand the difference between containers and Virtual Machines and the reason why enterprises are rapidly adopting containerization

as a superior approach to application development and management. As we can see in Figure 1, a virtual machine includes an entire operating system as well as the application. A physical server running three virtual machines would have a hypervisor and three separate operating systems running on top of it. By contrast a server running three containerized applications runs a single operating system, and each container shares the operating system kernel with the other containers. Shared parts of the operating system are read-only, while each container has its own mount for writing. That means the containers are much more lightweight and use far fewer resources than virtual machines.

The advantages that the containers offer can be summarized in the following points:

- **Portability**: A container creates an executable package of software that is abstracted away from the host operating system, and hence, is portable and able to run uniformly and consistently across any platform or cloud. Therefore the containers are a solution to the problem of how to get software to run reliably when moved from one computing environment to another. This could be from a developer's laptop to a test environment, from a staging environment into production, and perhaps from a physical machine in a data center to a virtual machine in a private or public cloud.

- **Speed**: We already mentioned that containers are more "lightweight," in comparison to Virtual Machines. This occurs because they share the machine's operating system (OS) kernel and are not bogged down with this extra overhead. Not only does this drive higher server efficiencies, it also reduces server and licensing costs while speeding up start-times as there is no operating system to boot.

- **Fault isolation**: Much like Virtual Machines, each containerized application is isolated and operates independently of others. The failure of one container does not affect the continued operation of any other containers. Development teams can identify and correct any technical issues within one container without any downtime in other containers.

- **Efficiency**: Software running in containerized environments shares the machine's OS kernel, and application layers within a container can be shared across containers. Thus, containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This allows for higher server efficiencies, reducing server and licensing costs.

- **Ease of management** : A container orchestration platform automates the installation, scaling, and management of containerized workloads and services. Container orchestration platforms can ease management tasks such as scaling containerized apps, rolling out new versions of apps, and providing monitoring, logging and debugging, among other functions.
- **Security** : The isolation of applications as containers inherently prevents the invasion of malicious code from affecting other containers or the host system. Additionally, security permissions can be defined to automatically block unwanted components from entering containers or limit communications with unnecessary resources.

Due to the aforementioned benefits, containers have become the de facto compute units of modern cloud-native applications.

2.1.2 Container Engines

The rapid growth in interest and usage of container-based solutions has led to the need for standards around container technology and the approach to packaging software code. The Open Container Initiative (OCI), established in June 2015 by Docker and other industry leaders, is promoting common, minimal, open standards and specifications around container technology. Because of this, the OCI is helping to broaden the choices for open source engines. Users will not be locked into a particular vendor's technology, but rather they will be able to take advantage of OCI-certified technologies that allow them to build containerized applications using a diverse set of DevOps tools and run these consistently on the infrastructure(s) of their choosing. Adopting and leveraging OCI specifications as these evolve will ensure that solutions are vendor-neutral, certified to run on multiple operating systems and usable in multiple environments.

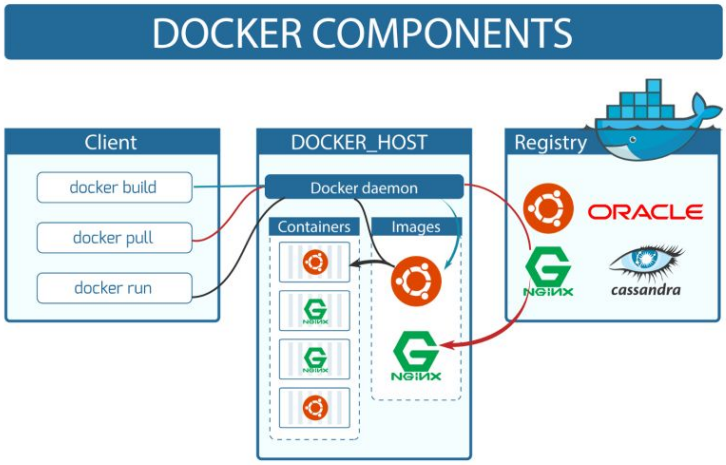


Figure 2: Containers and Virtual Machines

Today, Docker [6] is one of the most well-known and highly used container engine technologies. Docker is basically an operating system for containers. There are other container engines, but Docker is usually preferable, since it offers numerous benefits, in comparison to its rivals:

- **Improved and seamless container portability**: Docker containers run without modification across any desktop, data center and cloud environment.
- **Lightweight and granular updates**: Multiple processes can be combined within a single container. This makes it possible to build an application that can continue running while one of its parts is taken down for an update or repair.
- **Automated container creation:** Docker can automatically build a container based on application source code.
- **Container versioning**: Docker can track versions of a container image, roll back to previous versions, and trace who built a version and how. It can even upload only the deltas between an existing version and a new one.
- **Container reuse**: Existing containers can be used as base images—essentially like templates for building new containers.
- **Shared container libraries**: Developers can access an open-source registry containing thousands of user-contributed containers.

Finally, we will describe some of the tools, terms and technologies developers encounter when using Docker:

- **Docker images**: Docker images contain executable application source code as well as all the tools, libraries, and dependencies that the application code needs to run as a container. When you run the Docker image, it becomes one instance (or multiple instances) of the container. An instance describes the state that a container should be in, when it is initialized. An instance contains:
  - A complete image of the file system inside the container.
  - The processes that will be executed inside the container, once it is initialized, the directory it will be executed from, various environment variables and more.
- **DockerFile**: Every Docker container starts with a file containing instructions for how to build the Docker container image. The DockerFile automates the process of Docker image creation. It's essentially a list of command-line interface (CLI) instructions that Docker Engine will run in order to assemble the image. It's possible to build a Docker image from scratch, but most developers pull them down from common repositories, such as Docker Hub [7]. Multiple Docker images can be created from a single base image, and they'll share the commonalities of their stack. Docker images are made up of layers, and each layer corresponds to a version of the image. Whenever a developer makes changes to the image, a new top layer is created, and this top layer replaces the previous top layer as the current version of the image. Previous layers are saved for rollbacks or to be re-used in other projects.
- **Docker containers**: Docker containers are the live, running instances of Docker images. While Docker images are read-only files, containers are live, ephemeral, executable content. Users can interact with them, and administrators can adjust their settings and conditions using Docker commands.

2.2 Kubernetes

2.2.1 Overview

In order to execute these applications on a cluster of multiple machines such as a cloud infrastructure, we need Kubernetes. Kubernetes is an application orchestrator. For the most part, it orchestrates containerized cloud-native apps. An orchestrator is a system that deploys and manages applications. It can deploy applications and dynamically respond to changes. For example, Kubernetes can:

- Deploy an application

- Scale it up and down dynamically based on demand
- Self-heal it when things break
- Perform zero-downtime rolling updates and rollbacks

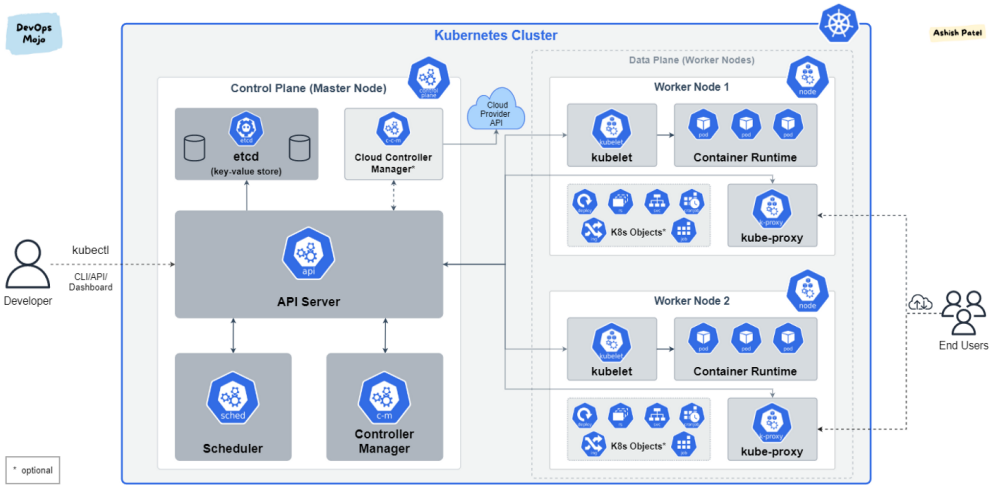


Figure 3: Kubernetes architecture

A cloud-native application is one that's designed to meet cloud-like demands of auto-scaling, self-healing, rolling updates, rollbacks and more. It's important to be clear that cloud-native apps are not applications that will only run in the public cloud. Yes, they absolutely can run on public clouds, but they can also run anywhere that you have Kubernetes, even an on-premises data center. So, cloud-native is about the way applications behave and react to events. In many ways, Kubernetes is like an operating system (OS) for the cloud.

2.2.2 Kubernetes and Container Engines

Kubernetes and a container engine like Docker are *complementary* technologies. Docker has tools that build and package applications as container images. It can also run containers. Kubernetes can't do either of those things. Instead, Kubernetes operates at a higher level providing orchestration services such as self-healing, scaling and updates.

It's common practice to use Docker for build-time tasks such as packaging apps as containers, but then use a combination of Kubernetes and Docker to run them. In this model, Kubernetes performs high-level orchestration tasks such as self-healing, scaling and rolling updates, but it needs a tool like Docker to perform low-level tasks such as starting and stopping containers.

Assume you have a Kubernetes cluster with 10 nodes to run your production applications. Behind the scenes, each cluster node is running Docker as its container run-time. This means Docker is the low-level technology that starts and stops the containerised applications. Kubernetes is the higher-level technology that looks after the bigger picture, such as deciding which nodes to run containers on, deciding when to scale up or down, and executing updates.

2.2.3 The cluster

A Kubernetes cluster is like any other cluster – a number of machines to host applications. We call these machines "nodes", and they can be physical servers, virtual machines, cloud instances, Raspberry Pis, and more. A Kubernetes cluster is made of a control plane and worker nodes :

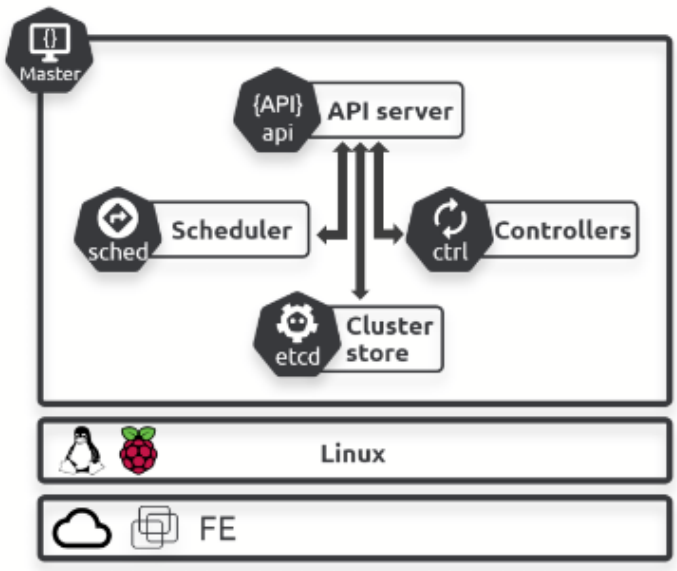


Figure 4: Structure of a control plane node

- **Control Plane**: A Kubernetes control plane node is a server running collection of system services that make up the control plane of the cluster. It is frequently called Master or Head node. The simplest setups run a single control plane node. However, this is only suitable for labs and test environments. For production environments, multiple control plane nodes configured for high availability (HA) is vital. Generally speaking, 3 or 5 is recommended for HA. It's also considered a good practice not to run user applications on control plane nodes. This frees them up to concentrate entirely on managing the cluster. The control plane is comprised of certain services:

- **The API server (kube-apiserver)**: All communication, between all components, must go through the API server. Much like the Dockerfiles that we use to define container images, there are YAML files, also known as manifests, that describe the desired state of an application. This desired state includes things like which container image to use, which ports to expose, and how many replicas to run. All requests to the API server are subject to authentication and authorization checks. Once these are done, the config in the YAML file is validated, persisted to the cluster store, and work is scheduled to the cluster.
- **Cluster store (etcd)**: The cluster store persistently stores the entire configuration and state of the cluster. As such, it's a vital component of every Kubernetes cluster. The cluster store is currently based on etcd, a popular distributed database. Considering its importance, you should run between 3-5 etcd replicas for high-availability, and you should provide adequate ways to recover when things go wrong. A default installation of Kubernetes installs a replica of the cluster store on every control plane node and automatically configures HA.
- **Controller manager (kube-controller-manager)**: The controller manager implements all the background controllers that monitor cluster components and respond to events. Its purpose is to ensure the observed state of the cluster matches the desired state. Each controller fullfils an specific purpose.
- **Scheduler (kube-scheduler)**: The scheduler watches the API server for new work tasks and assigns them to appropriate healthy worker nodes. Its task is to identify whether a node is capable of executing a task. When identifying nodes capable of running a task, the scheduler performs various predicate checks. Any node incapable of running the task is ignored, and those remaining are ranked according to parameters, such as its free resources and its current workload. After this evaluation, the most suitable node is selected to run the task. If the scheduler does not find a suitable node, the task is not scheduled and gets marked as *Pending*. The scheduler is not responsible for running tasks, just picking the nodes to run them. Running the tasks is the work of Pods, of which we will talk about in depth later on.

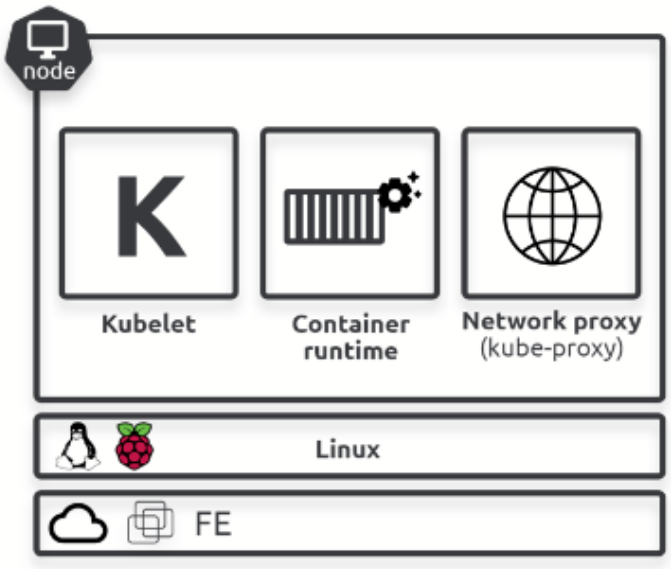


Figure 5: Structure of a worker node

- **Worker Nodes**: These nodes are servers that are the workers of a Kubernetes cluster. They have three main tasks :

  1. Watch the API server for new work assignments
  2. Execute them
  3. Report back to the control plane (via the API server)

  The worker node is comprised of three major components:

  – **Kubelet**: The kubelet is main Kubernetes agent and runs on every cluster node. When you join a node to a cluster, the process installs the kubelet, which is then responsible for registering it with the cluster. This process registers the node's CPU, memory, and storage into the wider cluster pool. One of the main jobs of the kubelet is to watch the API server for new work tasks. Any time it sees one, it executes the task and maintains a reporting channel back to the control plane. If a kubelet is incapable of running a task, it reports back to the control plane and lets the control plane decide what actions to take.

  – **Container engine**: The kubelet needs a container run-time to perform container-related tasks – things like pulling images and starting and stopping containers. In the early days, Kubernetes had native support for Docker. However, it is worth mentioning that Kubernetes 1.20 deprecated Docker as a engine. Container images created by Docker will continue to work as normal, and this won't change. But a future release of Kubernetes will end support of Docker as a engine. To

streamline the deprecation process, many Kubernetes clusters already ship with containerd as the default engine. Containerd is effectively a stripped-down version of Docker with just the components that Kubernetes needs.

- **Kube-proxy**: Kube-proxy runs on every node and is responsible for local cluster networking. It ensures each node gets its own unique IP address, and it implements local iptables or IPVS rules to handle routing and load-balancing of traffic on the Pod network.

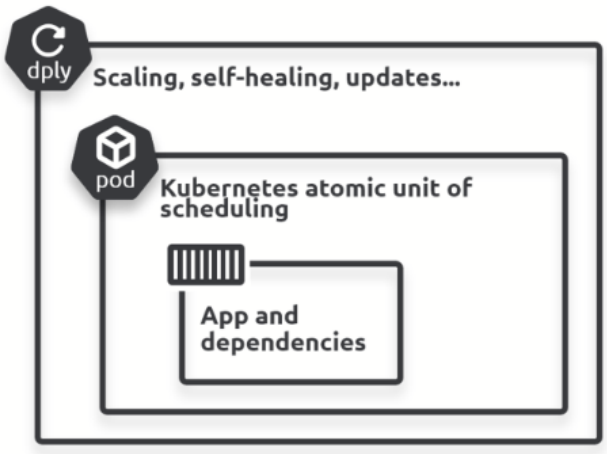


Figure 6: A containerized application inside a Pod

2.2.4 Pods

For an application to be deployed on a Kubernetes cluster, it must pass through certain stages:

1. You program the application in a coding language of your choice.
2. Secondly, you build it into a container image and store it in a registry. At this point, the application service is containerized.
3. Next, this container must be placed in a Pod
4. Finally, the application will be deployed on the cluster, by submitting a declarative manifest file (YAML) to the API Server.

Kubernetes demands that every container runs inside a Pod. In simple terms, a Pod is just a wrapper that allows a container to run on a Kubernetes cluster. The simplest model is to run a single container in every Pod. This is why we often use the terms “Pod” and “container” interchangeably. However, there

are advanced use-cases that run multiple containers in a single Pod. Once you've defined the Pod, you're ready to deploy the app to Kubernetes.

At the highest-level, a Pod is an environment to run containers. Pods themselves don't actually run applications – applications always run in containers. Pods ring-fence an area of the host OS, build a network stack, create a number of kernel namespaces, and run one or more containers. If you're running multiple containers in a Pod, they all share the same Pod environment. This includes the network stack, volumes, IPC namespace, shared memory, and more. As an example, this means all containers in the same Pod will share the same IP address (the Pod's IP). There is also a localhost interface, that allows containers in the same Pods, to communicate directly with each other.

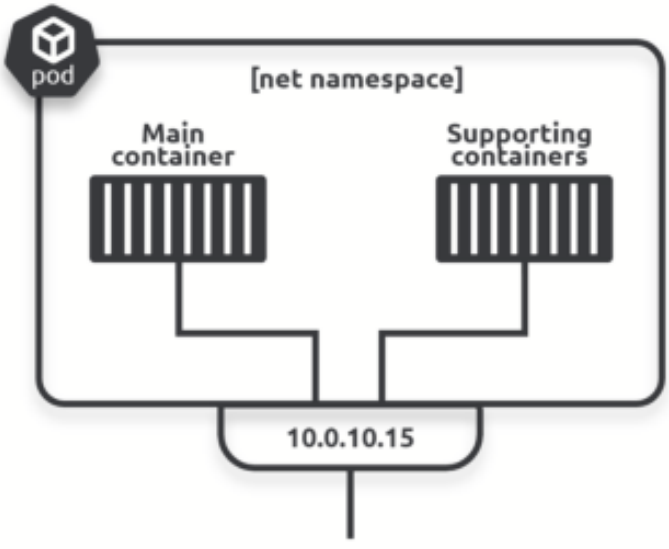


Figure 7: The anatomy of a Pod

Pods are the minimum unit of scheduling in Kubernetes. If you need to scale an app, you add or remove Pods. You do not scale by adding more containers to existing Pods. Multi-container Pods are only for situations where two different, but complimentary, containers need to share resources.

Lastly, we need to mention two important characteristics of a Pod :

- **Pods are ephemeral:** In other words, they're created, they perform certain tasks, and once finished, they are destroyed. If they are destroyed unexpectedly, Kubernetes starts a new one in its place. However, even though the new Pod looks nearly identical to the old one, it has a different ID and IP address.
- **Pods are immutable:** Once a Pod is running, you never change its con-

figuration. If you need to change or update it, you replace it with new one running the new configuration, by deleting the old one and deploying the new one.

2.2.5 Deployment Manifests

A Kubernetes deployment manifest is a YAML or JSON file that defines how to deploy a set of replicas of a particular application to a Kubernetes cluster. Here are some of the key components that can be specified in a Kubernetes deployment manifest:

- **apiVersion**: The version of the Kubernetes API to use.
- **kind**: The kind of resource being defined in the manifest. In the case of a deployment, this would be Deployment.
- **metadata**: Information about the deployment, including its name and any labels or annotations that should be applied to it.
- **spec**: The desired state of the deployment, including the number of replicas to run, the container image to use, and the resources to allocate to the replicas.
- **replicas**: The number of replicas of the application to run.
- **selector**: A label query used to identify the pods that belong to the deployment.
- **template**: A description of the pods that the deployment should create. This includes the container image to use, the ports to expose, and any environment variables or volumes to set.

Other components that may be included in a deployment manifest include liveness and readiness probes, resource limits and requests, and rolling update policies. Here is an example of a Kubernetes deployment manifest in YAML format:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: my-app
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
```

```
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:latest
        ports:
        - containerPort: 80
```

This deployment manifest creates a deployment named "my-app" that runs three replicas of a container based on the image "my-app:latest", exposed on port 80. The replicas are identified by the label "app: my-app", which is applied to the replicas and used in the deployment's selector to determine which pods belong to the deployment.

2.2.6 Storage

In Kubernetes, persistent storage is provided through the use of persistent volumes (PVs) and persistent volume claims (PVCs). PVs are resources that provide raw block storage that can be mounted as a filesystem on a pod. They are typically backed by a physical storage device, such as a disk or a networked storage system. PVCs are requests for a specific amount of storage from a PV. When a PVC is created, the Kubernetes control plane looks for a PV with sufficient capacity and the right access modes and binds the PVC to it. Pods can then mount the PVC as a volume, providing persistent storage for the pod's containers.

There are several types of persistent volumes that can be used in Kubernetes, including:

- **HostPath/Local**: A PV that uses a local directory on the node as the backing store.
- **NFS**: A PV that uses an NFS server as the backing store.
- **iSCSI**: A PV that uses an iSCSI target as the backing store.
- **Ceph RBD**: A PV that uses a Ceph Rados block device as the backing store.
- **GCEPersistentDisk**: A PV that uses a Google Compute Engine persistent disk as the backing store.

In the current thesis, we will use the first. In addition to PVs and PVCs, Kubernetes also provides support for dynamic provisioning of persistent storage. This allows storage to be automatically provisioned and attached to a cluster when a PVC is created, without the need to pre-create PVs. This can be useful in situations where storage needs are unpredictable or vary over time.

2.2.7 Grafana & Prometheus

Monitoring a Kubernetes Cluster is the need of the hour for any application following a microservices architecture. There are many solutions that one can implement to monitor their Kubernetes workload. The one we will deploy is *Prometheus* [8] and *Grafana* [9].

Kubernetes is a complex system with many moving parts. Effective monitoring of such a dynamic system requires tools with advanced capabilities. Prometheus is one such application. Prometheus is an open-source monitoring solution that can collect metrics from a number of target systems. It collects and stores the metrics as time-series data. It also features an excellent alerting mechanism that can integrate with popular team collaboration and incident management tools.

Grafana is an open-source visualization tool used to monitor infrastructure in real-time. The visualization of data through graphs helps in log analysis and troubleshooting real-time infra issues. In our case, the data source of Grafana will be the metrics collected by Prometheus.

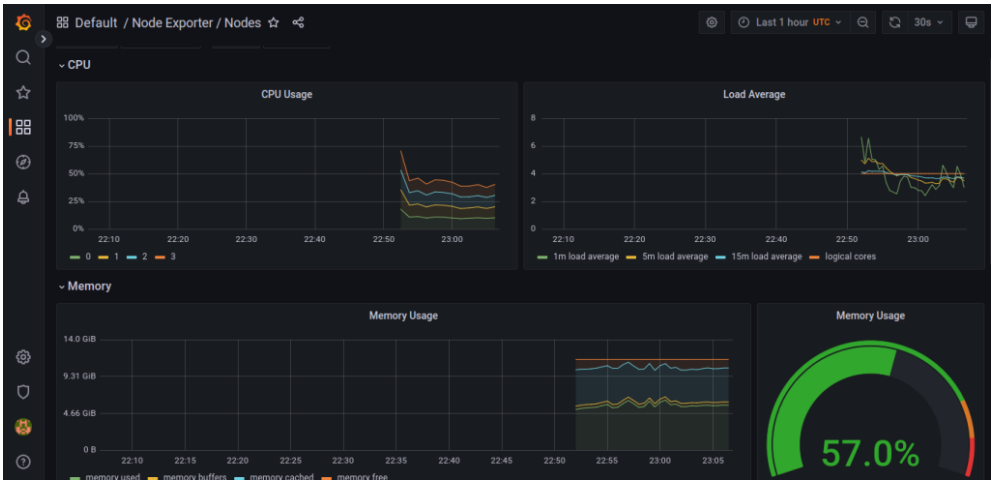


Figure 8: A Grafana Dashboard

2.2.8 Kubespray

Kubespray is an open-source tool for deploying and managing Kubernetes

clusters. It uses Ansible [10] to automate the deployment and configuration of Kubernetes clusters on multiple cloud platforms or on-premises. Ansible is an open-source software platform for automating and configuring computer systems. It can be used for a wide range of tasks, such as provisioning and maintaining servers, deploying applications, and orchestrating complex workflows. Ansible is written in Python and can be used to manage infrastructure on-premises and in the cloud, and it can be used to manage systems running Linux, Windows, and other operating systems.

One of the main benefits of using Kubespray is that it allows users to deploy and manage Kubernetes clusters in a consistent, repeatable way and it supports multiple Linux distributions, including Ubuntu, CentOS, and Debian. It provides a set of Ansible playbooks that can be used to deploy and configure a cluster, and it also includes a number of tools and scripts for tasks such as upgrading clusters, scaling clusters, and rolling back failed updates. An Ansible playbook is a file that specifies a series of tasks to be executed by Ansible. Playbooks are written in YAML and are a way to define the infrastructure and configuration of a system.

In addition to its core functionality, Kubespray also includes a number of optional add-ons and integrations that can be used to extend the capabilities of the deployed cluster. These include support for monitoring tools like Prometheus and Grafana, logging tools like ELK, and tools for managing secrets and certificates like Hashicorp Vault.

2.3 Machine Learning

2.3.1 Introduction

Our imagination has long been captivated by visions of machines that can learn and imitate human intelligence. Nowadays, software programs that can acquire new knowledge and skills through experience are becoming increasingly common. We use such machine learning programs to discover new music that we enjoy, and to quickly find the exact shoes we want to purchase online. Machine learning programs allow us to dictate commands to our smartphones and allow our thermostats to set their own temperatures. Machine learning programs can decipher sloppily-written mailing addresses better than humans and guard credit cards from fraud more vigilantly. From investigating new medicines to estimating the page views for versions of a headline, machine learning software is becoming central to many industries. Machine learning has even encroached on activities that have long been considered uniquely human, such as writing columns in a

newspaper.

When most people hear “Machine Learning,” they picture some kind of highly-advanced robot. But Machine Learning is not just a futuristic fantasy, it’s already here. In fact, it has been around for decades in some specialized applications, such as Optical Character Recognition (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: it was the spam filter. By observing thousands of emails that have been previously labeled as either spam or ham, spam filters learn to classify new messages. It was followed by hundreds of ML applications that now quietly power hundreds of products and features that you use regularly, from better recommendations to voice search. Machine learning is the design and study of software artifacts that use past experience to make future decisions; it is the study of programs that learn from data. The fundamental goal of machine learning is to generalize, or to induce an unknown rule from examples of the rule’s application.

2.3.2 Machine Learning Tasks

Machine learning tasks can be split into two major categories, supervised and unsupervised learning.

The majority of practical machine learning uses *supervised learning*. Supervised learning is where you have input variables (x) and an output variable (Y) and you use an algorithm to learn the mapping function from the input to the output (Y = f(x)). The goal is to approximate the mapping function so well that when you have new input data (x) that you can predict the output variables (Y) for that data. It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers, the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance. In Supervised learning, you train the machine using data which is well "labeled." It means some data is already tagged with the correct answer. A supervised learning algorithm learns from labeled training data, helps you to predict outcomes for unforeseen data. Supervised learning can be split into two main subcategories:

- Classification: A classification problem is when the output variable is a category, such as “red” or “blue” or “disease” and “no disease”.

- Regression: A regression problem is when the output variable is a real value, such as "dollars" or "weight".

Unsupervised learning is where you only have input data (X) and no corresponding output variables. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn more about the data. These are called unsupervised learning because unlike supervised learning there are no correct answers and there is no teacher. Algorithms are left to their own devices to discover and present the underlying structure in the data. Unsupervised learning can be split into two main subcategories:

- Clustering: It mainly deals with finding a structure or pattern in a collection of uncategorized data. Clustering algorithms will process your data and find natural clusters(groups) if they exist in the data. You can also modify how many clusters your algorithms should identify. It allows you to adjust the granularity of these groups.
- Association: This unsupervised technique is about discovering interesting relationships between variables in large databases. For example, people that buy a new home most likely to buy new furniture.

For this study, we are interested in classification problems.

2.3.3 Training data and test data

The observations in the *training set* comprise the experience that the algorithm uses to learn. In supervised learning problems, each observation consists of an observed response variable and one or more observed explanatory variables.

The *test set* is a similar collection of observations that is used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set. If the test set does contain examples from the training set, it will be difficult to assess whether the algorithm has learned to generalize from the training set or has simply memorized it. A program that generalizes well will be able to effectively perform a task with new data. In contrast, a program that memorizes the training data by learning an overly complex model could predict the values of the response variable for the training set accurately, but will fail to predict the value of the response variable for new examples.

Memorizing the training set is called *overfitting*. A program that memorizes its observations may not perform its task well, as it could memorize relations and

structures that are noise or coincidence. Balancing memorization and generalization, or over-fitting and under-fitting, is a problem common to many machine learning algorithms.

In addition to the training and test data, a third set of observations, called a *validation or hold-out set*, is sometimes required. The validation set is used to tune variables called *hyperparameters* (learning rate, batch size e.t.c.), which control how the model is learned. The program is still evaluated on the test set to provide an estimate of its performance in the real world; its performance on the validation set should not be used as an estimate of the model's real-world performance since the program has been tuned specifically to the validation data. It is common to partition a single set of supervised observations into training, validation, and test sets. There are no requirements for the sizes of the partitions, and they may vary according to the amount of data available. It is common to allocate 50% or more of the data to the training set, 25% to the test set, and the remainder to the validation set.

2.3.4 Introduction to Neural Networks

The most successful model in the context of pattern recognition is the feed-forward neural network. The term 'neural network' has its origins in attempts to find mathematical representations of information processing in biological systems (McCulloch and Pitts, 1943; Widrow and Hoff, 1960; Rosenblatt, 1962; Rumelhart et al., 1986). Indeed, it has been used very broadly to cover a wide range of different models, many of which have been the subject of exaggerated claims regarding their biological plausibility. From the perspective of practical applications of pattern recognition, however, biological realism would impose entirely unnecessary constraints.

Birds inspired us to fly, burdock plants inspired velcro, and nature has inspired many other inventions. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the key idea that inspired artificial neural networks (ANNs). However, although planes were inspired by birds, they don't have to flap their wings. Similarly, ANNs have gradually become quite different from their biological counterparts. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying "units" rather than "neurons"), lest we restrict our creativity to biologically plausible systems

ANNs are at the very core of Deep Learning. They are versatile, powerful and

scalable, making them ideal to tackle large and highly complex Machine Learning tasks, such as classifying billions of images (e.g. Google Images), powering speech recognition services (e.g. Apple's Siri) or recommending the best videos to watch to hundreds of millions of users every day (e.g. YouTube).

The artificial neural networks, that will concern us, are powerful non linear models for classification and regression that use a different strategy to overcome the perceptron's limitations. Artificial neural networks are described by three components. The first is the model's architecture, or topology, which describes the layers of neurons and structure of the connections between them. The second component is the activation function used by the artificial neurons. The third component is the learning algorithm that finds the optimal values of the weights.

There are two main types of artificial neural networks:

- Feed-forward neural networks are the most common type of neural net, and are defined by their directed acyclic graphs. Signals only travel in one direction—towards the output layer—in feed-forward neural networks. Feed-forward neural networks are commonly used to learn a function to map an input to an output.

- Feed-back neural networks, or recurrent neural networks, do contain cycles. The feed-back cycles can represent an internal state for the network that can cause the network's behavior to change over time based on its input. The temporal behavior of feed-back neural networks makes them suitable for processing sequences of inputs.

The multi-layer perceptron (MLP) is the one of the most commonly used artificial neural networks. The name is a slight misnomer; a multi-layer perceptron is not a single perceptron with multiple layers, but rather multiple layers of artificial neurons that can be perceptrons. The layers of the MLP form a directed, acyclic graph. Generally, each layer is fully connected to the subsequent layer; the output of each artificial neuron in a layer is an input to every artificial neuron in the next layer towards the output. MLPs have three or more layers of artificial neurons.The input layer consists of simple input neurons. The input neurons are connected to at least one hidden layer of artificial neurons. The hidden layer represents latent variables; the input and output of this layer cannot be observed in the training data. Finally, the last hidden layer is connected to an output layer.

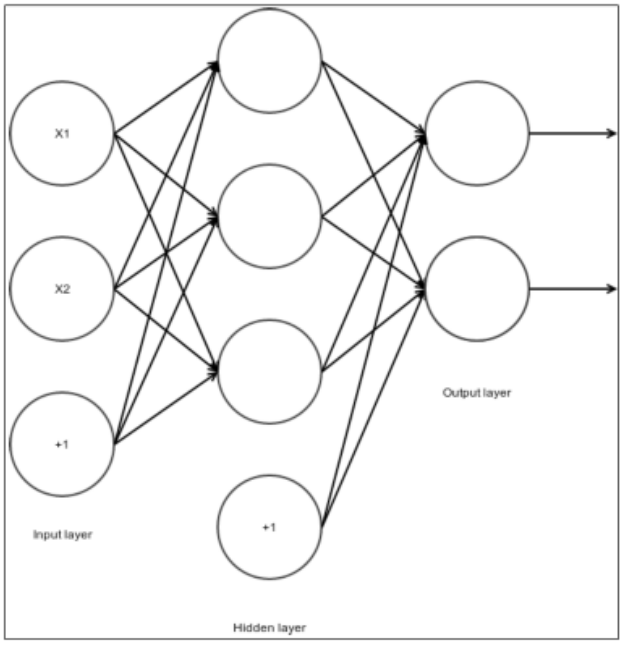


Figure 9: A simple neural network with two inputs, two outputs and one hidden layer with three neurons

The artificial neurons, or **units**, in the hidden layer commonly use non linear activation functions such as the hyperbolic tangent function and the logistic function, which are given by the following equations respectively:

$$f(x) = \tanh(x)$$

$$f(x) = \frac{1}{1+e^{-x}}$$

Figure 10: The most common activation functions

As with other supervised models, our goal is to find the values of the weights that minimize the value of a cost function. The mean squared error cost function is commonly used with multi-layer perceptrons. It is given by the following equation, where m is the number of training instances:

$$MSE = \frac{1}{m}\sum_{i=1} m\left(y_i - f(x_i)\right)^2$$

Figure 11: The most common loss function

Obviously, there are many other activation and loss functions, each suited to specific applications. The reason we wish to minimize the aforementioned loss

function is because we want the difference between the label y_i (or target) of the sample and the response $f(x_i)$ of the network (or prediction) about the sample to be a small as possible. In other words, we want for y_i to be equal to $f(x_i)$, so that the network classifies the samples correctly at (almost) all times. In the next section, we will discuss the process with which we accomplish this.

2.3.5 Deep Learning

Deep neural networks, which are remarkably effective for many machine learning tasks, define parameterized functions from inputs to outputs as compositions of many layers of basic building blocks, such as affine transformations and simple non linear functions. Commonly used examples of the latter are sigmoids and rectified linear units (ReLUs). By varying parameters of these blocks, we can "train" such a parameterized function with the goal of fitting any given finite set of input/output examples.

Deep learning aims to extract complex features from high-dimensional data and use them to build a model that relates inputs to outputs (e.g., classes). Deep learning architectures are usually constructed as multi-layer networks so that more abstract features are computed as non linear functions of lower-level features. We mainly focus on supervised learning, where the training inputs are labeled with correct classes, but in principle our approach can also be used for unsupervised, privacy-preserving learning, too. Multi-layer neural networks (2 hidden layers or more) are the most common form of deep learning architectures.

In general, the values computed in higher layers represent more abstract features of the data. The first layer is composed of the raw features extracted from the data, e.g., the intensity of colors in each pixel in an image or the frequency of each word in a document. The outputs of the last layer correspond to the abstract answers produced by the model. If the neural network is used for classification, these abstract features also represent the relation between input and output. The non linear function f and the weight matrices determine the features that are extracted at each layer. The main challenge in deep learning is to automatically learn from training data the values of the parameters (weight matrices) that maximize the objective of the neural network (e.g., classification accuracy).

2.3.6 Learning Process - Minimizing the cost function

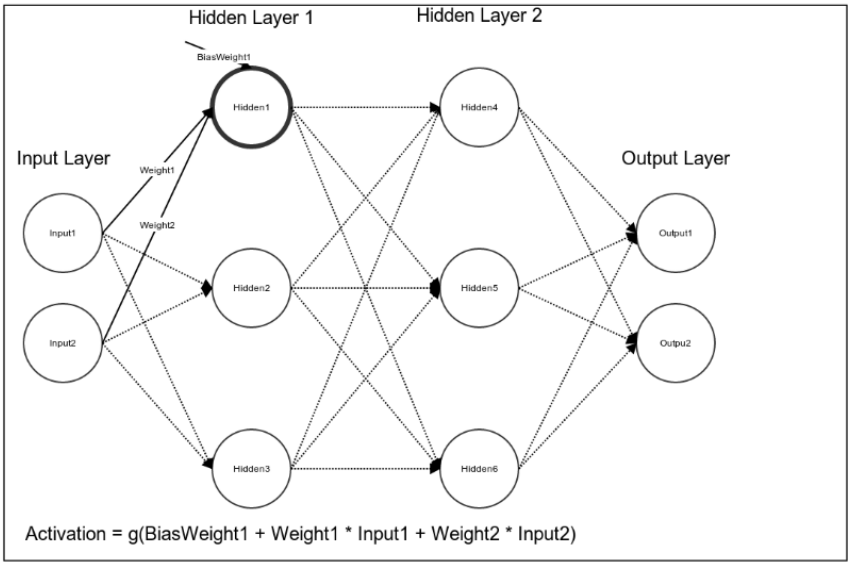
Learning the parameters of a neural network is a non linear optimization problem. In supervised learning, the objective function is the output of the neural

network. The algorithms that are used to solve this problem are typically variants of *gradient descent*. Simply put, gradient descent starts at a random point (set of parameters for the neural network), then, at each step, computes the gradient of the non linear function being optimized and updates the parameters so as to decrease the gradient. This process continues until the algorithm converges to a local optimum.

In a neural network, the gradient of each weight parameter is computed through feed-forward and back-propagation procedures. Feed-forward sequentially computes the output of the network given the input data and then calculates the error, i.e., the difference between this output and the true value of the function. Back-propagation propagates this error back through the network and computes the contribution of each neuron to the total error. The gradients of individual parameters are computed from the neurons' activation values and their contribution to the error.

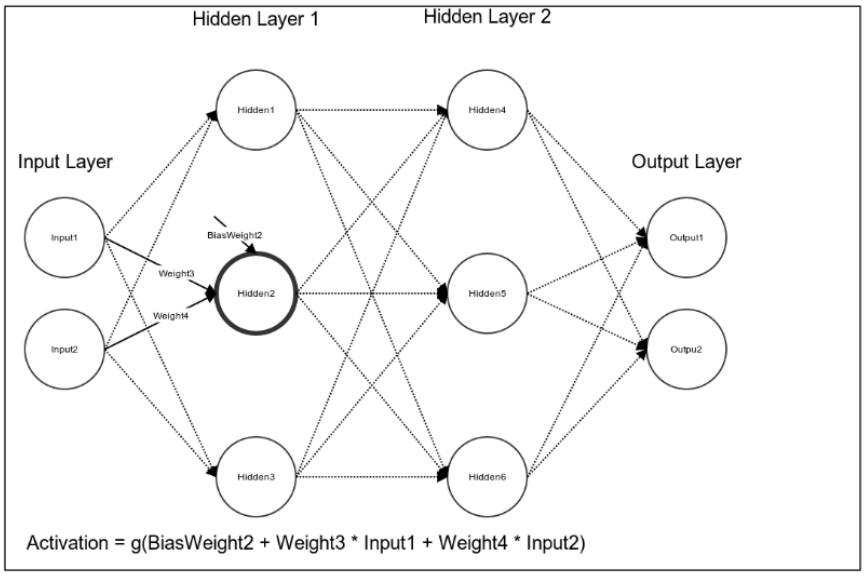
2.3.7 Forward propagation

During the forward propagation stage, the features are input to the network and fed through the subsequent layers to produce the output activations. First, we compute the activation for the unit Hidden1. We find the weighted sum of input to Hidden1, and then process the sum with the activation function. Note that Hidden1 receives a constant input from a bias unit that is not depicted in the diagram in addition to the inputs from the input units. In the following diagram, g(x) is the activation function:

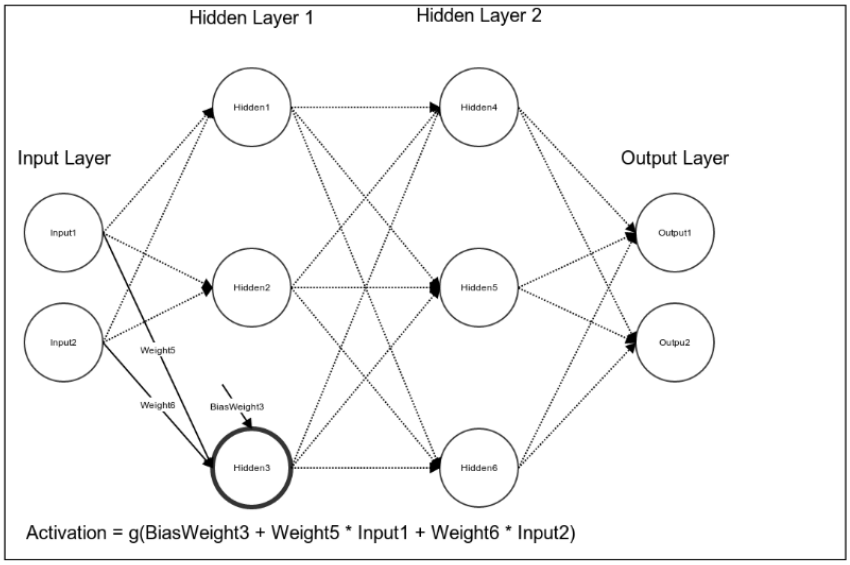


Next, we compute the activation for the second hidden unit. Like the first hidden unit, it receives weighted inputs from both of the input units and a con-

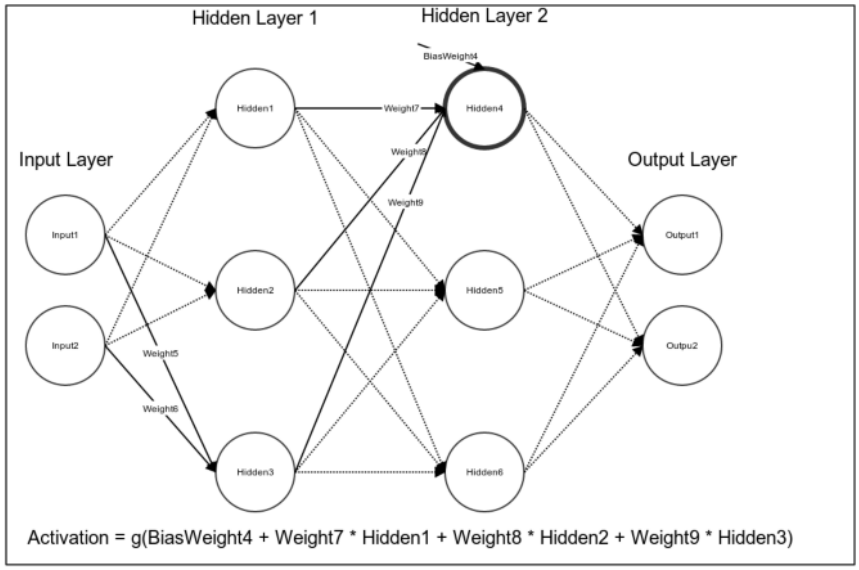
stant input from a bias unit. We then process the weighted sum of the inputs, or preactivation, with the activation function as shown in the following figure:



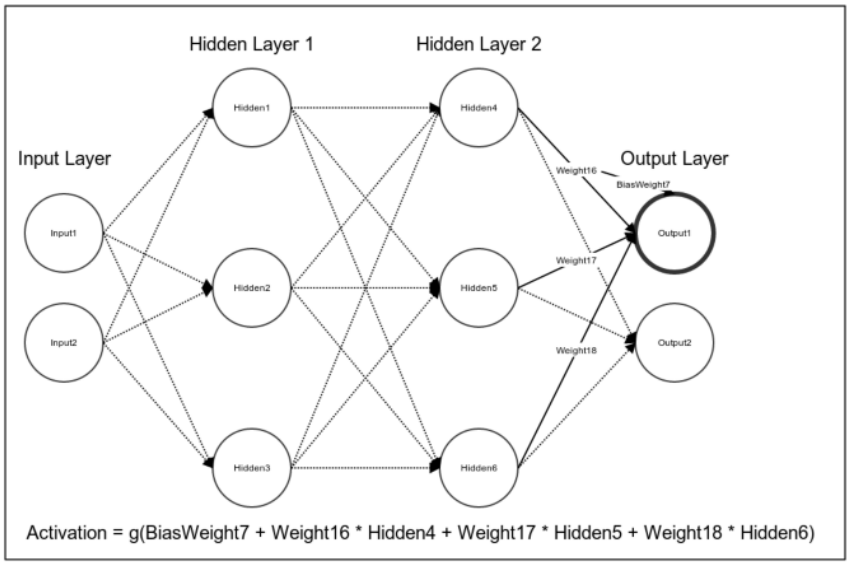
We then compute the activation for Hidden3 in the same manner:



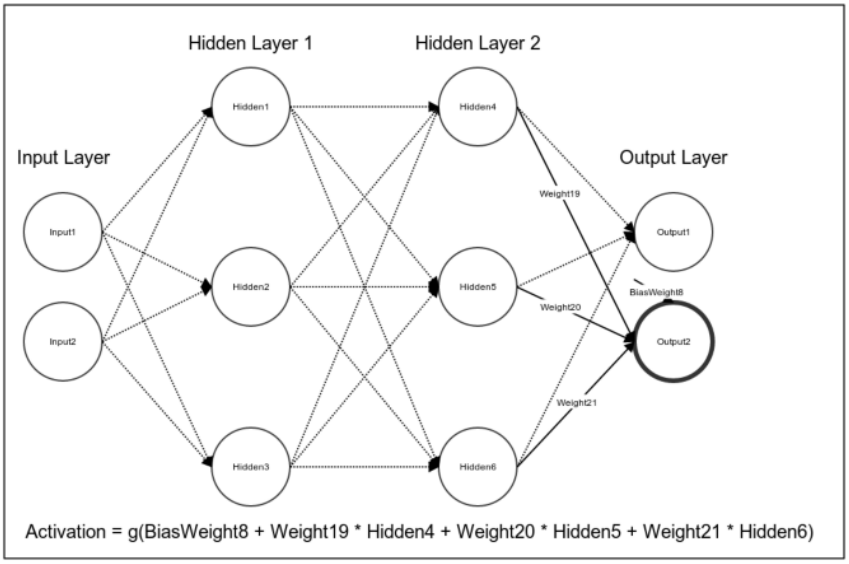
Having computed the activations of all of the hidden units in the first layer, we proceed to the second hidden layer. In this network, the first hidden layer is fully connected to the second hidden layer. Similar to the units in the first hidden layer, the units in the second hidden layer receive a constant input from bias units that are not depicted in the diagram. We proceed to compute the activation of Hidden4:



We next compute the activations of Hidden5 and Hidden6. Having computed the activations of all of the hidden units in the second hidden layer, we proceed to the output layer in the following figure. The activation of Output1 is the weighted sum of the second hidden layer's activations processed through an activation function. Similar to the hidden units, the output units both receive a constant input from a bias unit:



We calculate the activation of Output2 in the same manner:



We computed the activations of all of the units in the network, and we have now completed forward propagation. The network is not likely to approximate the true function well using the initial random values of the weights. We must now update the values of the weights so that the network can better approximate our function. We will do so using the algorithm of backpropagation.

2.3.8 Gradient Descent

In this section, we will discuss a method to efficiently estimate the optimal values of the model's parameters called gradient descent. Note that our definition of a good fit has not changed; we will still use gradient descent to estimate the values of the model's parameters that minimize the value of the cost function.

Gradient descent is sometimes described by the analogy of a blindfolded man who is trying to find his way from somewhere on a mountainside to the lowest point of the valley. He cannot see the topography, so he takes a step in the direction with the steepest decline. He then takes another step, again in the direction with the steepest decline. The sizes of his steps are proportional to the steepness of the terrain at his current position. He takes big steps when the terrain is steep, as he is confident that he is still near the peak and that he will not overshoot the valley's lowest point. The man takes smaller steps as the terrain becomes less steep. If he were to continue taking large steps, he may accidentally step over the valley's lowest point. He would then need to change direction and step toward the lowest point of the valley again. By taking decreasingly large steps, he can avoid stepping back and forth over the valley's lowest point. The blindfolded man continues to walk until he cannot take a step that will decrease his altitude; at this point, he has found the bottom of the valley.

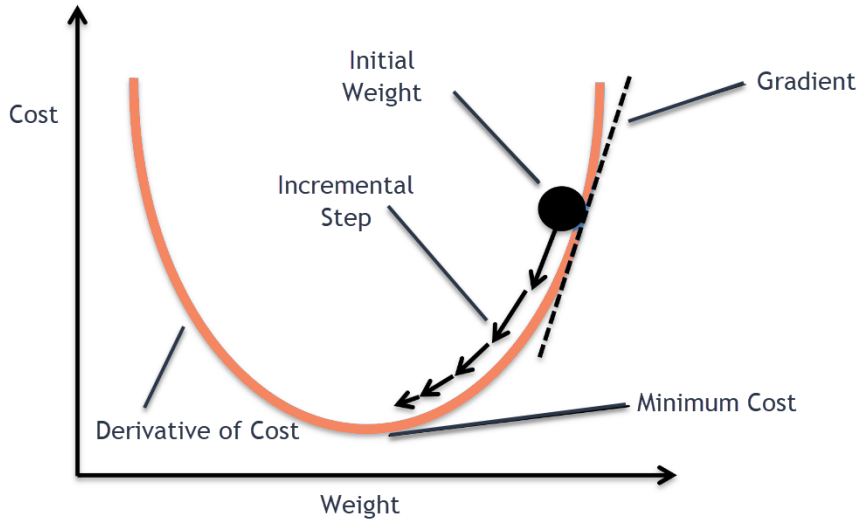


Figure 12: Finding the local optimum of an one-dimensional convex function using GD

Formally, gradient descent is an optimization algorithm that can be used to estimate the local minimum of a function. We can use gradient descent to find the values of the model's parameters that minimize the value of the cost function. Gradient descent iteratively updates the values of the model's parameters by calculating the partial derivative of the cost function at each step. It is important to note that gradient descent estimates the *local* minimum of a function. A three-dimensional plot of the values of a convex cost function for all possible values of the parameters looks like a bowl. The bottom of the bowl is the sole local minimum. Non-convex cost functions can have many local minima, that is, the plots of the values of their cost functions can have many peaks and valleys. Gradient descent is only guaranteed to find the local minimum; it will find a valley, but will not necessarily find the lowest valley. Fortunately, the residual sum of the squares cost function is convex.

An important hyperparameter of gradient descent is the *learning rate*, which controls the size of the blindfolded man's steps. If the learning rate is small enough, the cost function will decrease with each iteration until gradient descent has converged on the optimal parameters. As the learning rate decreases, however, the time required for gradient descent to converge increases; the blindfolded man will take longer to reach the valley if he takes small steps than if he takes large steps. If the learning rate is too large, the man may repeatedly overstep the bottom of the valley, that is, gradient descent could oscillate around the optimal values of the parameters.

There are two varieties of gradient descent that are distinguished by the number of training instances that are used to update the model parameters in each training iteration. The gradients of the parameters can be averaged over all available data. This algorithm, known as batch gradient descent, is not efficient, especially if learning on a large dataset. Stochastic gradient descent (SGD) is a drastic simplification which computes the gradient over an extremely small subset (mini-batch) of the whole dataset. In the simplest case, corresponding to maximum stochasticity, one data sample is selected at random in each optimization step.

Let w be the flattened vector of all parameters in a neural network, composed of W_k for every k. Let E be the error function, i.e., the difference between the true value of the objective function and the computed output of the network. The back-propagation algorithm computes the partial derivative of E with respect to each parameter in w and updates the parameter so as to reduce its gradient. The update rule of stochastic gradient descent for a parameter w_j is:

$$w_j := w_j - \alpha \frac{\partial E_i}{\partial w_j}$$

where α is the learning rate and E_i is computed over the mini-batch i. We refer to one full iteration over all available input data as an epoch. Note that each parameter in vector w is updated independently from other parameters. Some techniques set the learning rate adaptively, but still preserve this independence.

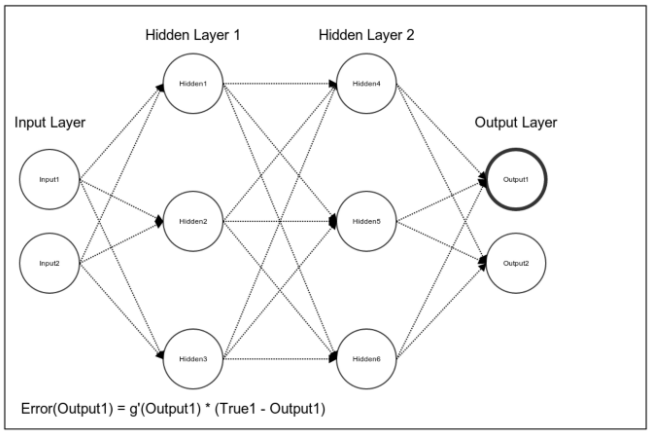
2.3.9 Backpropagation

The backpropagation algorithm is commonly used in conjunction with an optimization algorithm such as gradient descent to minimize the value of the cost function. The algorithm takes its name from a portmanteau of backward propagation, and refers to the direction in which errors flow through the layers of the network. Backpropagation can theoretically be used to train a feed-forward network with any number of hidden units arranged in any number of layers, though computational power constrains this capability.

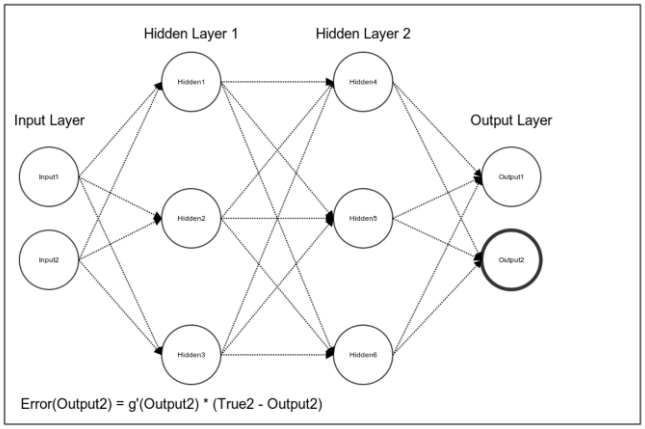
Backpropagation is similar to gradient descent in that it uses the gradient of the cost function to update the values of the model parameters. Unlike the linear models we have previously seen, neural nets contain hidden units that represent latent variables; we can't tell what the hidden units should do from the training data. If we do not know what the hidden units should do, we cannot calculate

their errors and we cannot calculate the gradient of cost function with respect to their weights. A naive solution to overcome this is to randomly perturb the weights for the hidden units. If a random change to one of the weights decreases the value of the cost function, we save the change and randomly change the value of another weight. An obvious problem with this solution is its prohibitive computational cost. Backpropagation provides a more efficient solution.

We can calculate the error of the network only at the output units. The hidden units represent latent variables; we cannot observe their true values in the training data and thus, we have nothing to compute their error against. In order to update their weights, we must propagate the network's errors backwards through its layers. We will begin with Output1. Its error is equal to the difference between the true and predicted outputs, multiplied by the partial derivative of the unit's activation:

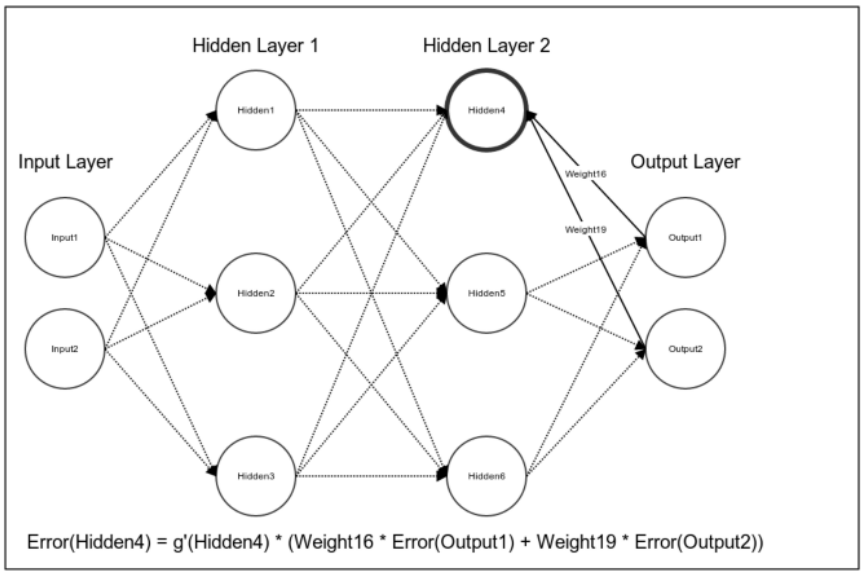


We then calculate the error of the second output unit:

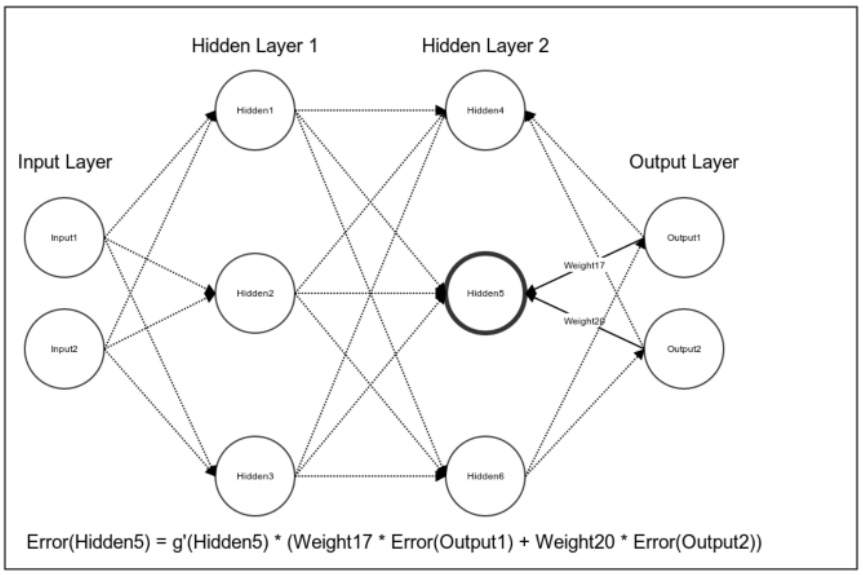


We computed the errors of the output layer. We can now propagate these errors backwards to the second hidden layer. First, we will compute the error of hidden unit Hidden4. We multiply the error of Output1 by the value of the weight

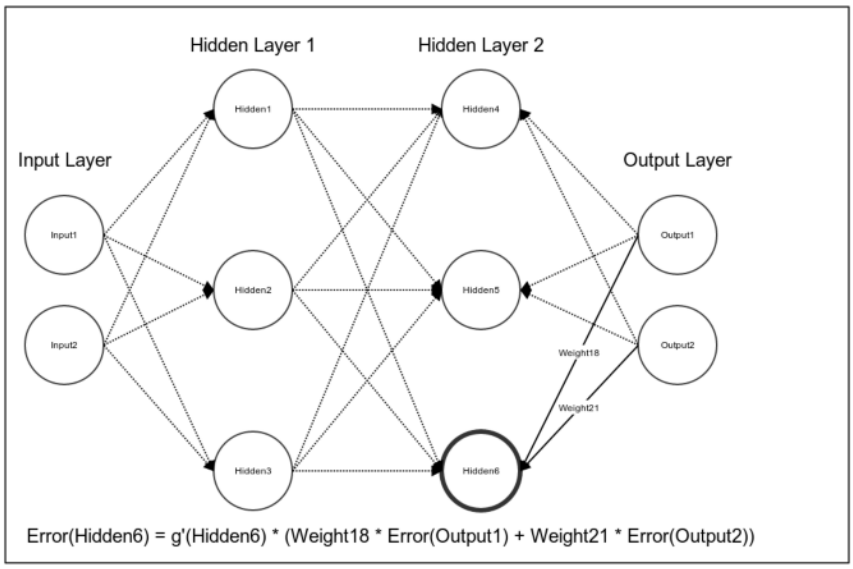
connecting Hidden4 and Output1. We similarly weigh the error of Output2. We then add these errors and calculate the product of their sum and the partial derivative of Hidden4:



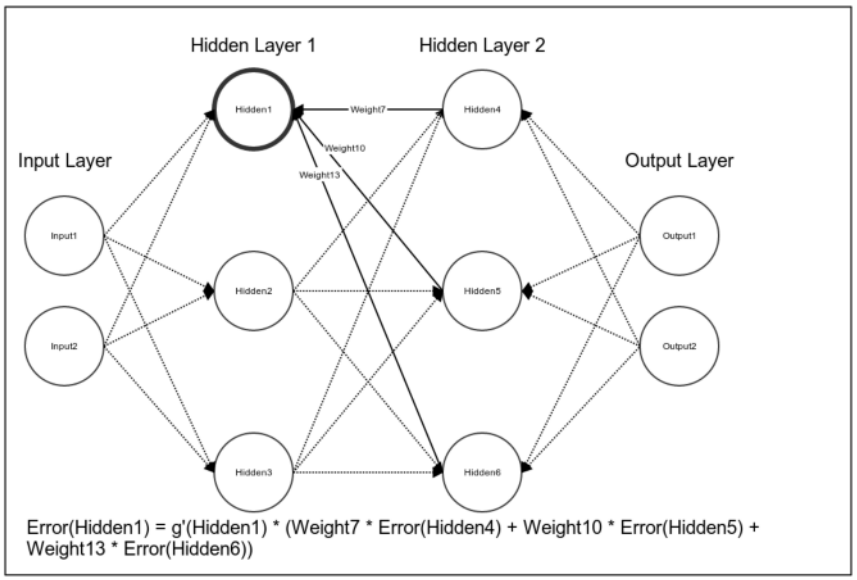
We similarly compute the errors of Hidden5:



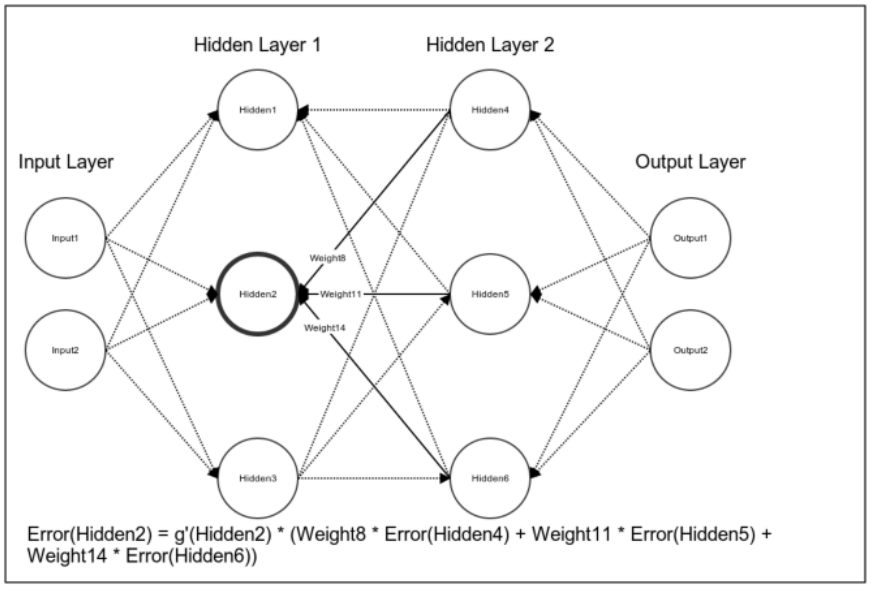
We then compute the Hidden6 error in the following figure:



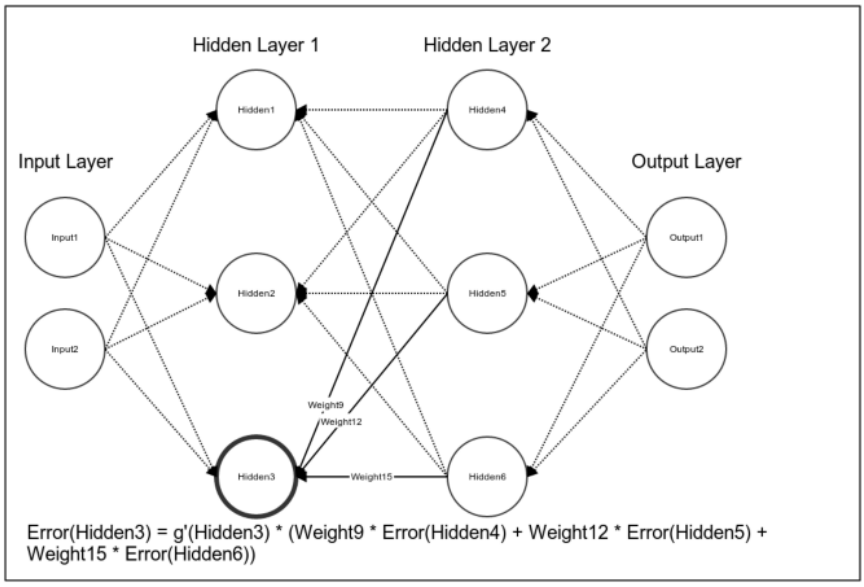
We calculated the error of the second hidden layer with respect to the output layer. Next, we will continue to propagate the errors backwards towards the input layer. The error of the hidden unit Hidden1 is the product of its partial derivative and the weighted sums of the errors in the second hidden layer:



We similarly compute the error for hidden unit Hidden2:

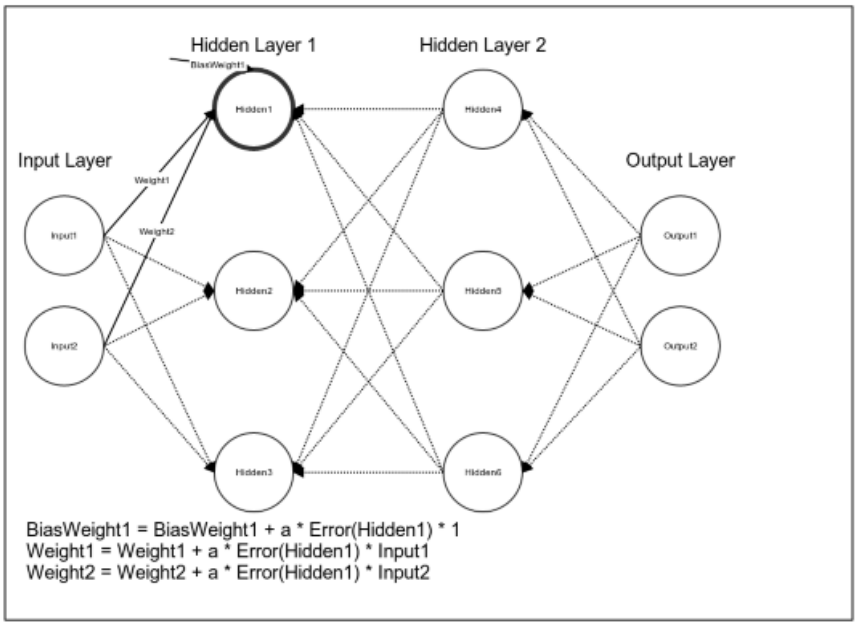


We similarly compute the error for Hidden3:

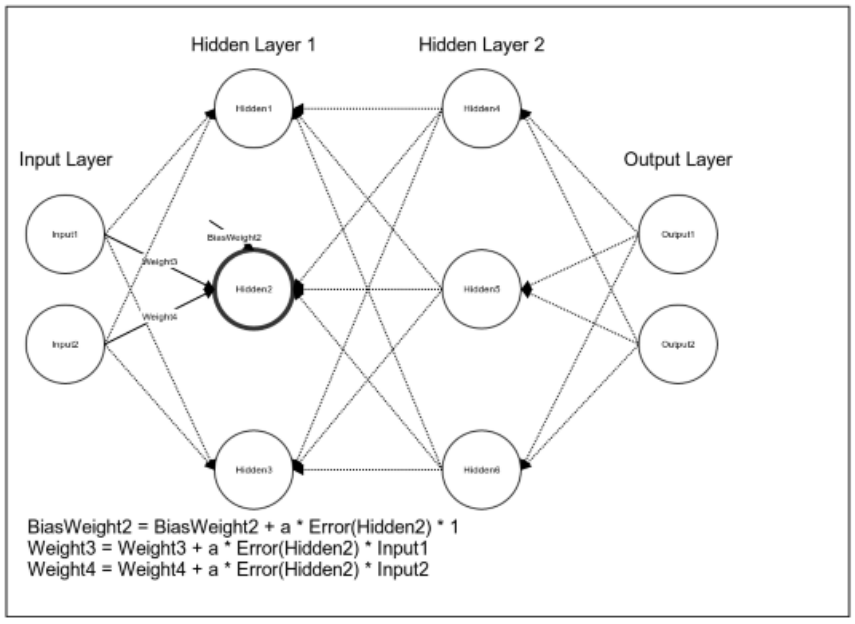


We computed the errors of the first hidden layer. We can now use these errors to update the values of the weights. We will first update the weights for the edges connecting the input units to Hidden1 as well as the weight for the edge connecting the bias unit to Hidden1. We will increment the value of the weight connecting Input1 and Hidden1 by the product of the learning rate, error of Hidden1, and the value of Input1.

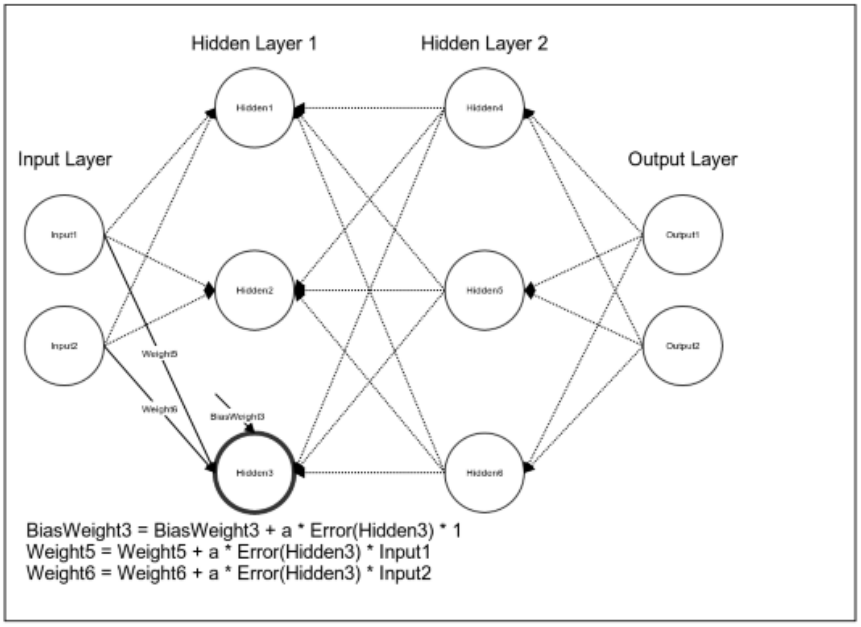
We will similarly increment the value of Weight2 by the product of the learning rate, error of Hidden1, and the value of Input2. Finally, we will increment the value of the weight connecting the bias unit to Hidden1 by the product of the learning rate, error of Hidden1, and one.



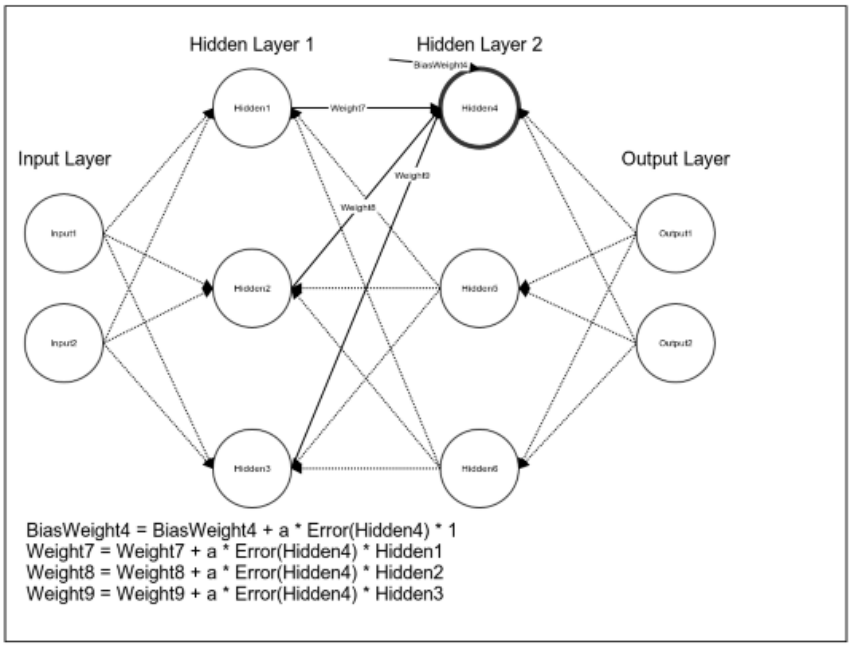
We will then update the values of the weights connecting hidden unit Hidden2 to the input units and the bias unit using the same method:



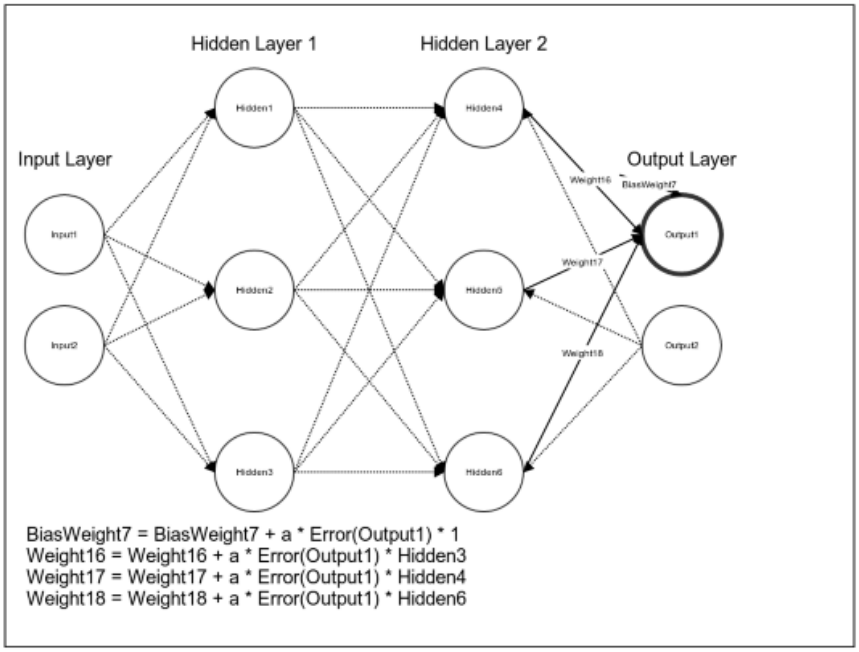
Next, we will update the values of the weights connecting the input layer to Hidden3:

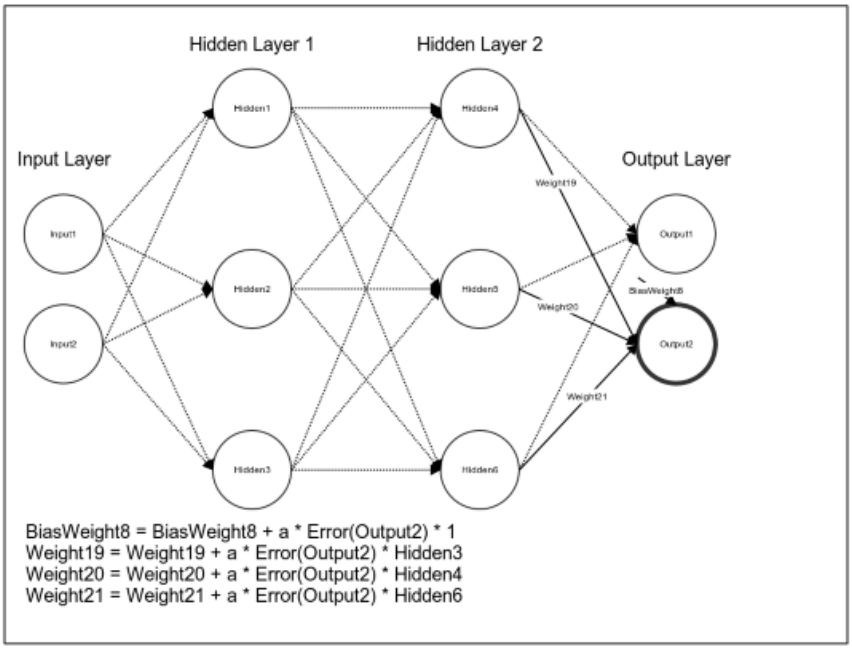


Since the values of the weights connecting the input layer to the first hidden layer is updated, we can continue to the weights connecting the first hidden layer to the second hidden layer. We will increment the value of Weight7 by the product of the learning rate, error of Hidden4, and the output of Hidden1. We continue to similarly update the values of weights Weight8 to Weight15:



The weights for Hidden5 and Hidden6 are updated in the same way. We updated the values of the weights connecting the two hidden layers. We can now update the values of the weights connecting the second hidden layer and the output layer. We increment the values of weights W16 through W21 using the same method that we used for the weights in the previous layers:





After incrementing the value of Weight21 by the product of the learning rate, error of Output2, and the activation of Hidden6, we have finished updating the values of the weights for the network. We can now perform another forward pass using the new values of the weights; the value of the cost function produced using the updated weights should be smaller. We will repeat this process until the model converges or another stopping criterion is satisfied. Unlike the linear models we have discussed, backpropagation does not optimize a convex function. It is possible that backpropagation will converge on parameter values that specify a local, rather than global, minimum. In practice, local optima are frequently adequate for many applications.

2.4 Kubeflow

2.4.1 Overview

Kubeflow is an open-source platform for developing, deploying, and running machine learning (ML) pipelines on top of Kubernetes. It aims to make it easy to build, deploy, and manage ML workflows on Kubernetes, by providing a set of tools, libraries, and APIs that can be used to build and deploy ML models.

Some of the key features of Kubeflow include:

- A user interface for managing and monitoring ML pipelines.
- A set of APIs and libraries for building and deploying ML models in a consistent, reproducible way.
- Support for popular ML frameworks such as TensorFlow, PyTorch, and XGBoost.
- Integration with other tools and services commonly used in ML workflows, such as Jupyter notebooks, Argo, and Seldon.

Kubeflow can be used to develop and deploy ML models in a variety of contexts, including on-premises data centers, cloud platforms, and edge devices. It is designed to be scalable, flexible, and easy to use, making it a popular choice for organizations looking to build and deploy ML applications at scale.

2.4.2 Kubeflow Pipelines

The part of Kubeflow that will concern us for our first experiments is *Kubeflow Pipelines*.

Kubeflow Pipelines is an open-source platform for building and deploying machine learning pipelines on Kubernetes. It allows users to build, deploy, and manage end-to-end machine learning workflows on cloud infrastructure, including on-premises and hybrid cloud environments.

Kubeflow Pipelines consists of several components, including the following:

- A user interface for designing and managing pipelines.
- A pipeline orchestration engine for executing pipelines.
- A set of pre-packaged, customizable pipeline components for common machine learning tasks such as data preprocessing, training, and evaluation.

- Integration with popular machine learning libraries and frameworks such as TensorFlow, PyTorch, and scikit-learn.

To deploy a Kubeflow pipeline, you will need to follow these steps:

- **Build your pipeline**: You can use the Kubeflow Pipelines user interface or the Kubeflow Pipelines Python SDK to build your pipeline. The SDK can connect directly to the Kubeflow API and deploy the pipeline or generate a YAML file that describes the pipeline. In the second case, we must upload the file to Kubeflow through the user interface. The pipeline should define the workflow of your machine learning tasks and the dependencies between them.
- **Deploy the pipeline**: You can use the kfp command-line tool or the Kubeflow Pipelines user interface to deploy your pipeline to the Kubeflow deployment.
- **Execute the pipeline**: You can use the Kubeflow Pipelines user interface or the kfp command-line tool to execute your pipeline and monitor its progress.

Kubeflow Pipelines also provides a number of features to help you manage and deploy your pipelines, such as pipeline versioning, resource management, and integration with popular machine learning frameworks.

2.4.3 Machine Learning workflow

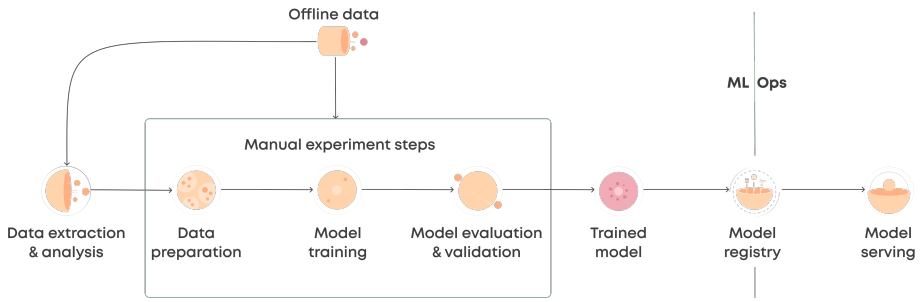


Figure 13: The structure of a Machine Learning Workflow

We have mentioned the term "Machine Learning workflow" many times until this point, but we have neglected to go into further detail. A Machine Learning workflow or pipeline is a set of steps that are followed to build and deploy a machine learning model. Here are some common steps that may be included in a machine learning pipeline:

1. **Data collection and preparation**: This step involves collecting and preparing the data that will be used to train and evaluate the machine learning model. This may involve tasks such as data cleaning, normalization, and feature selection.

2. **Data pre-processing**: This step involves transforming the raw data into a format that is suitable for model training. This may involve tasks such as feature scaling, one-hot encoding, and dimensionality reduction.

3. **Feature engineering**: This step involves designing and creating new features from the raw data that may be more useful for model training. This may involve tasks such as feature selection, feature transformation, and feature extraction.

4. **Model training**: This step involves using the pre-processed and engineered data to train a machine learning model. This may involve tasks such as selecting a model type, tuning hyperparameters, and evaluating model performance.

5. **Model evaluation**: This step involves evaluating the trained model on a holdout dataset to assess its performance. This may involve tasks such as calculating evaluation metrics, comparing the model to baseline benchmarks, and identifying any areas for improvement.

6. **Model deployment**: This step involves deploying the trained model to a production environment where it can be used to make predictions or take other actions. This may involve tasks such as setting up an API or integrating the model into an existing application.

These are just a few examples of steps that may be included in a machine learning pipeline. The specific steps and tasks will depend on the specific machine learning problem being solved and the tools and frameworks being used.

The purpose of a machine learning pipeline is to automate and streamline the process of building and deploying machine learning models. By using a pipeline, data scientists and machine learning engineers can focus on developing the individual components of the pipeline, rather than worrying about the overall workflow and orchestration of the process.

A machine learning pipeline can be implemented using a variety of tools and frameworks. For this thesis, we will use Kubeflow Pipelines [11] to build and

deploy a machine learning pipeline on Kubernetes. There are also other ways such as Apache Airflow.

2.4.4 Tensorflow Operator (TFJob)

TensorFlow [12] is one of the most commonly used machine learning frameworks, and Kubeflow provides a number of tools and resources for running TensorFlow workloads on Kubernetes. In Kubeflow, TensorFlow is executed as a Kubernetes operator. An operator is a Kubernetes custom resource that defines a set of APIs and controllers that automate the deployment and management of an application or service. The TensorFlow operator in Kubeflow is responsible for managing TensorFlow workloads in a Kubernetes cluster.

The TensorFlow operator in Kubeflow provides a number of features that make it easier to run TensorFlow workloads on Kubernetes. These features include:

- **Custom Resource Definitions (CRDs)**: The TensorFlow operator defines a set of custom resource definitions (CRDs) that allow users to create and manage TensorFlow jobs as Kubernetes resources. The CRDs define the specification for a TensorFlow job (**TFJob**), including the number of replicas, the image to use, and the command to run.

- **Automatic scaling**: The TensorFlow operator in Kubeflow can automatically scale the number of replicas of a TensorFlow job based on resource usage. This helps to ensure that the workload is evenly distributed across the cluster and that the resources are used efficiently.

- **Resource allocation**: The TensorFlow operator in Kubeflow can allocate resources to TensorFlow jobs based on the requirements specified in the job specification. This helps to ensure that the jobs have the resources they need to run efficiently.

- **Fault tolerance**: The TensorFlow operator in Kubeflow can automatically recover from failures by restarting failed replicas of a TensorFlow job. This helps to ensure that the workload is not interrupted and that the job completes successfully.

- **Monitoring and logging**: The TensorFlow operator in Kubeflow provides built-in monitoring and logging capabilities that allow users to track the progress of their TensorFlow jobs and debug any issues that arise.

Overall, the TensorFlow operator in Kubeflow provides a powerful set of tools and resources for running TensorFlow workloads on Kubernetes. With its automatic scaling, resource allocation, fault tolerance, and monitoring capabilities, the TensorFlow operator makes it easier for users to manage their TensorFlow workloads and achieve optimal performance.

2.4.5 TensorBoard

TensorBoard[13] is a visualization toolkit provided by TensorFlow, which is used to visualize the graph and other tools to understand, debug, and optimize the model. It provides various functionalities to plot/display various aspects of a machine learning pipeline. TensorBoard provides the visualization and tooling needed for machine learning experimentation such as tracking and visualizing metrics such as loss and accuracy, visualizing the model graph (ops and layers) and viewing histograms of weights, biases, or other tensors as they change over time.

2.5 Apache Hadoop

2.5.1 Overview

Apache Hadoop is an open-source software framework for distributed storage and distributed processing of large datasets on computer clusters. Hadoop was designed to scale up from single servers to thousands of machines, each offering local computation and storage.

Hadoop consists of two main components: the Hadoop Distributed File System (HDFS) and the MapReduce programming model. HDFS is a distributed file system that allows you to store large amounts of data across multiple servers. MapReduce is a programming model that allows you to process large datasets in parallel across a cluster of computers.

Hadoop is often used for big data workloads such as analyzing web logs, running machine learning algorithms, and performing data extract, transform, load (ETL) operations. It is a popular choice for data-intensive applications because it can handle large volumes of data efficiently and can be easily scaled out as the data size grows.

2.5.2 Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is the distributed storage com-

ponent of the Apache Hadoop platform. It is designed to store very large datasets (in the petabyte range) across a large number of commodity servers, providing very high aggregate bandwidth across the cluster.

The HDFS architecture consists of two types of nodes:

- **NameNode**: The NameNode is the master node that manages the file system namespace and controls access to files by clients. It maintains the file system tree and the metadata for all the files and directories in the tree. The NameNode does not store actual data blocks of the files; it stores only the metadata about the files, such as the file size, replication factor, and block location.
- **DataNode**: DataNodes are the slave nodes that store the actual data blocks of the files. Each DataNode stores a subset of the blocks in the file system, and it communicates with the NameNode to receive instructions and to report on the blocks that it stores.

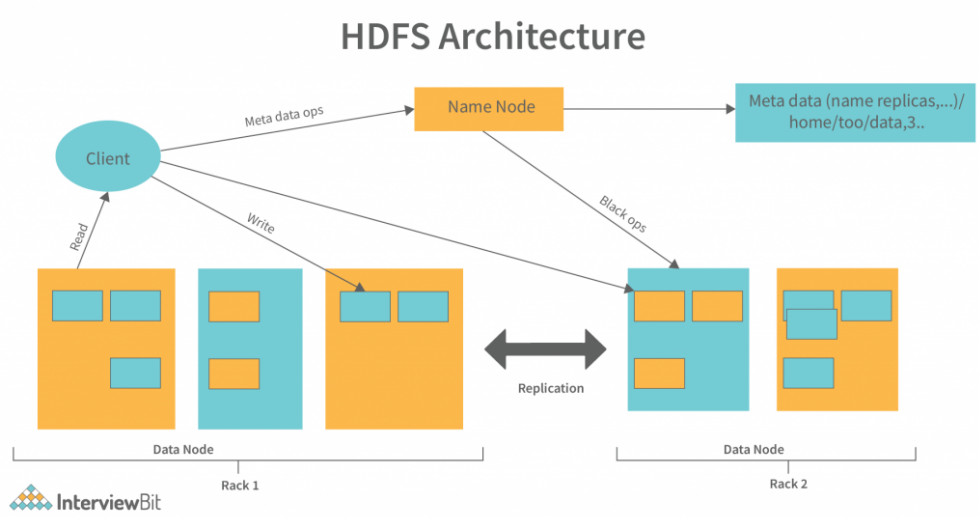


Figure 14: HDFS architecture

In a typical HDFS deployment, there is a single NameNode and a large number of DataNodes, often running on separate machines. The NameNode manages the file system namespace and the mapping of blocks to DataNodes, while the DataNodes store and serve the actual data blocks to clients.

When a client wants to read a file from HDFS, it sends a request to the NameNode, which responds with the locations of the blocks that make up the file. The client then retrieves the blocks directly from the appropriate DataNodes.

When a client wants to write a file to HDFS, it sends the data blocks to the DataNodes, which store them and report the successful storage to the NameNode.

2.6 Apache Spark

2.6.1 Overview

Apache Spark is an open-source, distributed computing system that provides a unified platform for data processing, machine learning, and analytics. It was designed to be fast and general-purpose, with the ability to handle a wide variety of data processing tasks.

Spark is built on top of the Hadoop ecosystem and can run on a variety of cluster managers, such as Hadoop YARN and Apache Mesos, as well as on cloud platforms such as Amazon EC2. It is implemented in Scala, but provides APIs in other languages such as Java, Python, and R, making it easy to use for a wide range of users.

One of the key features of Spark is its in-memory data processing engine, which allows it to perform faster than Hadoop MapReduce for many types of data processing tasks. Spark also includes a number of libraries for tasks such as stream processing, machine learning, and graph processing, making it a versatile platform for a wide range of data processing and analytics workloads.

In addition to its core data processing engine, Spark also provides a number of tools and libraries for interacting with data stored in external storage systems such as HDFS, Apache Cassandra, and Amazon S3. It also includes a number of libraries for working with structured data, such as Spark SQL, which allows you to use SQL to query data stored in Spark.

2.6.2 Architecture

The Apache Spark architecture consists of the following main components:

- **Spark Driver**: The Spark Driver is the central coordinator of a Spark application. It is responsible for maintaining the SparkContext and coordinating the execution of tasks on the worker nodes. The driver also communicates with the cluster manager to request resources for the application and to launch executors on the worker nodes.
- **Spark Executor**: Spark Executors are worker processes that run on the cluster nodes and execute tasks assigned to them by the Spark Driver.

Each executor is responsible for running one or more tasks in parallel, and it communicates with the driver to report the status of the tasks and to request additional tasks as needed.

- **Spark Worker**: A Spark Worker is a process that runs on a cluster node and manages the execution of tasks on that node. It launches the executors that run on the node and communicates with the driver to request resources and receive instructions.
- **Cluster Manager**: A Cluster Manager is a system that manages the resources of a cluster of machines and allocates them to Spark applications as needed. Spark can run on a variety of cluster managers, including Hadoop YARN, Apache Mesos, and Kubernetes.
- **Spark Storage**: Spark uses a variety of storage systems to store data, including HDFS, Apache Cassandra, and Amazon S3. Data can be stored in a distributed fashion across the worker nodes in the cluster, allowing it to be processed in parallel by the executors.
- **Spark SQL**: Spark SQL is a module of Spark that provides support for structured data processing, including the ability to query data using SQL. It includes a Dataframe API for working with structured data and a SQL API for querying data stored in Spark using SQL.

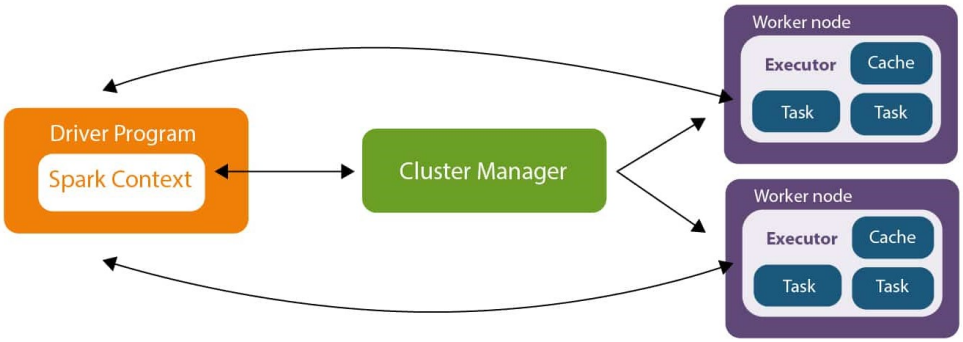


Figure 15: Spark architecture

These are the main components of the Apache Spark architecture. Spark applications are composed of a driver program and a set of executors that run on the cluster, and they are managed by a cluster manager and store data in a distributed storage system.

2.6.3 Spark-on-k8s-operator

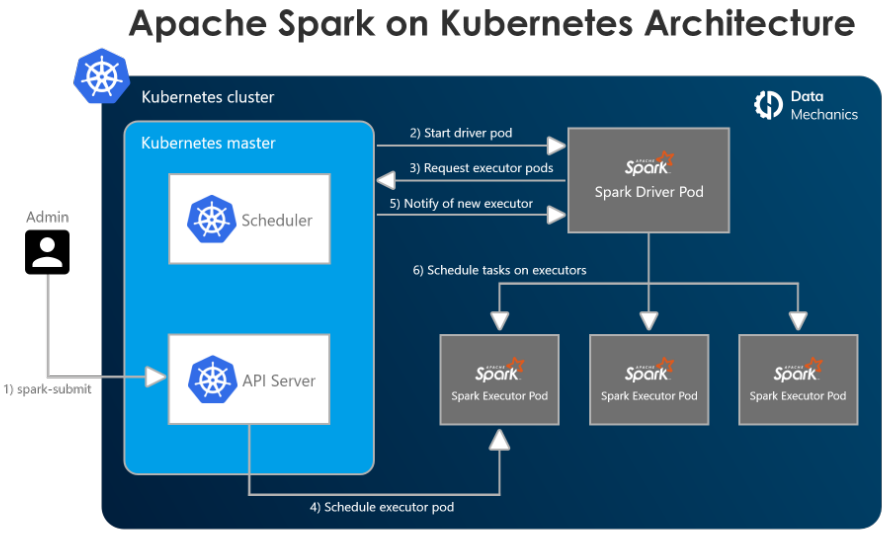


Figure 16: Spark on Kubernetes

In the current thesis, we will utilize Apache Spark on Kubernetes through the Spark-on-k8s-operator. Spark-on-k8s-operator is a Kubernetes operator for running Apache Spark applications on Kubernetes. It is designed to make it easier to deploy and manage Spark applications on Kubernetes by providing a declarative API for defining Spark applications and a controller that manages the lifecycle of Spark applications.

With spark-on-k8s-operator, you can define a Spark application as a Kubernetes resource, and the operator will take care of deploying and managing the Spark drivers and executors for you. You can use the operator to submit Spark jobs, scale the number of executors up or down, and monitor the status of Spark applications.

Spark-on-k8s-operator is built on top of the Kubernetes API and integrates with the Kubernetes ecosystem, making it easy to use with other Kubernetes-based tools and services. It is an open-source project that is developed and maintained by the Apache Spark community.

3 System Implementation

3.1 System components

The system that we will experiment on is an Open Source Cloud Computing Infrastructure (OpenStack [14]), which is comprised of 4 Virtual Machines (nodes). The system is provided by the Computing Systems Laboratory of the Electrical and Computer Engineering department and each Virtual Machine has the following specifications:

- **Operating System**: Ubuntu 20.04 64-bit
- **CPU**: 4 Cores
- **RAM**: 8GB
- **Hard Drive**: 64 GB

Our local workstation is a Windows 11 64-bit PC, with Windows Subsystem for Linux (WSL) with Ubuntu 22.04 64-bit installed. We connect to the remote system through SSH.

Our system will be comprised of the following components (Figure 17):

- **Kubernetes**: The Kubernetes cluster is the main part of our system and we need it to execute Kubeflow. There will be one control node (Master) and the other three will be worker nodes. The version that we will use is **Kubernetes 1.22.5**. We chose this version, so that it would be compatible with the latest version of Kubeflow.
- **Prometheus and Grafana**: We will install these two tools on Kubernetes to monitor the performance of our cluster. The version of Grafana we installed is **9.3.1** and the version of Prometheus is **2.37.2**.
- **Kubeflow**: Kubeflow will be installed on the Kubernetes cluster and will be used to execute our Machine Learning algorithms. The **TensorFlow operator** and **TensorBoard** are included in the full version of Kubeflow. The version that we will use is **Kubeflow 1.6**, the latest at this time of writing.
- **HDFS**: HDFS is independent of the Kubernetes cluster and is composed of one Name Node and 3 Data Nodes. The Name Node of the HDFS is one of the worker nodes of the cluster, so as to avoid overloading the Kubernetes control node. It is used by Kubeflow to load and store data. The version that we will use is **Apache Hadoop 3.3.1**.

- **Spark-on-k8s-operator**: We will use it to deploy Apache Spark Jobs on Kubernetes. The version that we will use is **Spark-on-k8s-operator 1.1.26**.

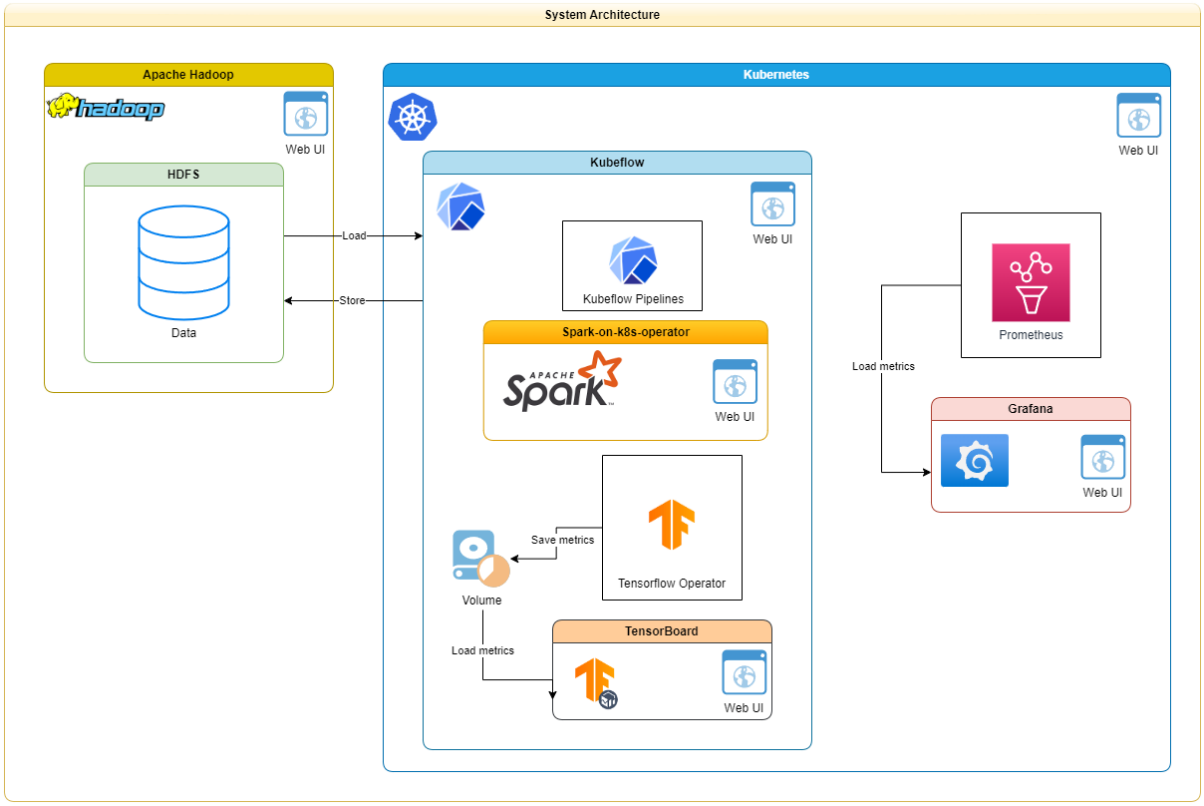


Figure 17: System Architecture

3.2 Installation

3.2.1 Kubernetes cluster

For our system, we use **Kubespray 2.18.1**, which installs Kubernetes 1.22.5 to our cluster of Virtual Machines. Here is a high-level overview of the steps involved in installing Kubespray:

1. **Prerequisites**: Before we could use Kubespray, we needed to install and configure the following components to install Python 3.8 to our workstation (WSL). We also updated the nodes using the *sudo apt update* and *sudo apt upgrade* commands.

2. **Clone the Kubespray repository**: Using the *git clone* command, we downloaded the Kubespray files from its repository on GitHub [15] to our workstation.

3. **Install required Python modules**: The Kubespray files contain a *requirements.txt* file, that includes the Python modules needed to run Kubespray. The main module included is Ansible, which we mentioned earlier.

4. **Customize the configuration**: Kubespray uses a configuration file (inventory/mycluster/inventory.ini) to specify the servers that you want to use for your cluster. We edited this file to include the IP addresses of the cluster nodes and assigned them their roles (Master and workers).

5. **Enable addons**: We edited certain lines in the addons file (inventory/mycluster/group_vars/k8s_cluster/addons.yml), in order to enable the Kubernetes Dashboard (a Graphical User Inteface that allows us to overview and manage our cluster) and the Local Path Storage Provisioner (A Local Persistent Volume).

6. **Run the installation playbook**: Once you have edited the configuration file, you can use Ansible to run the installation playbook and install Kubernetes on your servers. To do this, we executed the *ansible-playbook -i inventory/mycluster/hosts.yml –become –become-user=root cluster.yml* command from the root of the Kubespray repository. The playbook is comprised of centain tasks:

    - Installing and configuring the Kubernetes control plane components (e.g. kube-apiserver, kube-controller-manager, kube-scheduler)
    - Installing and configuring the necessary packages and dependencies on all of the hosts in the cluster
    - Installs the container engine on all nodes. The default container engine is *containerd*.
    - Installs and configures etcd, the distributed key-value store used by Kubernetes
    - Installing and configuring the Kubernetes node components (e.g. kubelet, kube-proxy)
    - Configuring network settings for the cluster (e.g. using flannel, calico, or Weave Net)
    - Generating SSL certificates for the cluster
    - Configuring firewall rules
    - Deploying additional Kubernetes components (e.g. the Kubernetes dashboard)

7. **Verify the installation**: Once the installation was complete, you had to verify that everything is working by checking the status of the nodes and pods in your cluster. In order to do so, we installed kubectl to our workstation, the Kubernetes Command Line management tool. We connected kubectl to the cluster, by copying the *kubeconfig* file from the */root/.kube* of the master node to the *$HOME/.kube* of the local workstation. The kubeconfig file contains the IP of the cluster as well as the admin certificate required to access the Kubernetes resources. Then we used the command *kubectl cluster-info* to make sure the Kubernetes cluster is running and the *kubectl get nodes* command, to make sure all nodes are functional and connected to the cluster.

```
NAME      STATUS   ROLES                  AGE   VERSION
master0   Ready    control-plane,master   19d   v1.22.5
worker1   Ready    worker                 19d   v1.22.5
worker2   Ready    worker                 19d   v1.22.5
worker3   Ready    worker                 19d   v1.22.5
```

Figure 18: Kubernetes Cluster nodes

```
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-8575b76f66-pkk6g   1/1     Running   0          8m57s
kube-system   calico-node-2j8n9                          1/1     Running   0          10m
kube-system   calico-node-75nt7                          1/1     Running   0          10m
kube-system   calico-node-cqgww                          1/1     Running   0          10m
kube-system   calico-node-qnwn7                          1/1     Running   0          10m
kube-system   coredns-8474476ff8-87xgq                   1/1     Running   0          8m1s
kube-system   coredns-8474476ff8-c2b9h                   1/1     Running   0          6m27s
kube-system   dns-autoscaler-7df78bfcfb-glbbq            1/1     Running   0          6m32s
kube-system   kube-apiserver-master0                     1/1     Running   0          14m
kube-system   kube-controller-manager-master0            1/1     Running   4          14m
kube-system   kube-proxy-6rgr8                           1/1     Running   0          12m
kube-system   kube-proxy-fgxjw                           1/1     Running   0          12m
kube-system   kube-proxy-n2sfw                           1/1     Running   0          12m
kube-system   kube-proxy-plggl                           1/1     Running   0          12m
kube-system   kube-scheduler-master0                     1/1     Running   4          14m
kube-system   nginx-proxy-worker1                        1/1     Running   0          12m
kube-system   nginx-proxy-worker2                        1/1     Running   0          12m
kube-system   nginx-proxy-worker3                        1/1     Running   0          12m
kube-system   nodelocaldns-2mr4v                         1/1     Running   0          6m31s
kube-system   nodelocaldns-cvgkx                         1/1     Running   0          6m31s
kube-system   nodelocaldns-fs69x                         1/1     Running   0          6m31s
kube-system   nodelocaldns-xshhh                         1/1     Running   0          6m32s
```

Figure 19: Kubernetes Running Pods

3.2.2 Kubeflow

There are many methods to install Kubeflow and after much experimentation, we choose to use the Charmed Kubeflow [16] deployment that is offered by Canonical. The steps required are the following:

1. **Install the Juju client**: To install Charmed Kubeflow, we needed to have a Juju[17] client installed on your local workstation. Juju is easily installed using the command *sudo snap install juju –classic*.

2. **Connect Juju to the Kubernetes cluster**: Juju is connected to our Kubernetes cluster, using the kubeconfig file in the $HOME/.kube directory and by using the command *juju add-k8s myk8s*. One may also specify a specific type of storage for Juju to use, but we used the default Local Path storage, we created earlier. Juju recognizes our cluster as an object known as a Cloud.

3. **Create a controller**: To operate workloads on our Kubernetes cluster, Juju uses controllers. In the Juju ecosystem, a controller is a central management entity that is responsible for coordinating the deployment and operation of applications in a cloud environment. The controller is responsible for creating and managing the model(s) that represent the deployable units in Juju, and it communicates with the cloud environment to deploy and manage the applications and their associated resources, such as storage, networking, and compute resources. Each Juju model is associated with a single controller, and the controller is responsible for managing the lifecycle of the model and its associated applications. We created a controller and connected it to the Cloud using the command *juju bootstrap myk8s my-controller*.

4. **Create a Juju model**: A Juju model represents a deployable unit in Juju. You will need to create a new model in which to deploy Charmed Kubeflow. We did so with the command *juju add-model kubeflow*. It is required by Charmed Kubeflow, that its model has the name *kubeflow*, because the same name will be used for the namespace that the Kubeflow Pods will be assigned to. By default, a namespace called *admin* is also created, where our pipelines (their Pods) will be assigned to.

5. **Deploy Charmed Kubeflow**: To deploy Charmed Kubeflow, we need to use the Juju client to add a charm to the *kubeflow* model. A charm is a package that contains the configuration and instructions needed to deploy and manage a specific application or service. To add the Charmed Kubeflow charm to our model, we used the *juju deploy kubeflow –trust* command.

6. **Check if Kubeflow is operational**: We execute the *watch -c juju status –color* command and we wait until all Kubeflow services are in the *active* state. This takes about 10-20 minutes.

7. **Access the Kubeflow dashboard**: Once Charmed Kubeflow has been deployed, we needed to access the Kubeflow dashboard to begin using Kubeflow and manage our machine learning workloads. To access the Kubeflow

dashboard, we needed to find the URL for the dashboard and use it to connect to the dashboard in our web browser. To that end, we used the command *kubectl -n kubeflow get svc istio-ingressgateway-workload -o jsonpath='.status.loadBalancer.ingress[0].ip'*. Once we had the URL, we used the following commands to enable access to the dashboard and set the credentials:

```
juju config dex-auth public-url=[URL]
juju config oidc-gatekeeper public-url=[URL]
juju config dex-auth static-username=[USERNAME]
juju config dex-auth static-password=[PASSWORD]
```

Finally, we had to set a *localhost:9999 SOCKS proxy* to our browser and connect to the node where Kubeflow dashboard was installed using the command *ssh -D 9999 USER_NAME@<machine_ip>*. Then we simply entered the URL in the browser and logged with the credentials we set earlier.

```
Model     Controller     Cloud/Region  Version  SLA          Timestamp
kubeflow  my-controller  myk8s         2.9.37   unsupported  19:17:24+02:00

App                       Version                  Status   Scale  Charm                     Channel         Rev  Address         Exposed
admission-webhook         res:oci-image@129fe92    active       1  admission-webhook         1.6/stable       60  10.233.4.100    no
argo-controller           res:oci-image@669ebd5    active       1  argo-controller           3.3/stable       99                  no
argo-server               res:oci-image@576d038    active       1  argo-server               3.3/stable       45                  no
dex-auth                                           active       1  dex-auth                  2.31/stable     129  10.233.50.176   no
istio-ingressgateway                               active       1  istio-gateway             1.11/stable     114  10.233.36.24    no
istio-pilot                                        active       1  istio-pilot               1.11/stable     131  10.233.23.28    no
jupyter-controller        res:oci-image@e05857e    active       1  jupyter-controller        1.6/stable      163                  no
jupyter-ui                res:oci-image@d55c600    active       1  jupyter-ui                1.6/stable      124  10.233.58.184   no
katib-controller          res:oci-image@03d47fb    active       1  katib-controller          0.14/stable      92  10.233.35.180   no
katib-db                  mariadb/server:10.3      active       1  charmed-osm-mariadb-k8s   latest/stable    35  10.233.36.158   no
katib-db-manager          res:oci-image@16b33a5    active       1  katib-db-manager          0.14/stable      66  10.233.49.92    no
katib-ui                  res:oci-image@c7dc04a    active       1  katib-ui                  0.14/stable      90  10.233.1.218    no
kfp-api                   res:oci-image@bf7d7d5    active       1  kfp-api                   2.0/stable      144  10.233.11.210   no
kfp-db                    mariadb/server:10.3      active       1  charmed-osm-mariadb-k8s   latest/stable    35  10.233.36.153   no
kfp-persistence           res:oci-image@abcf971    active       1  kfp-persistence           2.0/stable      141                  no
kfp-profile-controller    res:oci-image@b4de878    active       1  kfp-profile-controller    2.0/stable      125  10.233.36.194   no
kfp-schedwf               res:oci-image@9c9f710    active       1  kfp-schedwf               2.0/stable      155                  no
kfp-ui                    res:oci-image@47864af    active       1  kfp-ui                    2.0/stable      144  10.233.60.251   no
kfp-viewer                res:oci-image@94754c0    active       1  kfp-viewer                2.0/stable      152                  no
kfp-viz                   res:oci-image@23ab9b9    active       1  kfp-viz                   2.0/stable      134  10.233.14.185   no
kubeflow-dashboard        res:oci-image@6fe6eec    active       1  kubeflow-dashboard        1.6/stable      183  10.233.9.117    no
kubeflow-profiles         res:profile-image@cfd6935  active       1  kubeflow-profiles         1.6/stable       94  10.233.4.181    no
kubeflow-roles                                     active       1  kubeflow-roles            1.6/stable       49  10.233.28.58    no
kubeflow-volumes          res:oci-image@fdb4a5d    active       1  kubeflow-volumes          1.6/stable       84  10.233.8.60     no
metacontroller-operator                            active       1  metacontroller-operator   2.0/stable       48  10.233.33.19    no
minio                     res:oci-image@1755999    active       1  minio                     ckf-1.6/stable   99  10.233.32.211   no
oidc-gatekeeper           res:oci-image@32de216    active       1  oidc-gatekeeper           ckf-1.6/stable  176  10.233.40.22    no
seldon-controller-manager  res:oci-image@eb811b6   active       1  seldon-core               1.14/stable      92  10.233.13.36    no
tensorboard-controller    res:oci-image@51058f7    active       1  tensorboard-controller    1.6/stable       69  10.233.48.108   no
tensorboards-web-app      res:oci-image@eef68a5    active       1  tensorboards-web-app      1.6/stable       71  10.233.7.209    no
training-operator                                  active       1  training-operator         1.5/stable       65  10.233.32.17    no
```

Figure 20: Kubeflow Services

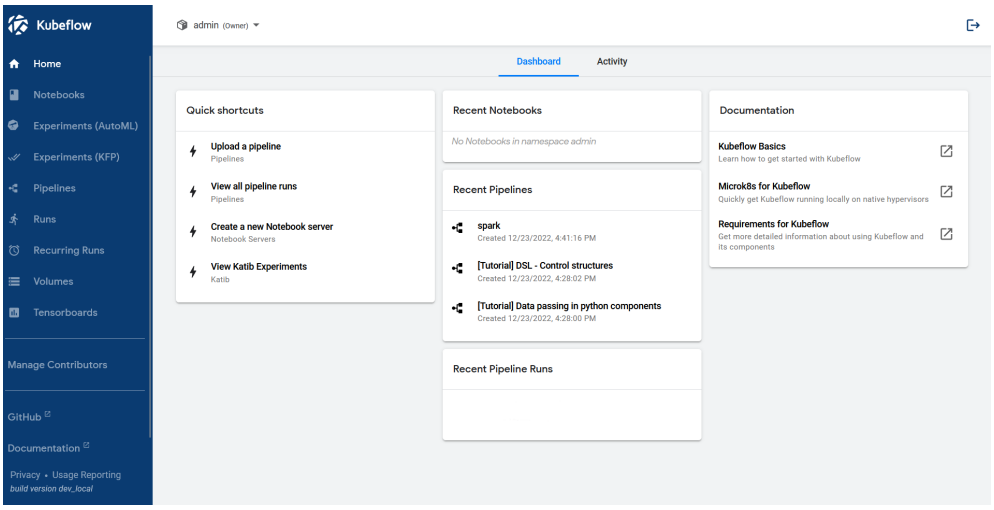


Figure 21: The Kubeflow Dashboard

3.2.3 Hadoop Distributed File System

Next, we needed to install Apache Hadoop to our cluster, to be able to use HDFS to store and load our data. The steps we followed are:

1. **Install Java**: Hadoop requires a Java runtime environment (JRE) to be installed on our cluster. Therefore, we had to install the Java JRE in all the nodes.

2. **Download Hadoop**: We downloaded Hadoop 3.3.1 from the official Hadoop website on the local workstation. Then we extracted it to a directory on our machine.

3. **Configure Hadoop**: Hadoop uses a number of configuration files to control its behavior. We edited these files to set up Hadoop for our environment. The most important configuration files are:

    - **etc/hadoop/hadoop-env.sh**: This file sets up the environment variables required by Hadoop. We had to set the JAVA_HOME variable to the location of your Java installation.
    - **etc/hadoop/core-site.xml**: This file contains configuration options for Hadoop's core services. We had to set the fs.defaultFS property to the URI of your default file system (e.g., hdfs://namenode:9000).
    - **etc/hadoop/hdfs-site.xml**: This file contains configuration options for the Hadoop Distributed File System (HDFS). We had to set the dfs.replication property to the desired replication factor for HDFS blocks.

We also created a *masters* and a *workers* file, inside the Hadoop directory, which contain the IP addresses of the name node and the data nodes respectively. Once we completed the configurations, we copied the Hadoop directory from the workstation to the *$HOME/hadoop* directory of all nodes.

4. **Start the Hadoop services**: Hadoop consists of several daemons that run on your machine and provide the various Hadoop services. We connected to the name node, formatted the HDFS cluster with the *hdfs namenode -format* command and we used the start-dfs.sh script to start all of the processes required for the HDFS.

5. **Test the Hadoop installation**: We tested our installation following these steps:

   (a) By running the *jps* command on the name node and the data nodes to ensure the required processes were running.

   (b) By accessing the Hadoop dashboard (http://NAMENODE_IP:9870) and checking that all nodes are in the *ready* state

   (c) By creating a directory in HDFS (*hdfs dfs -mkdir FOLDER*) and uploading a file in it (*hdfs dfs -put FILE FOLDER/*). We confirmed it was uploaded with the *hdfs dfs -ls FOLDER* command.

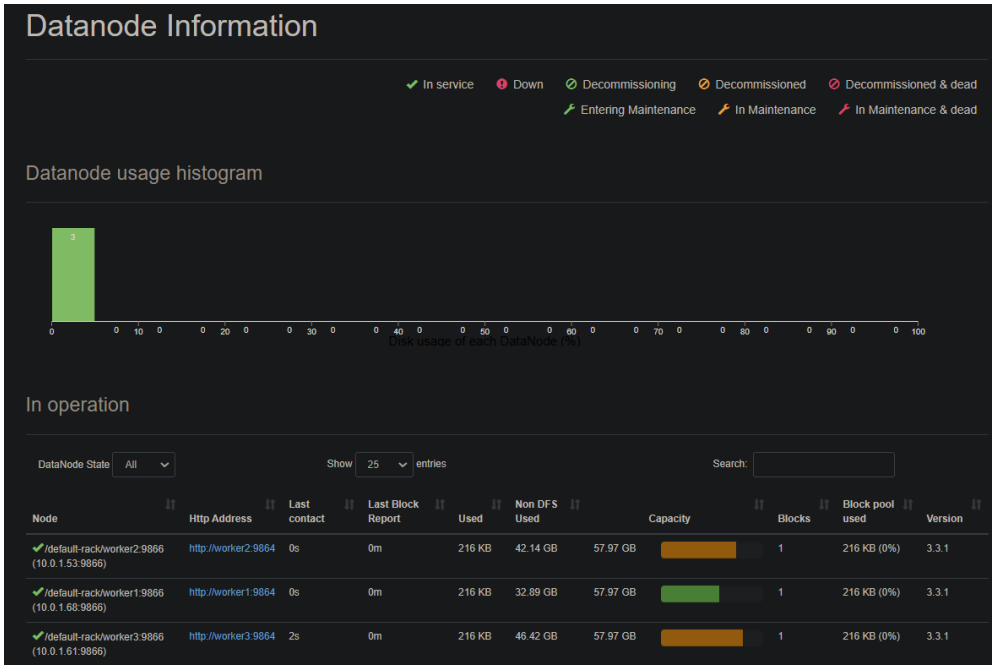


Figure 22: The Hadoop Dashboard showing the data nodes

3.2.4 Spark-on-k8s-operator

Finally we installed the Spark-on-k8s-operator on our cluster, using a Helm Chart [18]. A Helm chart is a package of pre-configured Kubernetes resources that can be deployed to a Kubernetes cluster. Helm charts are used to deploy applications, databases, and other services to a Kubernetes cluster. They provide a standard, reusable, and easy-to-use way to deploy applications and services to Kubernetes. A Helm chart consists of a collection of YAML files that define the Kubernetes resources that make up the application or service. These YAML files include resource definitions for pods, services, deployments, and other resources. The Helm chart also includes a values.yaml file that allows you to customize the configuration of the resources defined in the chart.

We installed Helm and then using it, we installed spark-on-k8s-operator with the *helm install spark spark-operator/spark-operator –namespace spark-operator –create-namespace* command. Finally, we tested that it was correctly installed by deploying the Spark Pi example to our cluster, using the YAML file from the spark-on-k8s-operator Github repository (*kubectl apply -f examples/spark-pi.yaml*). The Pi example calculates pi by "throwing darts" at a circle — it generates points in the unit square ((0,0) to (1,1)) and counts how many points fall within the unit circle within the square. The result approximates pi. The example was completed successfully.

4 Experiments

4.1 Machine Learning workflows with Kubeflow

4.1.1 MNIST Classification Pipeline

Using a Machine Learning pipeline on Kubeflow, we attempted to perform classification on the MNIST dataset [19]. MNIST is a large database of small, square 28x28 pixel grayscale images of handwritten single digits between 0 and 9. It consists of a total of 70,000 handwritten images of digits, with the training set having 60,000 images and the test set having 10,000. All images are labeled with the respective digit that they represent. There are a total of 10 classes of digits (from 0 to 9).

The pipeline receives the following inputs (hyperparameters) and values:

- **Learning Rate**: 0.001, 0.01
- **Epochs**: 10, 25, 50
- **Batch Size**: 32, 64
- **Optimizer**: Adam, SGD

Our pipeline is comprised of the following steps:

1. **Load Data**: This step involves loading the MNIST dataset and then the dataset is split into training and testing sets and then further split into data and labels.
2. **Preprocess Data**: In this step, we normalize the images by dividing the pixel values with 255, the maximum value an RGB image pixel can receive. By doing so, the image values range from 0 to 1.
3. **Define Model**: Here we construct the model that we will use to perform the classification on our data. Based on the existing bibliography, we have chosen to utilize a Convolutional Neural Network with the following structure:
   (a) A *2D Convolutional* layer with a kernel of size 5x5, 24 filters and ReLU as the activation function. A convolutional layer applies a set of filters to the input data, where each filter is designed to detect a specific feature or pattern in the input. The output of a convolutional layer is a set of feature maps, which are used as input to the next layer in the network

(b) A *2D Max Pooling* layer with a kernel of size 2x2. Max pooling is a technique used in convolutional neural networks to down-sample the spatial dimensions of the feature maps. It works by applying a max filter to non-overlapping subregions of the feature map, resulting in a new feature map with smaller spatial dimensions.

(c) A *2D Convolutional* layer with a kernel of size 5x5, 48 filters and ReLU as the activation function.

(d) A *2D Max Pooling* layer with a kernel of size 2x2.

(e) A *2D Convolutional* layer with a kernel of size 5x5, 64 filters and ReLU as the activation function.

(f) A *2D Max Pooling* layer with a kernel of size 2x2.

(g) A *Flatten* layer. As its name suggests, this layer flattens the input to a vector output.

(h) A *Dense* layer with an 1x256 vector as the output. A dense layer in a neural network is a type of fully connected layer, where each neuron in the layer is connected to every neuron in the previous layer. This means that the output of each neuron in the layer is the weighted sum of the inputs, passed through an activation function. In our case, the layer is comprised of 256 neurons. ReLU is used as the activation function.

(i) A *Dense* layer with an 1x10 vector as the output. This layer is the output layer of the network, therefore we need it to have the same number of outputs as the number of classes. Softmax is the activation function, so as to produce a probability distribution over the 10 classes.

4. **Train**: The most important step of our pipeline is to train the model we defined earlier. We train the model using the Sparse Categorical Cross entropy loss function. The class with the highest probability output is considered the correct one for the corresponding sample.

5. **Evaluate**: Having trained the model, we test its performance on samples unknown to it, the test set. We judge its performance based on the accuracy metric. Accuracy is defined as

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

We evaluated our pipeline for various values of the hyperparamenters. The results with the highest accuracy are:

Epochs	Learning Rate	Optimizer	Batch size	Accuracy
10	0.001	Adam	32	99.94%
25	0.001	Adam	32	99.47%
50	0.001	Adam	64	99.46%
10	0.01	SGD	32	99.47%

Based on these results, we make the following observations:

- We observe that 10 epochs seem to suffice for the training of this model. Therefore, we will consider it the optimal value, as a larger number of epochs would likely cause overfitting and decrease the overall performance. Overfitting is a common problem in machine learning, particularly in deep learning, where a model is trained to fit the training data too closely and as a result, generalizes poorly to new, unseen data. It occurs when a model is trained with too many parameters relative to the amount of data, and it starts to learn the noise in the training data instead of the underlying pattern. This can lead to a model with a high training accuracy but poor performance on unseen data.

- The Adam optimizer seems to produce the highest accuracy in most cases.

- Learning rate = 0.001 seems to be the optimal value for this model.

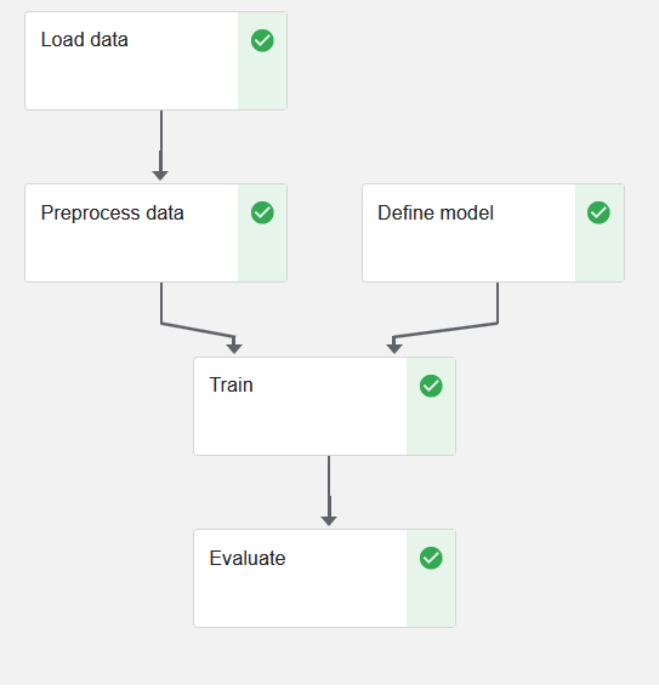


Figure 23: MNIST Pipeline (Successful Execution)

4.1.2 CIFAR-10 Classification Pipeline

We attempted to do the same on the CIFAR-10 dataset [20]. The CIFAR-10 dataset is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

The pipeline receives the following inputs (hyperparameters) and values:

- **Learning Rate**: 0.001, 0.01
- **Epochs**: 25, 50, 100, 200
- **Batch Size**: 32, 64
- **Optimizer**: Adam, SGD

Our pipeline is comprised of the following steps:

1. **Load Data**: This step involves loading the CIFAR-10 dataset using the Keras Dataset module. Then the dataset is split into training and testing sets and then further split into data and labels.
2. **Preprocess Data**: In this step, we normalize the images like we did with the MNIST dataset. However, since our labels are strings here and not integers like in MNIST, we one-hot encode our labels to convert them to numeric values.
3. **Define Model**: Here we construct the model that we will use to perform the classification on our data. Based on the existing bibliography, we have chosen to utilize a Convolutional Neural Network with the following structure:
   (a) A *2D Convolutional* layer with a kernel of size 3x3, 32 filters and a ReLU activation function.
   (b) *A Batch Normalization layer.* Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.
   (c) A *2D Convolutional* layer with a kernel of size 3x3, 32 filters and a ReLU activation function.

(d) *A Batch Normalization layer.*

(e) A *2D Max Pooling* layer with a kernel of size 2x2.

(f) A *2D Convolutional* layer with a kernel of size 3x3, 64 filters and a ReLU activation function.

(g) *A Batch Normalization layer.*

(h) A *2D Convolutional* layer with a kernel of size 3x3, 64 filters and a ReLU activation function.

(i) *A Batch Normalization layer.*

(j) A *2D Max Pooling* layer with a kernel of size 2x2.

(k) A *2D Convolutional* layer with a kernel of size 3x3, 128 filters and a ReLU activation function.

(l) *A Batch Normalization layer.*

(m) A *2D Convolutional* layer with a kernel of size 3x3, 128 filters and a ReLU activation function.

(n) *A Batch Normalization layer.*

(o) A *2D Max Pooling* layer with a kernel of size 2x2.

(p) A *Flatten* layer.

(q) A *Dropout* layer with a rate equal to 0.2. The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting

(r) A *Dense* layer with an 1x1024 vector as the output and ReLU as the activation function.

(s) A *Dropout* layer with a rate equal to 0.2.

(t) A *Dense* layer with an 1x10 vector as the output. This layer is the output layer of the network, therefore we need it to have the same number of outputs as the number of classes. Softmax is used as the activation function.

4. **Train**: This step is identical to that of the MNIST pipeline.

5. **Evaluate**: Having trained the model, we test its performance on samples unknown to it, the test set. We judge its performance based on the accuracy metric.

We evaluated our pipeline for various values of the hyperparamenters. The results with the highest accuracy are:

Epochs	Learning Rate	Optimizer	Batch size	Accuracy
200	0.001	SGD	32	85.03 %
200	0.001	Adam	64	85.03 %
100	0.001	Adam	64	84.67 %
100	0.001	Adam	32	84.24 %

Based on these results, we make the following observations:

- The CIFAR-10 is more complicated than the MNIST dataset. The CIFAR-10 dataset images have color instead of being black and white and they depict real-world objects instead of handwritten digits. As a result, we receive slightly lower accuracy values.
- Increasing the number of epochs seems to increase the accuracy.
- Adam optimizer seems to be the optimal one
- Learning rate = 0.001 seems to be the optimal value.

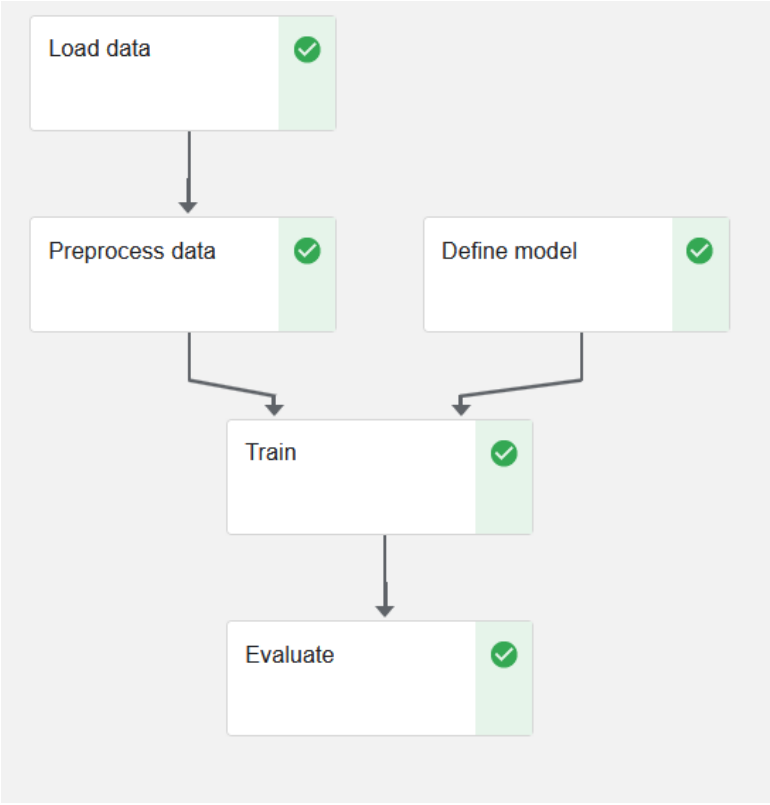


Figure 24: CIFAR-10 Pipeline (Successful Execution)

4.2 TensorFlow Jobs

4.2.1 Train a model using a TensorFlow Job

In the previous section, we trained two models on the MNIST and CIFAR-10 datasets respectively using Machine Learning Pipelines. These pipelines were constructed using Functional Components, Python functions that were converted into pipeline steps. While these components allow us to construct a pipeline easily, they do not enable us to utilize the resources of our cluster fully. For example, they do not provide us with the ability to train a model in parallel. That may not be a problem with simple datasets but for more complicated ones, distributed training is requirement for the process to be completed in acceptable time. Therefore, in this section we will train two Machine Learning models for MNIST and CIFAR-10 again, but we will use a TensorFlow Job in order to implement distributed training.

The strategy we will use to the train our model is known as Multi-Worker Mirrored Strategy. Multi-Worker Mirrored Strategy is a distributed training strategy in TensorFlow that allows training a deep learning model on multiple workers in a distributed environment, where each worker has access to one or more machines. It is used to scale up the training process and reduce the training time of deep learning models.

In Multi-Worker Mirrored Strategy, the training is done by synchronizing the model's parameters across multiple devices on multiple machines. Each worker receives a copy of the model and the training data, and the updates made to the parameters on one worker are shared with the other workers to keep the model synchronized. This way, all workers train on different batches of data at the same time, and the collective gradient is computed and applied to the model's parameters.

The main advantage of using Multi-Worker Mirrored Strategy is that it can speed up the training process of deep learning models significantly. By distributing the training process across multiple workers, each worker can train on a subset of the data, and the entire training process can be completed in less time than it would take on a single machine.

For Kubeflow to execute a TensorFlow Job, an appropriate YAML file must be applied to the cluster. An example of a YAML file for a TensorFlow Job is the following:

```yaml
apiVersion: Kubeflow.org/v1
kind: TFJob
metadata:
  name: mnist
  namespace: Kubeflow
spec:
  tfReplicaSpecs:
    Worker:
      replicas: 4
      restartPolicy: Never
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          containers:
            - name: TensorFlow
              image: TF_IMAGE
              imagePullPolicy: Always
              args:
                - --epochs=5
                - --per_worker_batch=128
                - --optimizer=1,
                - --learning_rate=0.001,
                - --momentum=0.9
```

The most important properties of the YAML file are the following:

- **apiVersion**: This field specifies which version of the TFJob custom resource you are using. The corresponding version (in this case v1) needs to be installed in your Kubeflow cluster.
- **kind** This field identifies the type of the custom resource—in this case a TFJob.
- **metadata**: This field is common to all Kubernetes objects and is used to uniquely identify the object in the cluster — you can add fields like name and namespace here.
- **tfReplicaSpecs**: This field is the most important part of the schema. This is the actual description of your TensorFlow training cluster and its desired state. This field is comprised of :
  - **Worker.replicas** : We have 4 Workers that will be working in parallel.
  - **Worker.spec.containers.image** : Here we name the docker image that the containers will use. The image was built and pushed to DockerHub beforehand.

– **Worker.spec.containers.args** : The arguments that the Python script contained with the image will require. Here we include arguments for the number of epochhs, the size of the batch, the type of the optimizer (0 for SGD and 1 for Adam), the learning rate and the momentum.

MNIST

At the evaluation of the model, similar accuracy results to the ones in the previous section are yielded. As we expected, increasing the number of workers, decreased the time required for the training to complete. The time required for each training process depending on its number of workers is shown in Figure 25.

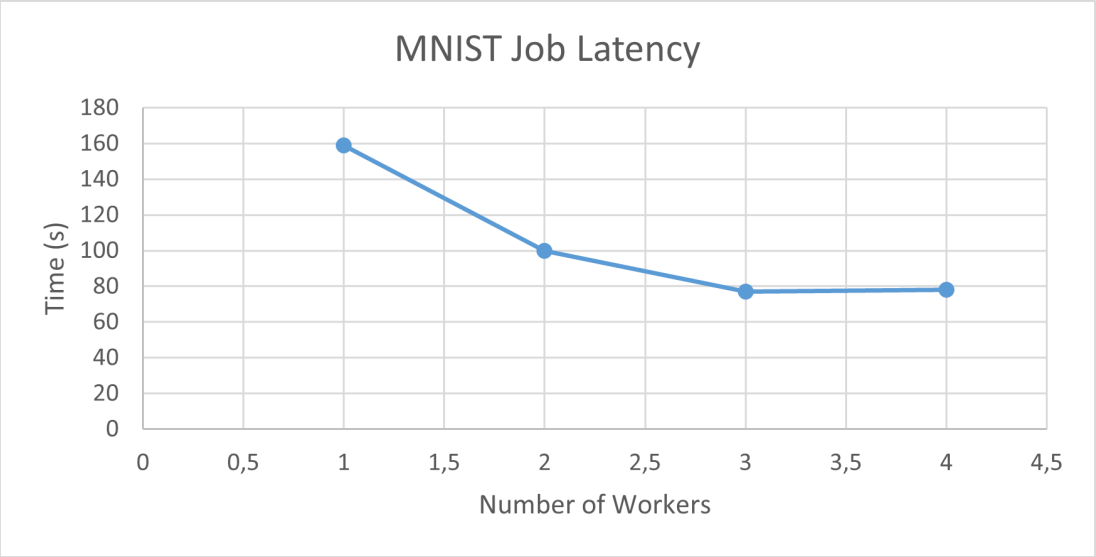
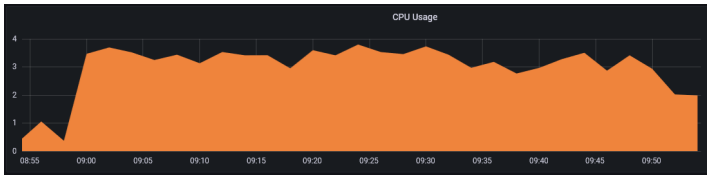


Figure 25: MNIST TFJob Latency

In Figures 26 and 27 , we observe the toll that our TensorFlow Application takes on our cluster in terms of CPU, memory and storage resources. These graphs as well as similar graphs for our next experiments were extracted from Grafana, which uses Prometheus as a data source.

(a) CPU



(b) Memory

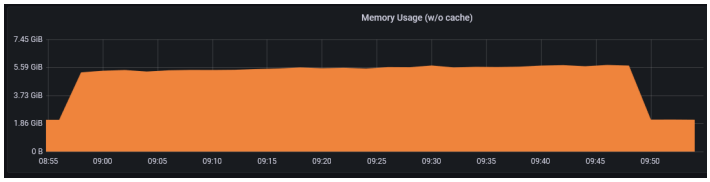
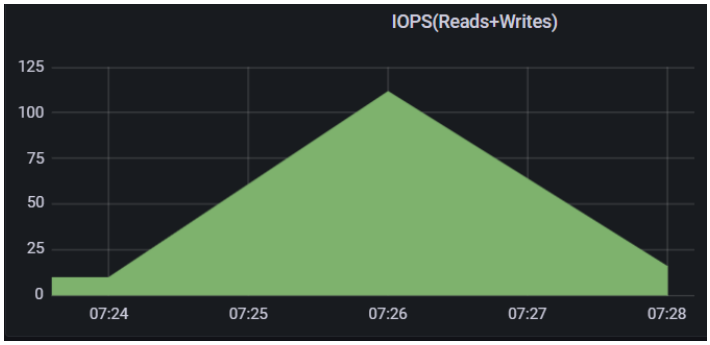


Figure 26: Resources required by the MNIST TensorFlow Application (1)

(a) Storage - IOPS



(b) Storage - Throughput

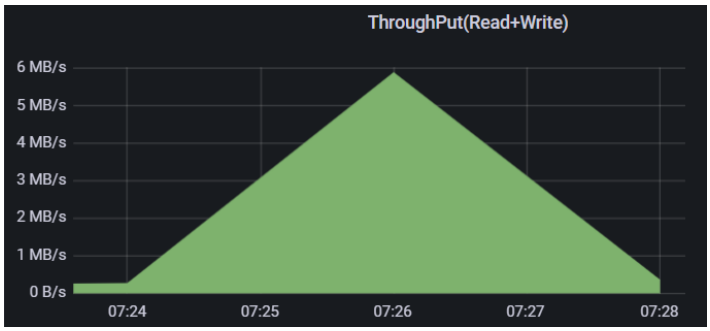
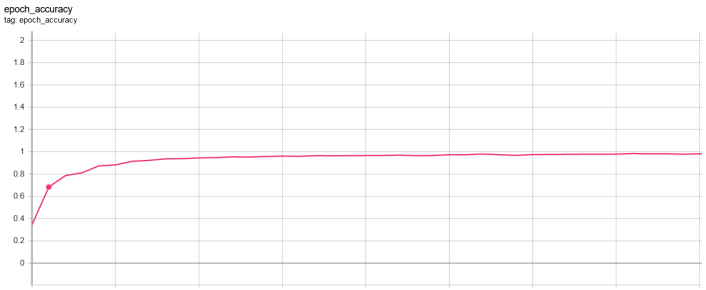


Figure 27: Resources required by the MNIST TensorFlow Application (2)

Finally, using TensorBoard, we display certain Figures (28) concerning the training process of our model. The model was trained for 60 epochs. As we anticipated, the accuracy tends to become 1 and the loss 0 as the epochs progress.

(a) Accuracy per epoch



(b) Loss per epoch

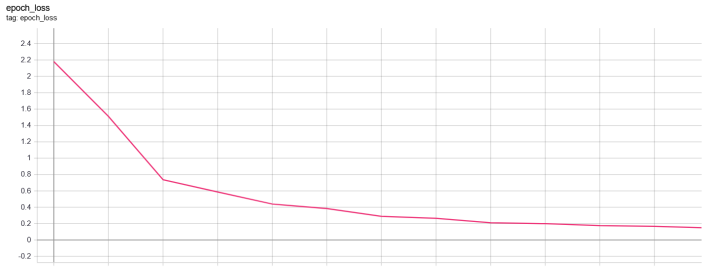


Figure 28: Observing the training process using the accuracy and loss metrics

CIFAR-10

At the evaluation of the model, similar accuracy results to the ones in the previous section are yielded. Again, increasing the number of replicas decreases the time required like we expected. The time required for each training process depending on its number of workers is shown in Figure 29.

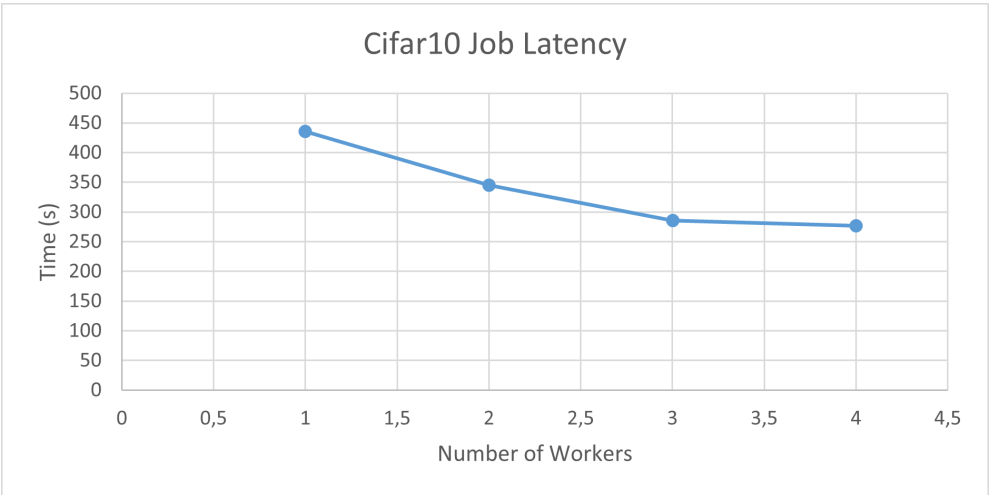
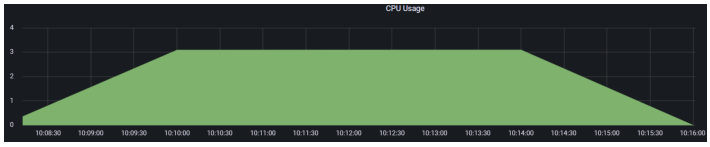


Figure 29: CIFAR-10 TFJob Latency

In Figures 30 and 31, we observe the toll that our TensorFlow Application takes on our cluster in terms of CPU, memory and storage resources.

(a) CPU



(b) Memory

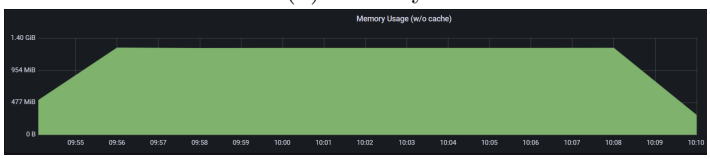
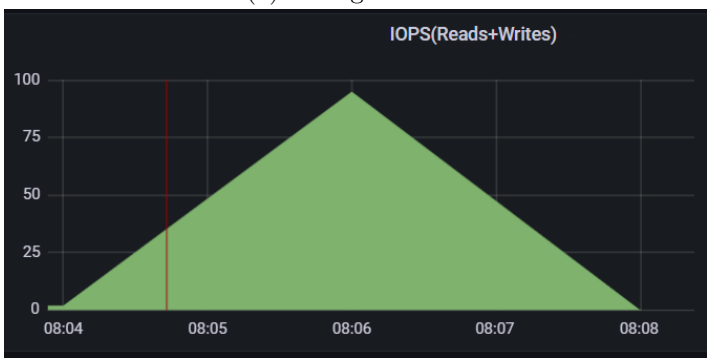


Figure 30: Resources required by the CIFAR-10 TensorFlow Application (1)

(a) Storage - IOPS



(b) Storage - Throughput

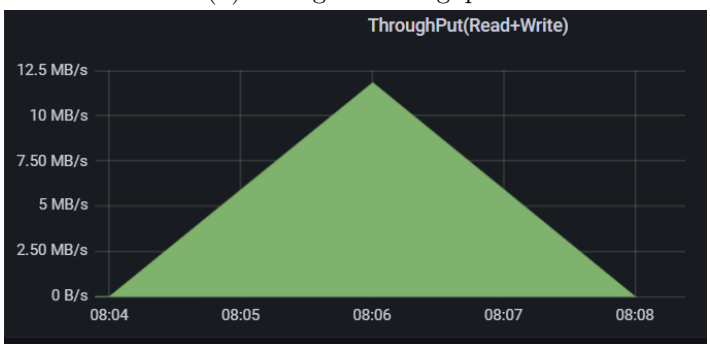
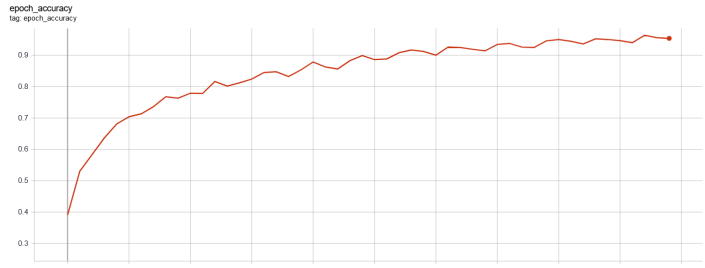


Figure 31: Resources required by the CIFAR-10 TensorFlow Application (2)

Finally, using TensorBoard, we display certain Figures (32) concerning the training process of our model. The model was trained for 60 epochs. As we anticipated, the accuracy tends to become 1 and the loss 0 as the epochs progress.

(a) Accuracy per epoch



(b) Loss per epoch

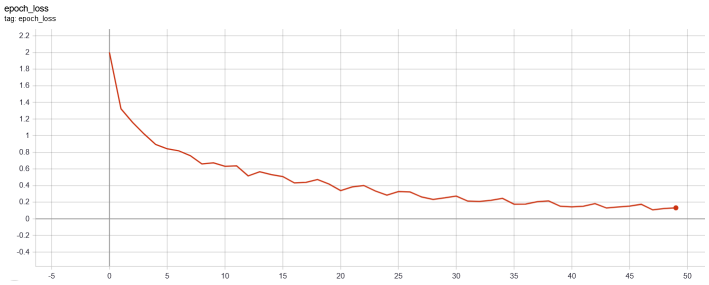


Figure 32: Observing the training process using the accuracy and loss metrics

4.3 Working on a more complicated dataset

4.3.1 The Common Crawl dataset

In this section, we will perform some experiments on the Common Crawl dataset [21], the dataset that the artificial intelligence chatbot ChatGPT by OpenAI [22] was trained on. The Common Crawl dataset is a massive corpus of web crawl data that is freely available to the public. It is designed to provide a comprehensive snapshot of the internet, with data collected from billions of web pages.

Here are some key details about the Common Crawl dataset:

- **Size**: The Common Crawl dataset is one of the largest publicly available web crawl datasets. As of 2021, it contains over 150 petabytes of data, with billions of web pages included.
- **Frequency**: The dataset is updated on a regular basis, with new crawls being added on a monthly basis.
- **Format**: The data is provided in several formats, including WARC, WAT, and WET. These formats are designed to make it easy to work with the data and extract information from it. The WARC files which store the raw crawl data, the WAT files which store computed metadata for the data stored in the WARC and the WET files which store extracted plain text from the data stored in the WARC.

- **Coverage**: The Common Crawl dataset covers a wide range of languages, with data collected from websites in over 200 countries.
- **Access**: The dataset is freely available to the public, with no restrictions on usage. This makes it a valuable resource for researchers, developers, and anyone else who wants to analyze or work with web data.

Overall, the Common Crawl dataset is an incredibly valuable resource for anyone interested in web data and analysis. Its massive size and frequent updates make it an ideal tool for studying trends in web content, tracking changes over time, and much more. Due to the limitations on resources, we will experiment on a sample dataset, which contains website data from March 2017.

4.3.2 Spark Applications

In this section, we will attempt to extract information from the datasets using the spark-on-k8s operator of Kubeflow, to run Spark Applications on the data. The data were extracted and uploaded to HDFS, from where they will be loaded by the Spark Applications. The Spark Applications will also write their output to HDFS, so that we have easy access to it.

Word Count

For our first experiment, we will count the frequency of each word in the scraped websites and we will sort them in descending order. We will use the the WET file for this purpose. The YAML file of the Spark Application that we used is the following:

```
apiVersion: sparkoperator.k8s.io/v1beta2
kind: SparkApplication
metadata:
  name: wordcount
  namespace: default
spec:
  type: Python
  pythonVersion: '3'
  mode: cluster
  image: docker.io/chrz95/spark-py:commonCrawl
  imagePullPolicy: Always
  mainApplicationFile: hdfs://10.0.1.66:9000/scripts/wordCount.py
  sparkVersion: 3.1.1
  hadoopConf:
    fs.defaultFS: hdfs://10.0.1.66:9000
  restartPolicy:
```

```
    type: OnFailure
    onFailureRetries: 1
    onFailureRetryInterval: 10
    onSubmissionFailureRetries: 5
    onSubmissionFailureRetryInterval: 20
  driver:
    cores: 1
    coreLimit: 1200m
    memory: 512m
    labels:
      version: 3.1.1
    serviceAccount: spark
  executor:
    cores: 1
    instances: 1
    memory: 512m
    labels:
      version: 3.1.1
```

The most important properties of the YAML file are the following:

- **apiVersion**: This field specifies which version of the SparkApplication custom resource you are using. The corresponding version (in this case v1beta2) needs to be installed in your Kubeflow cluster.
- **kind** This field identifies the type of the custom resource — in this case a SparkApplication.
- **metadata**: This field is common to all Kubernetes objects and is used to uniquely identify the object in the cluster — you can add fields like name and namespace here.
- **spec.image** : Here we name the docker image that the containers will use. The image was built and pushed to DockerHub beforehand.
- **spec.mainApplicationFile** : The location of the Python file that will be executed. In this case, we utilize PySpark and we store the file in HDFS.
- **driver/executor** : The resources (CPU and Memory) that the driver and executor workers will utilize to perform their tasks. The instances parameters indicates how many executors will be created to perform this task in parallel.

The stages of the Spark Application are shown in Figure 33. The first stage includes reading the dataset, mapping the words to pairs ((word,1)) and then performing a reduce operation to get the appearance frequency of each word.

The second stages sorts the frequencies in descending order. The other Spark Applications are comprised of similar stages.

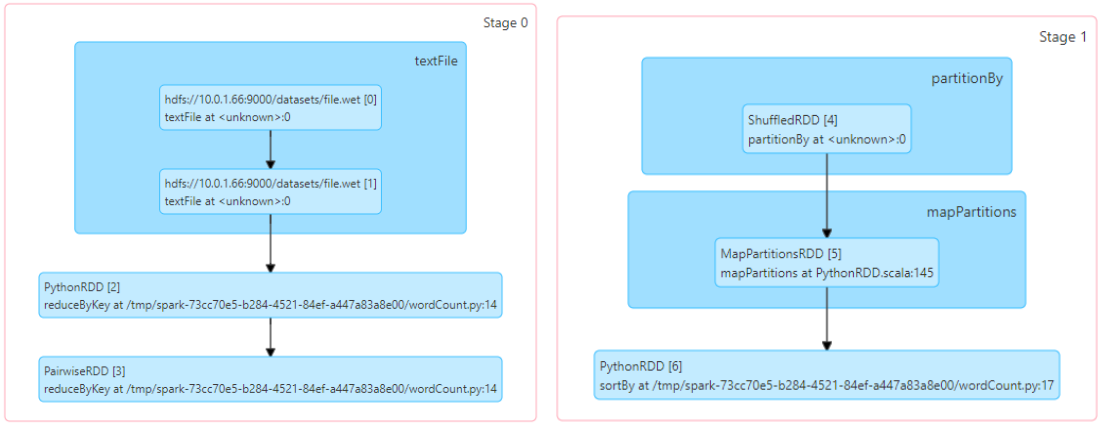


Figure 33: The stages of the Word Count Application

The results we received (the first six words) are the following:

Word	**Count**
the	600720
and	450283
to	448739
of	395002
a	371072
in	298380

The results come as no surprise, since these words are some of the mostly used in the English language.

In Figure 34, we can see the execution time decreases while the number of executors increase, which is to be expected. However, when the number of executors becomes greater than 3, the executor time remains stable. That is due to the synchronization overhead of the distributed execution and the limitations on the resources or our cluster.

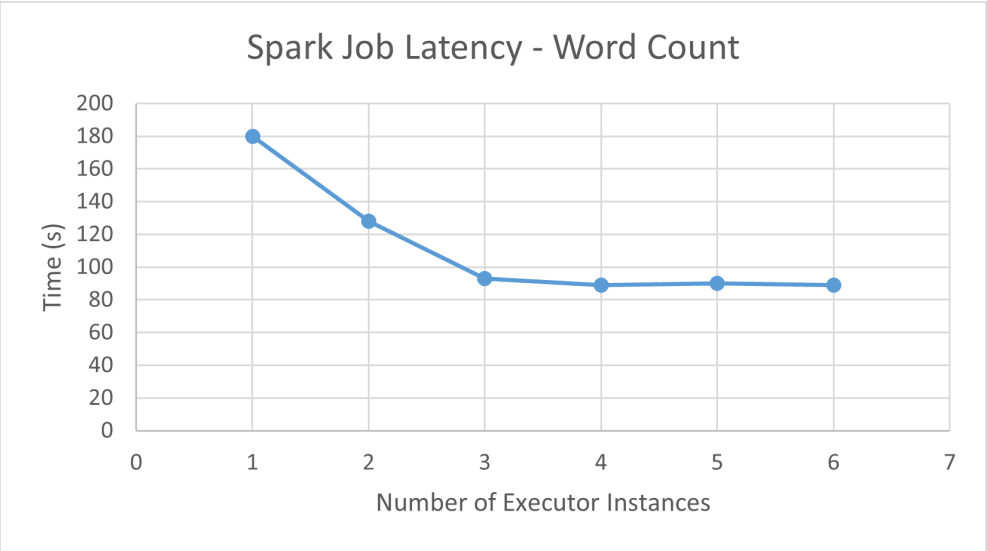
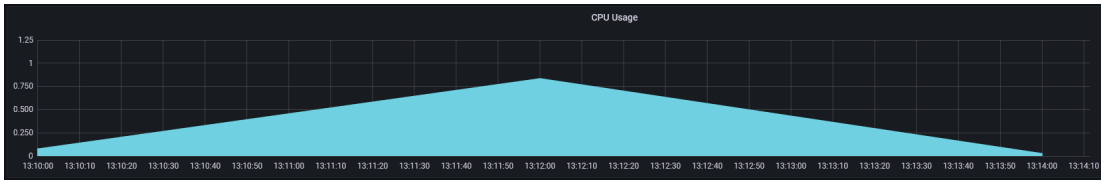


Figure 34: Execution Time per number of Executors - Word Count

In Figures 35 and 36, we observe the toll that our Spark Application takes on our cluster in terms of CPU, memory and storage resources.

(a) CPU



(b) Memory

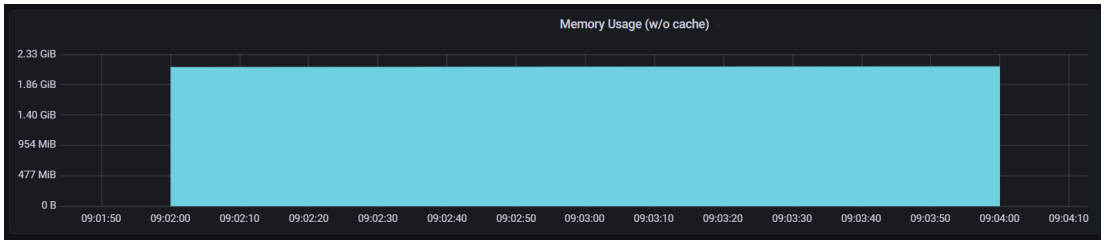
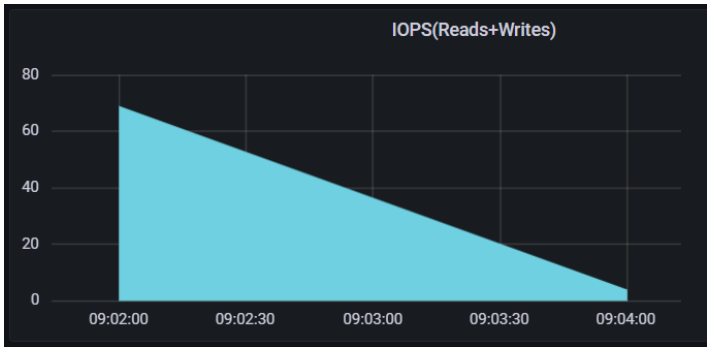


Figure 35: Resources required by the Word Count Application (1)

(a) Storage - IOPS



(b) Storage - Throughput

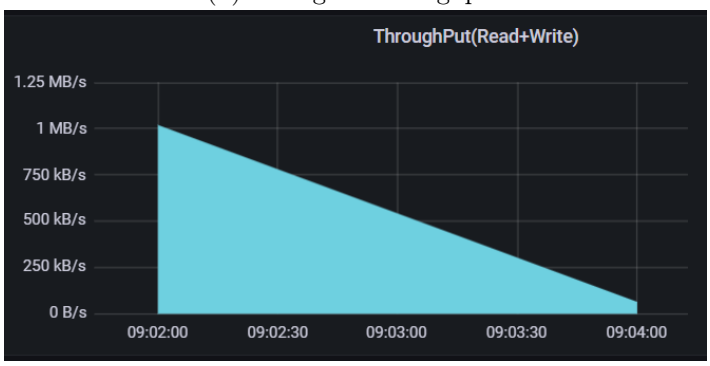


Figure 36: Resources required by the Word Count Application (2)

Count HTML Tags

For our next experiment, we will count the HTML Tags used in the scraped websites. We will use the the WARC file for this purpose. The YAML file we used here is very similar to the one in the previous experiment, but it uses a different mainApplicationFile.

The stages of the Spark Application are shown in Figure 37. They are similar to the previous application.

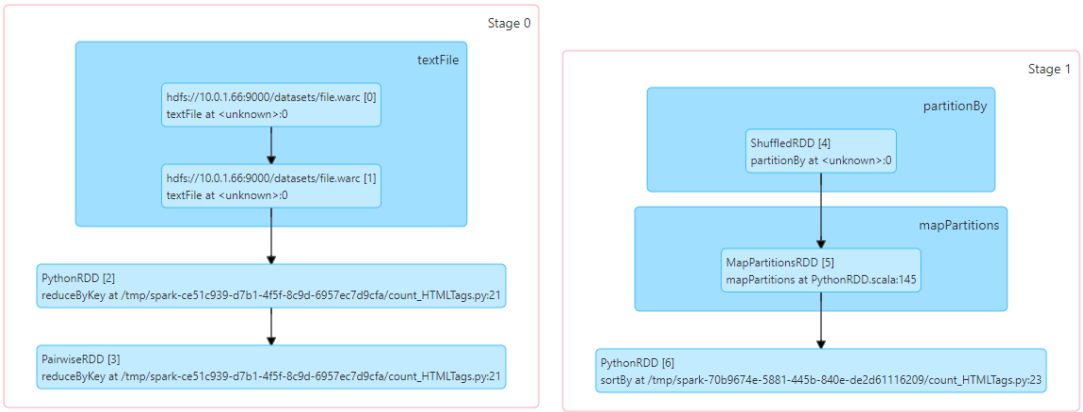


Figure 37: The stages of the Count HTML Tags Application

The results we received (the first six HTML tags) are the following:

HTML Tag	Count
a	8667841
div	7451445
li	5191723
span	4306088
img	1466498
td	1283063

Again the results are to be expected, especially from someone used to HTML code, such as a Web Developer, since these tags are very frequently used in all websites.

In Figure 38, we can see the execution time decreases while the number of executors increase, which is to be expected. However, when the number of executors becomes greater than 5, the executor time remains stable. That is due to the synchronization overhead of the distributed execution and the limitations on the resources or our cluster.

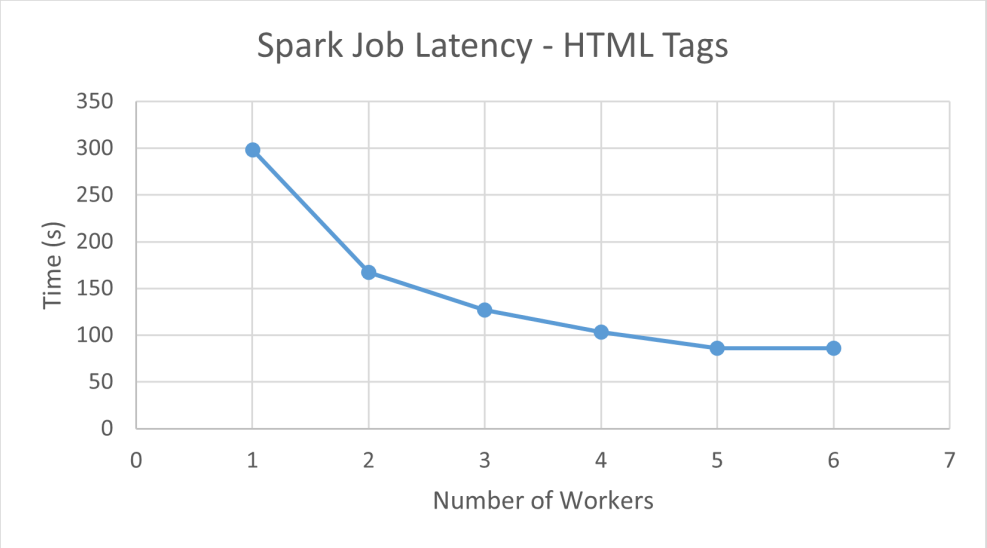
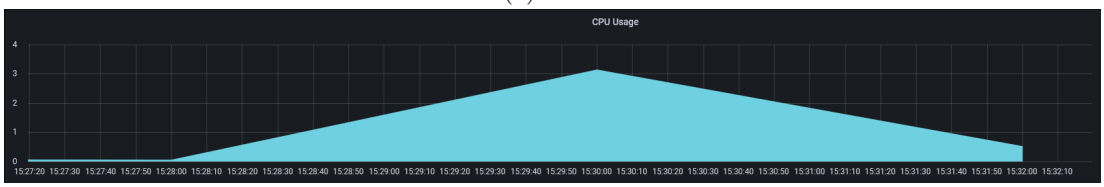


Figure 38: Execution Time per number of Executors - Count HTML Tags

In Figures 39 and 40, we observe the toll that our Spark Application takes on our cluster in terms of CPU, memory and storage resources.

(a) CPU



(b) Memory

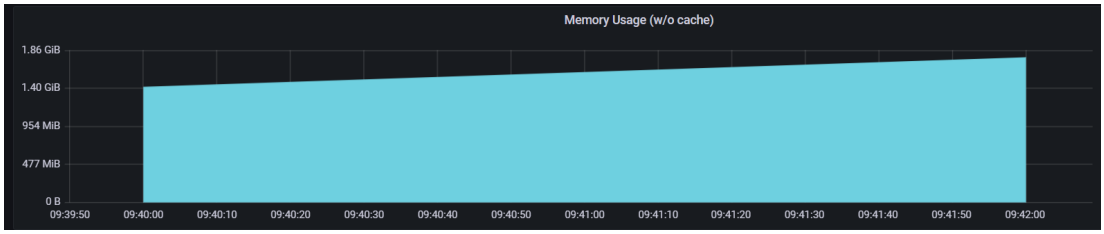
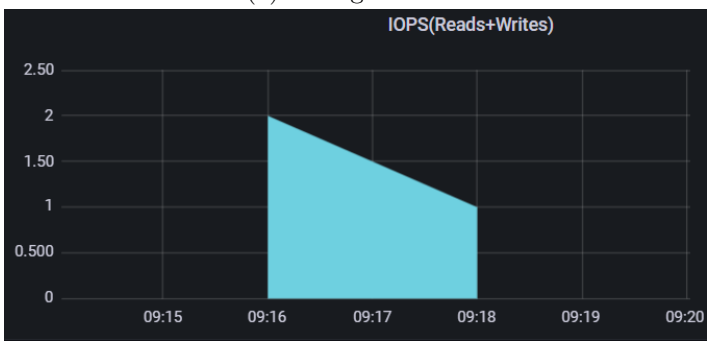


Figure 39: Resources required by the Count HTML Tags Application (1)

(a) Storage - IOPS



(b) Storage - Throughput

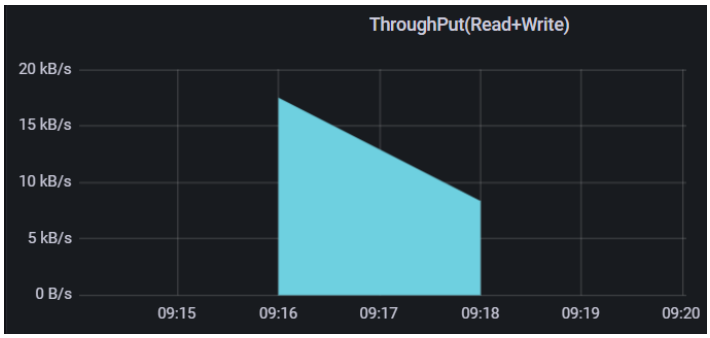


Figure 40: Resources required by the Count HTML Tags Application (2)

Count Web Server Names

For our next experiment, we will count the Web Server name that were used to host the scraped websites. We will use the the WARC file for this purpose. The YAML file we used here is very similar to the ones in the previous experiments, but it uses a different mainApplicationFile.

The stages of the Spark Application are shown in Figure 41. They are similar to the previous application.

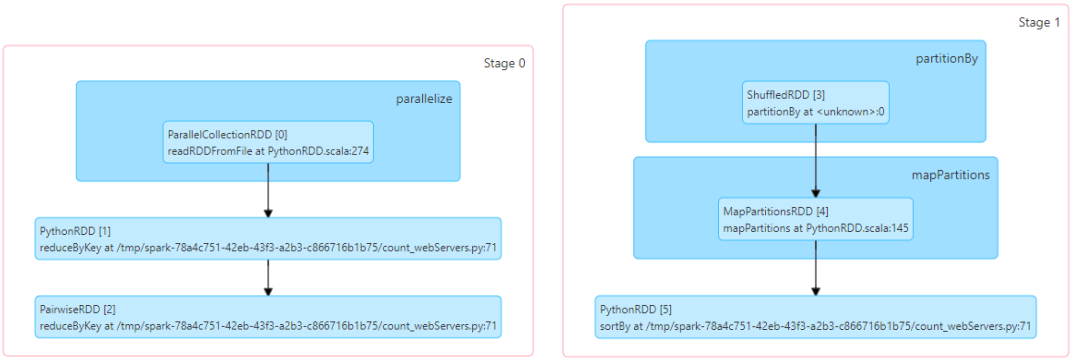


Figure 41: The stages of the Count Webservers Application

The results we received (the first six servers) are the following:

Server	Count
Apache	9084
nginx	7543
cloudflare-nginx	3538
GSE	2562
Microsoft-IIS/7.5	1802
Microsoft-IIS/8.5	1412

In Figure 42, we can see the execution time decreases while the number of executors increase, which is to be expected. However, when the number of executors becomes greater than 5, the executor time remains stable. That is due to the synchronization overhead of the distributed execution and the limitations on the resources of our cluster.

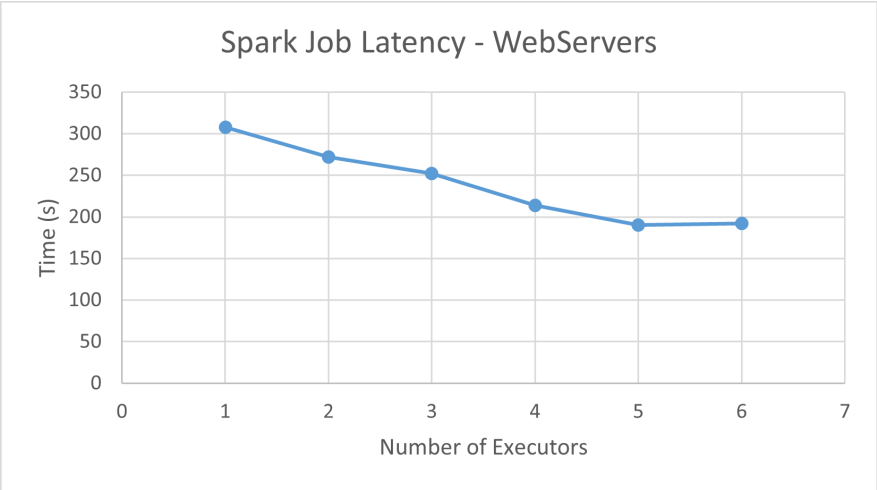
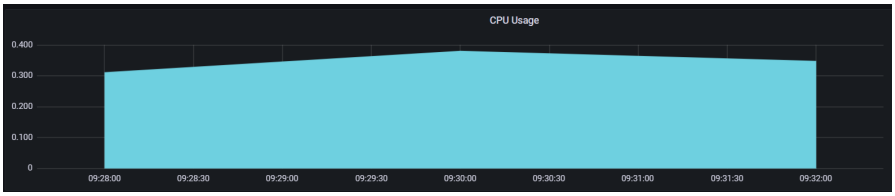


Figure 42: Execution Time per number of Executors - Count Webservers

In Figures 43 and 44, we observe the toll that our Spark Application takes on our cluster in terms of CPU, memory and storage resources.

(a) CPU



(b) Memory

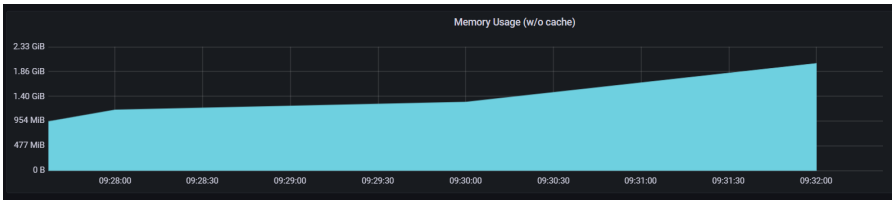
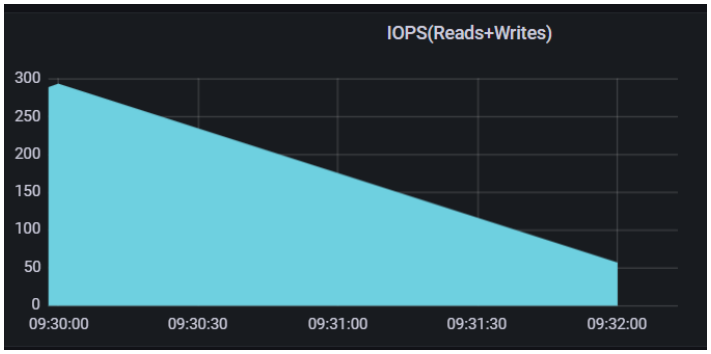


Figure 43: Resources required by the Count Web Servers Application (1)

(a) Storage - IOPS



(b) Storage - Throughput

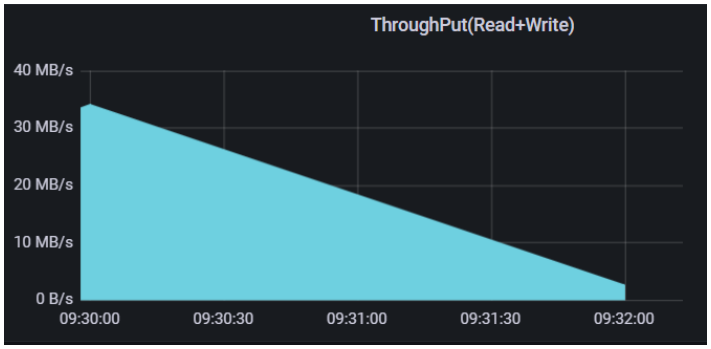


Figure 44: Resources required by the Count Web Servers Application (2)

List of hostnames and IP addresses

For our next experiment, we will count the IP Addresses and corresponding hostnames that were used from the scraped websites. We will use the the WARC file for this purpose. The YAML file we used here is very similar to the ones in the previous experiments, but it uses a different mainApplicationFile.

The stages of the Spark Application are shown in Figure 45. They are similar to the previous application.

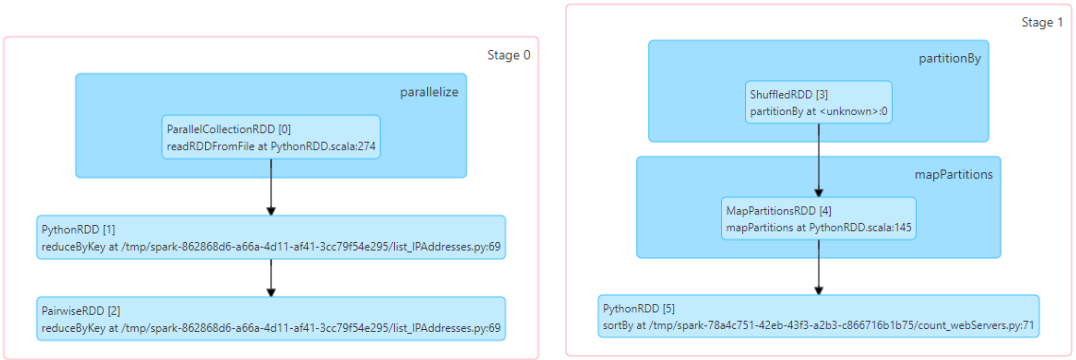


Figure 45: The stages of the Count IP Addresses Application

The results we received (the first five IP Addresses) are the following:

IP Address	Count
172.217.7.129	403
172.217.5.225	235
172.217.3.33	230
172.217.7.161	226
172.217.7.225	221

All these IP addresses belong to Google servers. More specifically, the first IP address corresponds to the Google server in Salt Lake City of the United States.

In Figure 46, we can see the execution time decreases while the number of executors increase, which is to be expected. However, when the number of executors becomes greater than 5, the executor time remains stable. That is due to the synchronization overhead of the distributed execution and the limitations on the resources or our cluster.

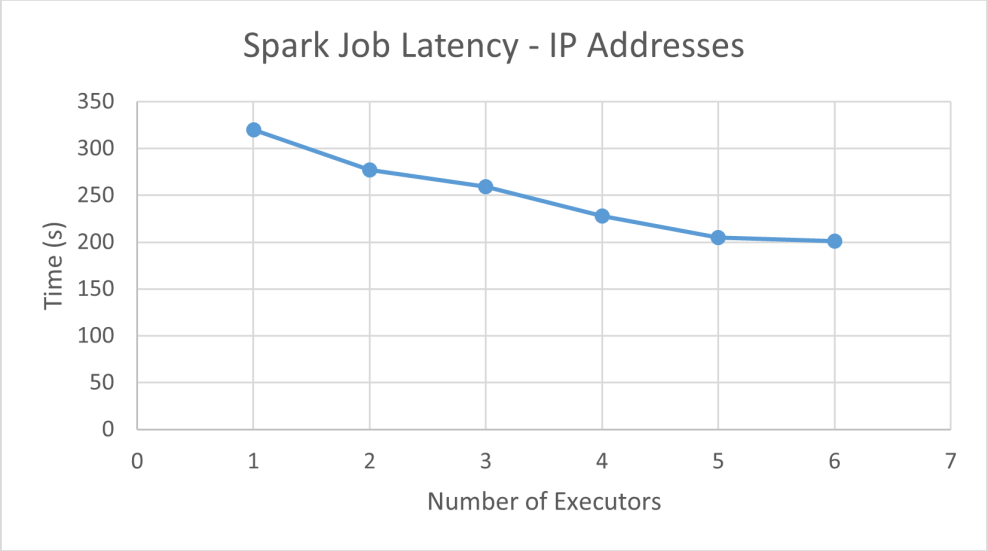
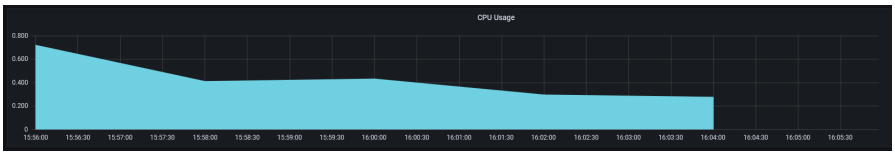


Figure 46: Execution Time per number of Executors - Count IP Addresses

In Figure 47 and 48, we observe the toll that our Spark Application takes on our cluster in terms of CPU, memory and storage resources.

(a) CPU



(b) Memory

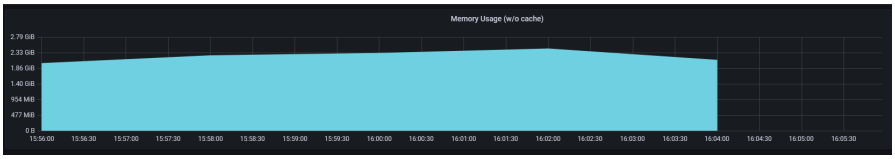
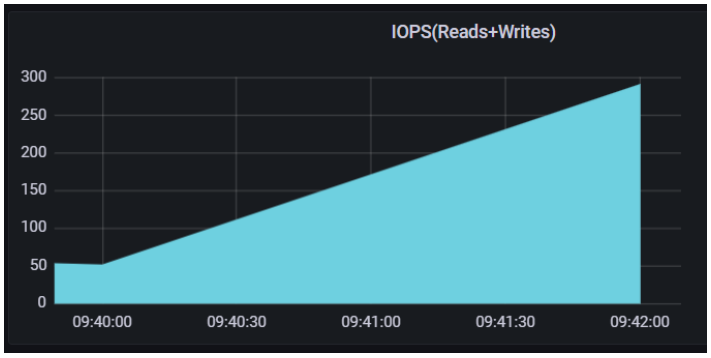


Figure 47: Resources required by the Count IP Addresses Application (1)

(a) Storage - IOPS



(b) Storage - Throughput

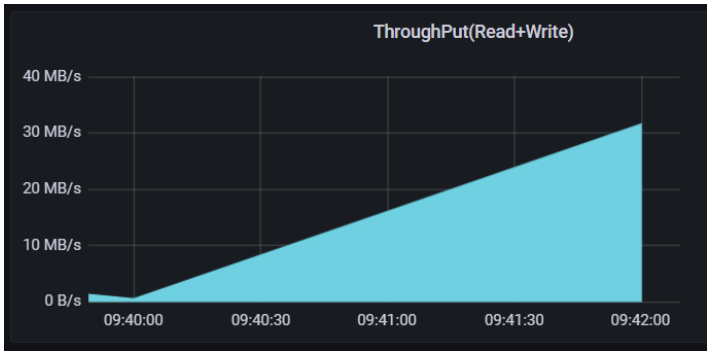


Figure 48: Resources required by the Count IP Addresses Application (2)

5 Conclusion

5.1 Recap

During the implementation of this thesis, we :

- We learned about the usefulness of containerized applications and container engines such as Docker and how to use them to deploy our own applications.
- In order to execute such applications on a cloud infrastructure, we created a Kubernetes cluster using Virtual Machines to simulate such an infrastructure.
- We learned the fundamentals of certain Machine Learning and Data Science techniques and applied them on public datasets such as MNIST, CIFAR10 and Common Crawl using Kubeflow, a program specifically created to execute Machine Learning applications on Kubernetes.
- In order to store the data and the results of our experiments, we installed the Apache Hadoop framework and the corresponding Hadoop Distributed File System (HDFS) on our cluster of Virtual Machines.
- Finally, in order to monitor our experiments and their impact on the cluster, we installed and utilized tools such as Prometheus, Grafana and TensorBoard.

5.2 Future Work

Our work may be expanded upon in a number of ways. Some of which we recommend are:

- Use the same techniques on many different datasets and observe the results.
- During this thesis, we mainly used (Convolutional) Neural Networks to perform classification experiments. However, the TensorFlow library offers many more capabilities and models, such as LSTM Neural Networks and Transformers, which can yield very interesting results.
- The Common Crawl dataset on which we performed some simple experiments, contains the information of the entirety of the Internet and it is the dataset on which the famous chatbot ChatGPT was trained on. This fact alone attests to the limitless potential that this data contains.

References

[1] Kubernetes - Production-Grade Container Orchestration - https://kubernetes.io/

[2] Kubeflow - The Machine Learning Toolkit for Kubernetes - https://www.kubeflow.org/

[3] Apache Hadoop - Open-source software for reliable, scalable, distributed computing - https://hadoop.apache.org/

[4] Apache Spark - Unified engine for large-scale data analytics - https://spark.apache.org/

[5] The Kubernetes Book - Nigel Poulton

[6] The official Docker web page - https://www.docker.com/

[7] DockerHub - Container Image Library - https://hub.docker.com/

[8] Prometheus - Kubernetes Metrics Collection Tool - https://prometheus.io/

[9] Grafana - Analytics monitoring solution - https://grafana.com/

[10] Ansible Github repository - https://github.com/ansible/ansible

[11] KubeFlow Pipelines - Machine Learning Workflows Framework - https://www.kubeflow.org/docs/components/pipelines/v1/introduction/

[12] TensorFlow - Machine Learning Framework - https://www.tensorflow.org/

[13] TensorBoard - TensorFlow Monitoring Tool - https://www.tensorflow.org/tensorboard

[14] OpenStack - Open Source Cloud Computing Infrastructure - https://www.openstack.org/

[15] KubeSpray GitHub Repo - Kuberneres Deployment Tool - https://github.com/kubernetes-sigs/kubespray

[16] Charmed Kubeflow - https://charmed-kubeflow.io/

[17] Juju - Charmed Operator Framework - https://juju.is/

[18] Helm - https://helm.sh/

[19] MNIST - Digits Recognition Dataset - https://www.tensorflow.org/datasets/catalog/mnist

[20] Cifar10 - Objects Recognition Dataset - https://www.tensorflow.org/datasets/catalog/cifar10

[21] Common Crawl - https://commoncrawl.org/

[22] ChatGPT - https://chat.openai.com/chat