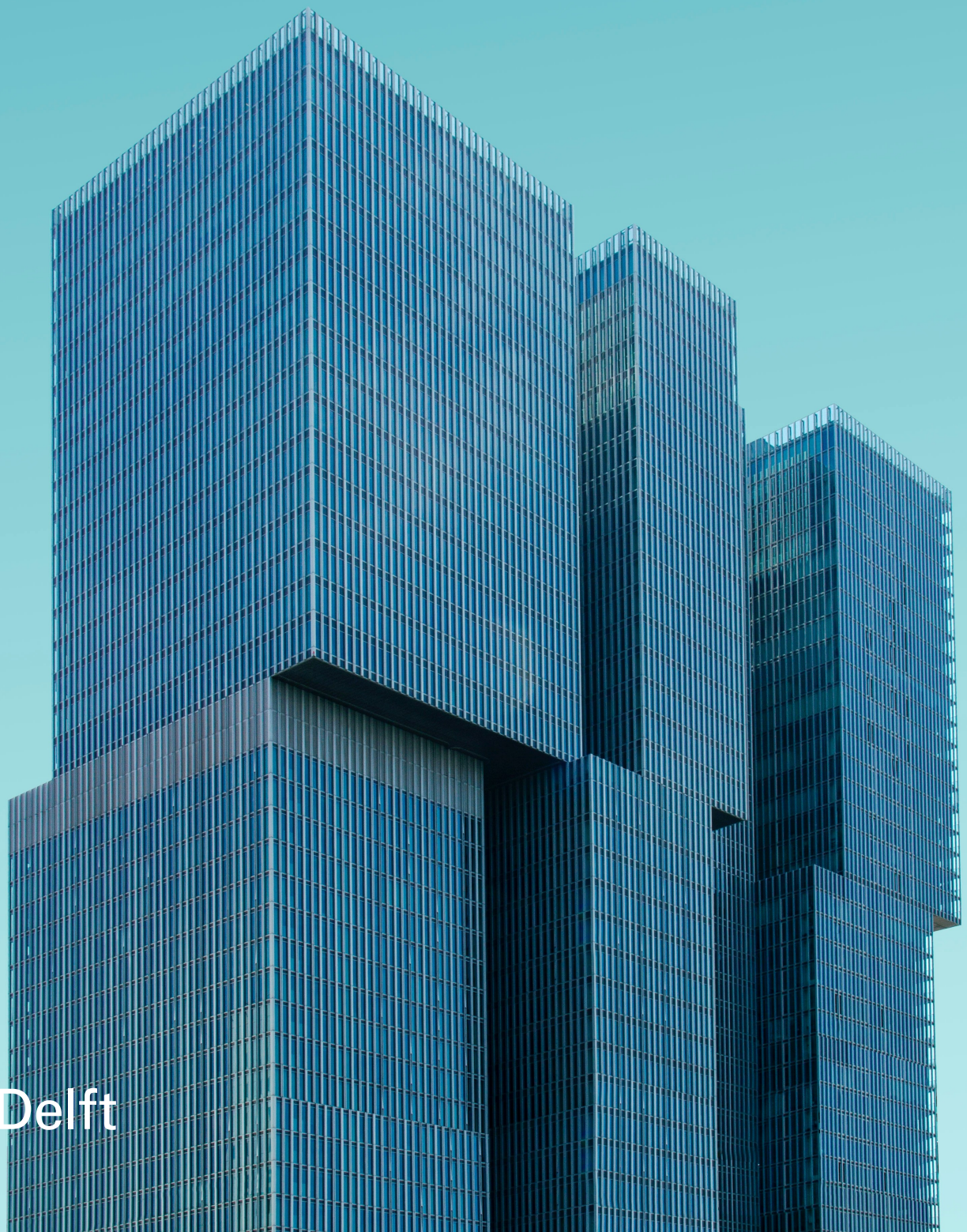
KubeML

An Efficient Serverless Platform for Scalable Deep Learning

Diego Albo Martínez

Technische Universiteit Delft



TU Delft

KubeML

An Efficient Serverless Platform for Scalable Deep Learning

by

Diego Albo Martínez

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday June 17, 2021 at 10:00 AM.

Student number:	5043204	
Project duration:	November 9, 2020 – June 17, 2021	
Thesis committee:	Dr. J.S. Rellermeyer,	TU Delft, supervisor
	Prof. dr. ir. D.H.J. Epema,	TU Delft
	Dr. A. Katsifodimos,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Serverless computing is an emerging paradigm for structuring applications in such a way that they can benefit from on-demand computing resources and achieve horizontal scalability. As such, it is an ideal substrate for the resource-intensive and often ad-hoc task of training deep learning models. However, the design and stateless nature of serverless platforms make it difficult to translate distributed machine learning systems directly to this new world. With KubeML, we present a purpose-built serverless machine learning system that runs atop Kubernetes and seamlessly embeds into the popular PyTorch framework. Unlike alternative systems, KubeML fully embraces GPU acceleration and is able to outperform TensorFlow, especially with smaller local batches, while allowing for higher resource density. KubeML reaches a 3.98x faster time-to-accuracy with small batch sizes, and maintains a 2.02x speedup between the top results of both platforms for commonly benchmarked machine learning models like ResNet34.

Preface

This thesis is the result of 7 months of work that started last November with a rather vague idea and eventually turned into a project that I can really feel proud of. This work also puts an end to two years studying at TU Delft, which have been a great experience of adapting to a new culture and environment, and also have helped me discover new areas of work that I am passionate about. Along the process of my studies, I have met many talented and wonderful people which have in one way or the other had an effect on this project, and who I want to thank for their support.

First and foremost, I would like to thank my supervisor, Jan Rellermeyer. When I approached you more than a year ago, I only had a small idea of what I wanted to work on for my thesis, but you were able to point me in the direction of a project that mixed all of my interests, and allowed me to take on technologies and problems that truly excited me. I would also like to thank you for supporting me in our weekly meetings and discussions, and pushing me towards new objectives and milestones.

Second, I would like to thank all the people that contributed in other ways to this project. In these times of lock-downs and restrictions, having people to spend time with in any way possible becomes more important than ever, so I want to thank all of my friends, both in Delft, back in Spain, and everywhere else for the fun times, calls and chats that also helped make this project the way it is.

Last, but certainly not least, I want to thank my family and my girlfriend for all the love and support, and for being the ones that had to put up with me the most during these years. I want to especially dedicate this thesis to my grandparents, who I was not able to see for almost a year, but have always been my biggest supporters.

Diego Albo Martínez
Delft, June 2021

Contents

1 Introduction

Artificial intelligence (AI) has become a transformational force to traditional services and business logic while at the same time enabling exciting and demanding new applications such as self-driving cars [4]. Due to advances in hardware technology, artificial neural networks can now learn patterns from myriads of available sources in order to fuel data-driven decision making in modern organizations. The problem of training machine learning models is, however, still a highly resource-intensive endeavor since the demand for training data and therefore the effort to train the model tends to grow exponentially with the size of the model [57]. At this point, the jury is still out on whether large-scale machine learning will follow a trajectory similar to High Performance Computing (HPC) and become mission critical enough for organizations to afford the best available (even if custom and exotic) hardware, or if it will trend towards commoditization and the cloud [66]. If the latter is true, we should expect scale-out architectures to become the dominant platform for training.

A great example of the commoditization of resources is the one of Serverless Computing. This new computing paradigm lets users focus on the application logic while deployment and management details are abstracted and handled by a cloud platform or provider. Serverless also follows a clear scale-out approach, opting for running complex tasks as many small functions running in parallel instead of increasing performance by investing in more powerful hardware [22]. Its ease of use and its unlimited scalability make serverless a tempting route to consider when implementing distributed machine learning systems. However, some obstacles prevent the adoption of serverless as a tool for accessible distributed deep learning such as restricted communication and hardware platforms [6].

In this work, we present KubeML, a platform that is able of providing machine learning jobs with access to accelerators like high-end GPUs, even in multi-tenant environments, while allowing for frictionless scaling on large clusters and the cloud. KubeML is designed to operate on top of Kubernetes[1] and offer a serverless experience to users while seamlessly embedding into into PyTorch [52], one of the most popular machine learning libraries. By doing so, it absorbs the complexity of partitioning the training jobs and managing the deployment to the cluster. In our experiments, we were able to show that KubeML scales better than state-of-the-art systems like TensorFlow [1] for small and medium size models by achieving comparable accuracy in a fraction of the time. KubeML is the first system to apply a serverless paradigm to distributed deep learning while integrating GPUs as first-level citizens.

With KubeML, we propose solutions for the commonly discussed drawbacks of serverless computing which have limited its adoption in the distributed deep learning field. We investigate a new architecture that is able to incorporate vital support for GPUs, and discuss and analyze multiple options to improve communication efficiency while adhering to traditional serverless restrictions. We also focus on maintaining a core principle of serverless such as ease of use.

[1] `https://kubernetes.io`

1.1. Problem Statement

Based on the characteristics of Serverless Applications discussed beforehand, and the restrictions derived from there, we identify the key characteristics to be implemented to focus both on performance and usability.

1. Ease of use and portability
2. Utilization of GPU Resources
3. Efficient Communication Model

As we will observe in the coming sections, most of the work developing applications on top of FaaS do so by using solutions provided by cloud platforms, with AWS Lambda being the most popular. However, one of the main shortcomings of employing these proprietary cloud solutions is the resulting **vendor lock-in** of the application. With the libraries and architecture of each of the services differing between the options, changing between providers implies migrating to the related services offered by the alternative provider. This is a significant important obstacle to be overcome in order to truly develop a portable and convenient solution for users.

Another issue with these platforms often experienced by practitioners is the lack of **local testing** tools to develop applications. Since these platforms are proprietary, code needs to be tested in the cloud rather than locally, thus slowing down the development process as well as increasing the cost.

Finally, none of these platforms offer as of today support for the usage of hardware typically used for Deep Learning such as GPUs and TPUs. We argue that for a solution to be adopted, there needs to be a significant performance equivalence or improvement with respect to traditional systems. In terms of Deep Learning, this translates into the requirement of **GPU availability** to accelerate the training process of neural networks and putting it on par with non-serverless systems.

Additionaly, many of the well known deep learning systems use highly optimized collective communication libraries to communicate efficiently between different workers, like MPI [43], Facebook Gloo [15], or directly between GPUs wih Nvidia NCCL[2]. With the restriction of serverless functions not being able to communicate directly, we must find an **efficient communication** solution which compensates for the lack of optimization in this regard.

1.2. Research Questions

Taking into account the problems present nowadays in serverless computing, preventing it from accommodating deep learning tasks, we define the following research questions which we will address in this thesis:

(RQ1) How can we design a serverless system specialized for deep learning tasks?

(RQ2) How can we take advantage of GPU resources in a serverless environment?

(RQ3) How can we optimize communication to improve latency in a serverless environment?

(RQ4) How can we minimize configuration overhead and changes to local code to create deep learning tasks?

(RQ5) How well does the proposed system perform when compared to state-of-the-art distributed deep learning systems?

1.3. Organization

Towards answering RQ1, we study the different approaches for distributing deep learning tasks and developing serverless applications in Chapter 2. In Section 2.1 and Section 2.2 we focus on distributed deep learning and its properties such as communication and statistical efficiency. We also investigate the different options available in terms of serverless platforms in Section 2.4, and study which frameworks allow us the flexibility that we are looking for. Based on the conclusions

[2] https://github.com/NVIDIA/nccl

of Section 2.4, and the previous research on serverless ML that we summarize in Section 2.5, we choose a platform that lets us include GPUs into KubeML, thus developing RQ2.

In Chapter 3, we provide an in-depth look at KubeML's architecture, and discuss how we can optimize communication to make up for the inconveniences characteristic of serverless (RQ3). We also propose a way to reduce the amount of effort needed to transition from local code to training distributedly on KubeML by introducing a custom Python library, which streamlines job deployment, and addresses RQ4.

Finally, we address RQ5 in Chapter 4, comparing KubeML in a variety of scenarios with a state-of-the-art deep learning system such as TensorFlow. On top of that, we also analyze properties such as the effect of communication efficiency in the final result, and the resource usage of KubeML.

2

Background & Related Work

In this chapter we provide an introduction to several aspects that this systems builds upon. In Section 2.1 we provide an introduction to Deep Learning and the training process of a network, and highlight the difficulties that arise when trying to distribute the training process. In Section 2.2 we summarize different approaches and algorithms used to train neural networks in a distributed manner, and describe some systems constituting the state-of-the-art in Section 2.3. After that, in Section 2.4, we turn our attention to the other vital part of the system, serverless computing, and discuss the currently available solutions. We end this chapter by summarizing applications and previous work that merges these two technologies in Section 2.5.

2.1. Deep Learning

Artificial Intelligence has experienced an enormous growth in recent years. The constant drive towards development and innovation in fields like computer vision, speech recognition and natural language generation prompted thorough research in new methods of extracting and treating features, since traditional machine learning methods were not able of achieving good enough performance. This resulted in the rapid development of Deep Learning methods, whose use quickly showed state-of-the-art performance in those tasks in which the extraction of features was not straightforward. After these applications emerged in specific fields, Deep Learning expanded to other areas of work, many times replacing more traditional machine learning methods.

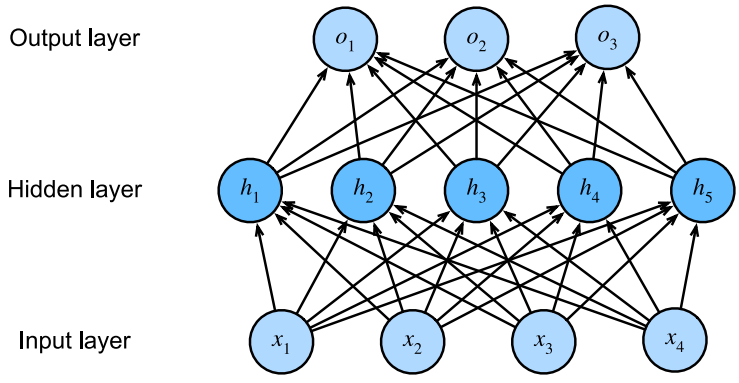


Figure 2.1: Example of a simple neural network with a hidden layer [74]

Deep learning is a subset of machine learning algorithms in which artificial neural networks are used to learn a function fitting the distribution of a certain dataset and match it to a certain output. The name artificial neural networks comes from the resemblance to the neurons present in the human brain. Each neuron is connected to a number of preceding neurons whose output signals are used as input to the next layer of neurons. This process goes on propagating the

activations through the network until an output layer is reached. An example of a neural network architecture is shown in Figure 2.1.

The main advantage of deep learning when compared to more traditional machine learning is that it allows the automation of the feature extraction and selection step. With traditional machine learning algorithms, the raw data had to be treated and features extracted from it using a process called feature extraction. This process was costly and complex, with extra steps often needed such as feature selection, which involved obtaining the right combination of features.

In contrast to this, deep learning takes care of the feature extraction and selection phases for the user. The weights and biases of a network are automatically learned during the training process, to model the function that best maps the input features to the output.

2.1.1. Types of Deep Neural Networks

Depending on their architecture, neural networks can be subdivided into several groups. Here we explain the most famous types of networks.

1. **Feedforward Networks.** Also referred to as Multi-Layer Perceptrons (MLP), these are the simplest type of neural networks. They are composed of an input layer, one or more hidden layers, and and output layer. In this networks, the layers are fully connected, that is, all neuron outputs from a layer are used as input by all the neurons of the following layer.

2. **Convolutional Networks.** Also abbreviated as CNNs, are a specialized kind of MLPs, ideal for processing data which follows a grid-like topology. Some examples of data typically used as input to CNNs are time-series and images. Convolutional layers' weights are referred to as *filters*, one layer can have multiple filters that are applied to an input using a convolution operation, each corresponding to a layer output. These filters are automatically learned during the training and are used to recognize features of the input, such as edges in an image.

   The usage of the convolution operation has certain consequences that differentiate CNNs from feedworward networks:

   - *Sparse interactions*. By making filters smaller than the input data, neurons have only part of the data as input. This allows the network to be more efficient in terms of space, by having to store fewer parameters.
   - *Parameter sharing*. Filters are shared among all the neurons in a layer. That means that instead of each neuron having its own set of weights, neurons in a layer have their weights tied to others.
   - *Equivariant representation*. Due to filters being shared among neurons, a feature in the input will be detected no matter its relative position in the input. This is useful for example in object detection, where no matter the object's location in an image, the network will be able to identify it.

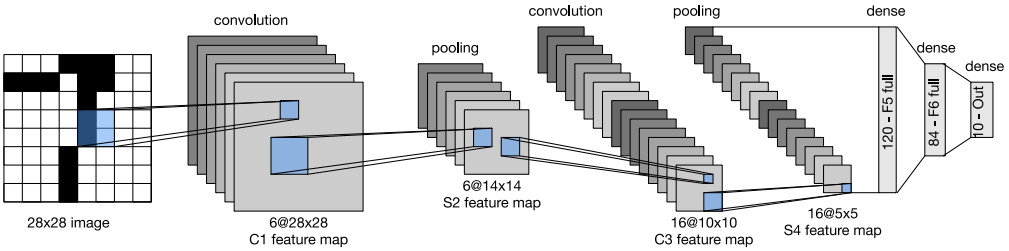


Figure 2.2: Architecture of an example convolutional network, the LeNet5, consisting of two convolutional layers of decreasing filter size, and three fully connected layers [74]

3. **Recurrent Networks.** Similar to the way CNNs share weights across space, RNNs share their weights across time. RNNs incorporate the notion of time into neural networks by having an inner state that they share across timesteps. RNNs are most applied with inputs that have variable length and long term dependencies, such as speech recognition or machine translation.

2.1.2. Training Deep Neural Networks

The process of training neural networks varies depending on the subset of machine learning problems that we are trying to solve. These subsets of problems include Supervised Learning, where we train on labeled data, Semi-supervised Learning, when only a part of the data is labeled, and Unsupervised Learning, when data is not labeled. In this section, we dive into the process of training neural networks, specifically focusing on supervised learning tasks such as classification, since the baselines that we will use to assess the system performance are classification tasks.

During the training of a neural network, the weights and biases of a network are trained with the objective of fitting a function that is able to map the input features to the output labels. Training is an iterative process which can be divided into two phases: the *forward pass* and the *backward pass*.

The Forward Pass

In the forward pass, the inputs $\boldsymbol{x}$ are fed to the network, resulting in an output. To calculate the result of each layer, the input vector is multiplied by the weight matrix formed by the neurons $\boldsymbol{W}$ and adding the bias vector $\boldsymbol{b}$.

$$\boldsymbol{h} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{2.1}$$

However, a network that just performs weighted sums of the inputs can only ever fit linear functions, which are often not enough to model more complex problems. For this reason, neural networks often apply an activation function or *nonlinearity* (σ) to the result of the previous equation. Some examples of frequently used activation functions are ReLu (Rectified Linear Unit), Hyperbolic Tangent or the Sigmoid function.

$$\boldsymbol{h} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.2}$$

Eventually, the forward pass reaches the last layer, which outputs the prediction of the network ($\hat{\boldsymbol{y}}$). In classification tasks, a common output layer is the *softmax* function, which normalizes the output into a probability distribution over the range of possible classes.

The Backward Pass

After obtaining the predictions from the network $\hat{\boldsymbol{y}}$ by performing the forward pass, we need to compare these predictions with the actual labels of the inputs $\boldsymbol{y}$. To do this, we use a loss function $\mathcal{L}$ to determine how erroneous the predictions of our network are, so we can adjust its parameters in the following iterations. Some popular loss functions include the Squared Error, Hinge Loss, Logistic Loss, and the Cross Entropy Loss.

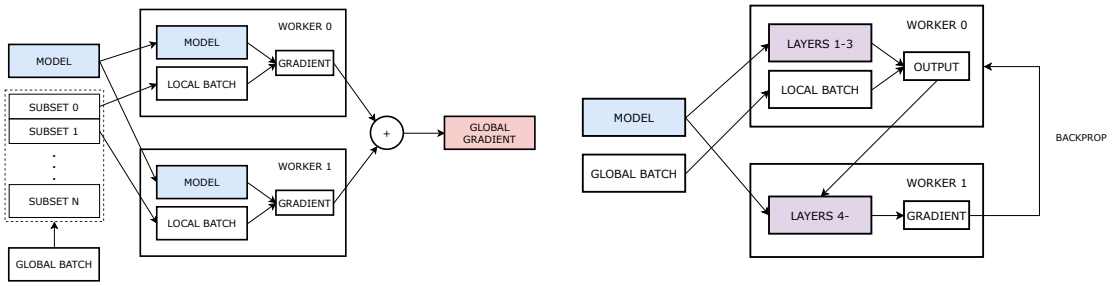
Once the loss is calculated, we perform the actual backward pass by calculating the gradient of the loss with respect to the weights of the network. Performing all the calculations directly is highly inefficient, since a lot of partial results can be reutilized for prior layers. A common way to increase computation efficiency is to use the backpropagation algorithm [59], which uses dynamic programming principles to memorize results and reuse them in later evaluations. After the backward pass, we have the gradients $\mathcal{G}$ of the network, which we will use to optimize the network weights.

$$\mathcal{G} = \frac{\partial \mathcal{L}(\boldsymbol{W}; \boldsymbol{x}, \boldsymbol{y})}{\partial \boldsymbol{W}} \tag{2.3}$$

The most well-known algorithm to train Deep Neural Networks is Stochastic Gradient Descent (SGD). This algorithm sequentially updates the weights of a neural network based on the loss achieved by the model on a set of datapoints. In general, instead of processing one datapoint at a time, many of them are packed together in batches of a constant size, which are fed to the network and used to estimate the loss for that step. This approach is called **Mini-Batch Stochastic Gradient Descent**.

$$\boldsymbol{W}^{(t+1)} = \boldsymbol{W}^{(t)} - \eta \mathcal{G}^{(t)} \tag{2.4}$$

After calculating the loss, the weights of the network are updated following the direction of the loss space in which the gradient becomes lower. The size of this step is determined by another



(a) Data Parallelism

(b) Model Parallelism

Figure 2.3: Comparison between data and model parallelism. In data parallelism each worker has an exact replica of the network, which is trained on a different subset of the data. This makes backpropagation straightforward. In model parallelism, each worker has a subset of the layers of the network, thus need to communicate with each other during the forward and backward pass, increasing latency, but reducing the memory footprint since the model is not stored fully at any worker.

vital parameter of the training process, the learning rate η. This parameter makes sure that the steps taken are big enough that we are able to exit local minima in the loss space, and also small enough that we can explore in detail promising areas of the loss space.

2.2. Distributing the Training Process

As could be seen in the previous section, training neural networks is mostly a sequential process. Inputs are fed into a network to calculate the outputs, based on which gradients are computed and the weights corrected accordingly. However, with and increase in the size of the datasets or their complexity, comes the need for more complex models, built deeper with more layers.

These bigger models and datasets make the training process more expensive. Bigger models include more parameters, which consequently make the forward and backward passes more time consuming. Moreover, bigger datasets make it sometimes infeasible to train a model on a single machine.

All these circumstances prompted the development of distributed machine learning algorithms and platforms, which convert the inherently sequential process of training a neural network into a parallelizable process using multiple GPUs or servers.

2.2.1. Types of Parallelism

There are two main approaches to distribute the training phase of a neural network, distributing the data over multiple devices or distributing the model over several devices, we can see a visual representation of the alternatives in Figure 2.3.

Data Parallelism. This approach divides the dataset in as many subsets as there are workers. Each worker trains a replica of the network on a different subsets of the data. The network is generally replicated in each of the workers and consequently, each model will have a different set of weights after each iteration that need to be merged into a single model in order to maintain a common view of the network. This step varies depending on the optimization algorithm used during training. We will cover the most important ones in Section 2.2.2.

Model Parallelism. In this paradigm, different parts of the model are fit on the complete dataset. The complete model is thus the result of combining all the model parts. This can be done by splitting the layers of the network among workers. This saves up memory, since the full model is not stored in any single worker, however increases the communication overhead, since results from a layer have to be shared with the workers that hold the following layer.

Generally, Data Parallelism is applicable to all problems in which the data follows an independent and identically distribution (i.i.d), while Model Parallelism is only applicable when the model parameters can efficiently be separated among the workers [66]. This makes Data Parallelism more common in distributed deep learning systems.

Additionally, more complex parallelism methods can be applied. Some systems use a mix of model and data parallelism [11], or use Pipeline Parallelism [24, 46] to concurrently process multiple batches of data to increase efficiency and resource usage in model parallelism.

2.2.2. Optimization

Training neural networks in a distributed manner adds extra requirements and complications to the training process which require the use of a specialized training algorithm to merge the model weights found in each of the workers to reach a final model. In this section we cover the main approaches and algorithms used to distribute the training of networks. We focus on algorithms used with Data Parallel approaches since these are the most predominant.

In general, training algorithms will try to replicate the sequential behavior of SGD to a certain extent by periodically synchronizing the model weights between workers. To achieve this, the two main approaches include exchanging gradients or the model weights directly.

When exchanging gradients, workers perform a forward pass of the data, and calculate the gradients of the weights during the backpropagation step. These gradients are then exchanged between workers until all of them have the addition of the gradients of all workers in the system. The mechanisms used to transmit the gradients efficiently will be covered in Section 2.2.3.

On the other hand, workers can instead communicate the weights of their layers rather than the gradients and define the new reference model by combining each of the partial models. The most common way of combining the models is model averaging [54, 55, 78, 34], in which the different layers' weights and biases are added using a weighted average whose coefficients can be all equal or relative to some other parameter.

Distributed Optimization Algorithms

When classifying the different approaches for training deep neural networks distributedly, we can establish two main differentiating factors: (1) the strictness of the synchronization step, and (2) the frequency of the synchronization points.

Using the first criteria we can divide algorithms based on how much freedom is given to workers in terms of not having to wait for others to continue with the next iteration after synchronizing:

1. **Bulk Synchronous Parallel (BSP).** Synchronous algorithms maintain a consistent view of the model throughout the training process by synchronizing between every communication and computation phase. The most prominent example of this type of algorithms is Synchronous-SGD (S-SGD) [8], which synchronizes workers after each iteration. Workers compute the loss and gradients of their local batch of data, and receive the gradients obtained by other workers. These are added and applied to each local model, resulting in all of them using the same weights at the start of each iteration. These algorithms offer better convergence since they try to approximate the sequential training in a single worker, however incur in higher communication overhead since the workers communicate after each batch is processed.

   Moreover, at each sync step, all workers must have finished before the reference model is computed. This means that sync approaches are more sensitive to straggler workers, since the whole computation is stuck until the slowest worker finishes in each step. Some algorithms address this by increasing the number of workers, and only wait for a subset of them to complete before continuing to the next iteration [20].

2. **Stale-Synchronous Parallel (SSP).** Tries to address the sensibility to stragglers of synchronous algorithms by allowing workers to move ahead to the next iteration until reaching a maximum number of epochs. As a result, some of the workers might operate on incomplete or out-of-date parameters. With a small staleness level, these algorithms offer still strong convergence guarantees, however, these guarantees diminish with higher staleness between workers.

   Within this algorithm type we can distinguish different approaches with regards to how staleness is understood. Some algorithms interpret staleness as the maximum number of iterations a fast worker can be ahead of the last of the stragglers [79, 13]. In case this max number is reached, those faster workers need to wait until all of the other workers finish their iterations and the difference returns to the acceptable interval.

Another way of interpreting staleness if the absolute deviation from the value of the local model with respect to the reference model. This approach, however, is less frequently used in deep learning since the huge number of parameters make it difficult to decide which particular variable should define the significance of an update [66].

3. **Asynchronous Parallel (ASP).** Asynchronous solutions completely disregard the synchronization steps and tracking parameter staleness and allow workers to compute and update parameters without waiting for other nodes. An example of this paradigm is Hogwild! [58], which removes the lock preventing concurrent update of the model parameters by multiple workers and allows them to asynchronously access and update the model parameters.

   Another well known example is Elastic Averaging SGD (EASGD) [78]. This method exchanges the model weights instead of the model gradients and allows each of the workers to proceed individually and update its weights after multiple iterations instead of at every iteration. This delay is referred to as τ and helps to reduce communication overhead and balance exploration and exploitation. To update both the local and reference models, workers apply an elastic average on the model weights, hence the name of the method.

   These algorithms do not force a consistent view of the model or even conflict-less parameter updates, thus provide worse convergence properties than sync and stale-sync algorithms. However, in exchange, these algorithms improve throughput, as workers do not need to wait for each other or even lock before updating the model before continuing with the computation. Initially, asynchronous training was considered to be the only performant option, with synchronous methods being considered too slow and sensitive to stragglers [11]. However, recently some studies have quantified the benefits of using asynchronous methods, showing that synchronous methods are more effective for more complex models and problems than asynchronous algorithms [69].

Relaxing Communication

Until now, we have explained algorithms that exchanged model updates in each of the iterations. These parameter exchanges could be performed by waiting for other to complete their computation stage, or asynchronously. However, another axis in which training algorithms can differ is in the period in which these synchronization steps occur. Let us refer to this delay of communication as τ. Varying the value of this parameter can get different algorithms which allow workers to compute for multiple iterations at once between communication steps:

1. $\tau = 1$. This is equivalent to the algorithms explained previously, which synchronize at every iteration, be it synchronously or asynchronously.

2. $\tau = \infty$. Also known as *One-shot Averaging*, averages the models only once after the training process is complete. This effectively minimizes communication since all of the workers train an the data for all epochs and exchange parameters only once. However, these solutions have been proved to show sub-optimal convergence properties, and often result in a bad generalization performance [77, 19].

3. *Variable* τ. This algorithm is usually referred to as Local SGD [80, 81, 63, 41]. The idea behind this algorithm is that by synchronizing every τ iterations, communication overhead is reduced by a factor of τ, but we still have a soft guarantee that the models will be merged periodically into a common view of the parameters, so they will not diverge and hinder convergence. The τ parameter represents a balance between exploration and exploitation. In [80], authors test the convergence of networks under different values of τ, and reach the conclusion that synchronizing after each iteration is not the optimal for some situations, highlighting the benefits of allowing the different workers to explore distinct areas of the loss space.

4. *Ensemble Learning.* Similar to the One-Shot Averaging technique explained previously, ensemble methods minimize communication by having workers train for the entire duration of the process without synchronizing models. However, unlike One-Shot Averaging, ensemble learning does not average the models to obtain a reference model. Instead, ensemble

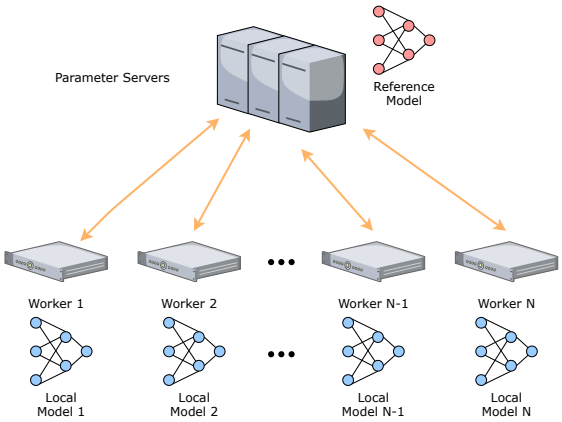


Figure 2.4: An example of the PS architecture. Workers have their own model which they train on a subset of the data, and periodically synchronize with the parameter servers by pushing/pulling updates. This synchronization can be either synchronous, asynchronous or stale-synchronous

learning methods keep the different models disjoint, and at prediction time, the outputs of all model replicas are combined by a voting strategy, which defines the output of the whole ensemble.

2.2.3. Communication

In the previous section we explored the different algorithms available for distributed deep learning from the point of view of how often synchronization occurs or how strict they are with workers having to wait for each other. We did not explore, however, how the model weights or gradients are aggregated and how the new reference model is computed. In this section, we go through the main approaches for combining the model weights into the reference model.

Parameter Server

In the Parameter Server (PS) architecture [11, 40], some servers in the cluster hold the parameters of the reference model, while the rest, or *workers* perform the training process and periodically synchronize their parameters with the master servers. When a synchronization step is reached, workers push the weights or gradients to the parameter servers, which are responsible for aggregating them and calculating the new reference model.

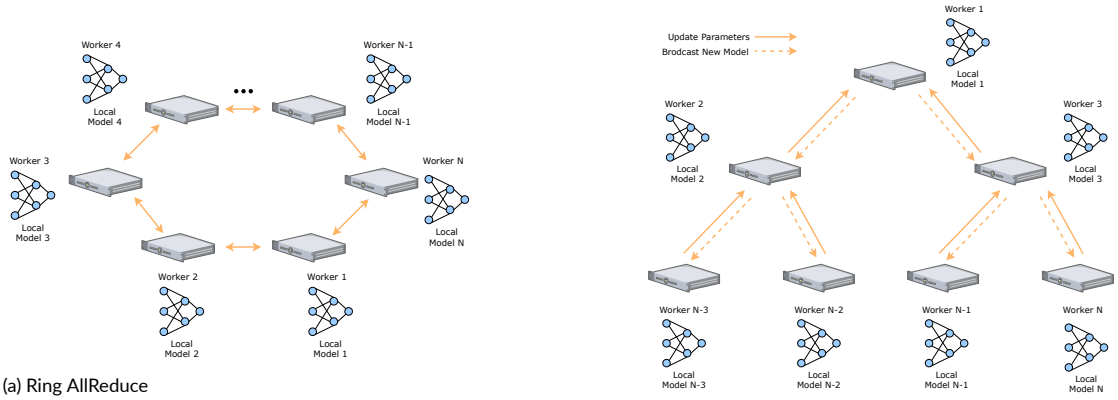
In this architecture, the parameter servers act as the epicenter of the topology, with whom all workers communicate to synchronize and send their updates. This makes the PS architecture a centralized architecture. Despite this, the PS architecture is widely used in distributed training, especially when performing multi-node training [39, 38, 23, 49].

A common problem with the PS architecture is that it suffers from communication congestion [64]. Especially with BSP algorithms, communication tends to occur in very concentrated intervals at the time when the workers complete their iterations. This is alleviated in SSP and ASP algorithms since workers progress more unequally, so their communication periods with the server do not have to coincide to the same degree.

AllReduce

AllReduce algorithms try to overcome the limitations introduced by a centralized server by efficiently aggregating the gradients or weights by having workers communicate directly with each other. To allow for latency and bandwidth efficient communication, workers are usually structured following predetermined topologies such as trees or rings. AllReduce algorithms are inspired by the popular directive in libraries such as Message Passing Interface (MPI) [43].

Ring AllReduce organizes the workers following a ring-like topology. In this way, workers only need to exchange parameters with their neighbors. These algorithms are bandwidth optimal, however its latency scales linearly with the number of workers [64], so it is mainly used with a small number of workers. An example of this is training on multiple GPUs within a single node.



(a) Ring AllReduce

(b) Tree AllReduce

Figure 2.5: Comparison between data and model parallelism. In data parallelism each worker has an exact replica of the network, which is trained on a different subset of the data. This makes backpropagation straightforward. In model parallelism, each worker has a subset of the layers of the network, thus need to communicate with each other during the forward and backward pass, increasing latency, but reducing the memory footprint since the model is not stored fully at any worker.

Nvidia's high performance communication library NCCL[1] used rings as their main topology until version 2.3[2].

In *Tree AllReduce* workers are organized following a binary tree structure. In this way, they only need to receive the partial gradients or weights from their descendants, and communicate the updated parameters to its parent. Thus, a single sync step involves a propagation up to the root of the tree and back to the leaves. Tree Allreduce algorithms scale better with a bigger amount or workers in terms of latency, and are the standard in NCCL since version 2.4.

AllReduce algorithms, being inherently synchronous in their data exchange step, are not portable to ASP and SSP algorithms, which is one of the reasons for the prevalence of the PS paradigm especially in multi-node training, since it provides more flexibility in terms of available distributed communication algorithms. Another important factor is that AllReduce algorithms are very sensitive to worker failures [66].

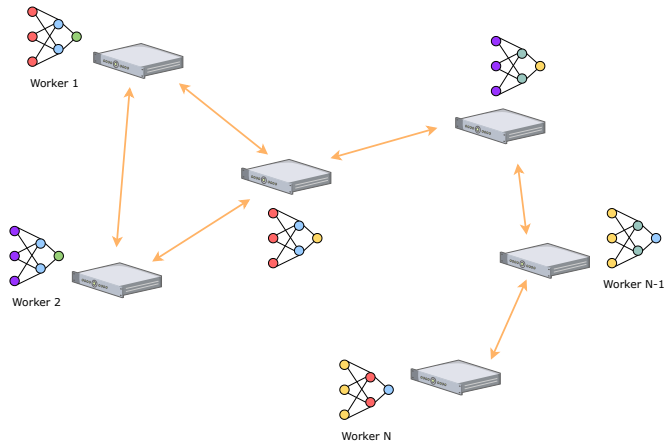


Figure 2.6: An example of Gossip Learning. Each worker periodically exchanges information about its model with its neighboring nodes. Notice that in this architecture, nodes can have very different models, and only eventually reach convergence into a common view

[1] https://developer.nvidia.com/nccl
[2] https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/

Gossip

Gossip algorithms are mainly used in peer-to-peer decentralized settings. A main characteristic of these environments is that unlike in the previously explained methods, there is no reference model, since there is no strict synchronization and workers can proceed independently. Workers exchange their model weights with their peers or neighbors, and these updates eventually reach the entirety of the network after subsequent communication between neighbors.

2.3. Distributed Deep Learning Systems

The increasing model and dataset size has prompted the development of Deep Learning systems that allow to distribute the training process among multiple workers in an easy and efficient way. In this section, we take a look at the most well-known distributed deep learning systems and their key characteristics.

2.3.1. DistBelief

DistBelief [11] was one of the precursors of the distributed deep learning field, built by Google. DistBelief tries to tackle the problem of models not fitting into GPU memory, and needing to split them among multiple GPUs or machines efficiently to enable training. Originally, Distbelief only supported training on CPU, but GPU support was later added to improve performance.

DistBelief represents the network and its operations as a directed acyclic graph (DAG), where each edge represents an N-dimensional matrix, and each vertex performs a transformation on the inputs which defines the outputs. To distribute the computation, DistBelief relies on model parallelism. This results in different nodes calculating the outputs of different parts of the network. In consequence, network size stops being a key limitation since no single node needs to store all of the weights in memory. On top of model parallelism, DistBelief also incorporates data parallelism using parameter servers to store the model weights during training, these are sharded to increase throughput.

To train the networks, DistBelief includes implementations for a couple of asynchronous algorithms, since at the time synchronous algorithms were deemed too sensitive to stragglers to be used at large scale in production [1]. The authors propose two asynchronous algorithms and compare its performance:

- **Downpour Stochastic Gradient Descent** is an asynchronous algorithm, in which each replica of the model iteratively fetches the most recent parameters of the model, trains on batches of the data, and pushes its updated parameters to the parameter servers. Models do not need to perform the synchronization of parameters at every iterations, but can be given some leeway by setting the n_{fetch} and n_{push} variables, which define the number of iterations before fetching or pushing new parameters. These quantities do not have to be equal. Notice that this is in fact a method of relaxing the communication frequency as we studied already in Section 2.2.3.

  This method makes the training process more resilient to stragglers or even failures, since in the case of failure, the remaining workers can keep training and updating the weights without having to wait for each other, unlike in synchronous SGD methods. Additionally, authors affirm that using Adagrad [12] to tune the learning rate to be used on different parameters to improve convergence performance.

- **Sandblaster L-BGFS** uses an extra coordinator process which assigns work to model replicas and instructs parameter servers on which operations to perform. As implied in the name, rather than using SGD, L-BGFS is used as a training algorithm. The use of a coordinator helps reduce the impact of straggling workers by assigning more work to faster workers once they are finished, so that fast workers end up processing more data than slower workers. This method, however, does incur in considerably higher computational needs, making it less effective under a limited number of CPUs.

2.3.2. TensorFlow

TensorFlow [1] is the evolution of DistBelief, also developed by Google. DistBelief enabled deep learning applications to be distributed among multiple workers, but lacked extensibility and porta-

bility. TensorFlow borrows the dataflow abstraction from DistBelief, also representing neural networks as a DAG whose edges carry tensors or N-dimensional matrices, and its vertices perform mathematical operations on them. It also uses the parameter server paradigm to synchronize the model weights between workers. However, in TensorFlow parameter servers are not actually servers, but a special kind of *task*, just as any other worker process, making deployment easier than in DistBelief.

TensorFlow allows for easily defining new layers and algorithms, thus making experimentation with new training techniques feasible. Moreover, operations on tensors are implemented for different computational backends, which permits running TensorFlow models in a variety of devices such as CPUs, GPUs, TPUs and mobile devices with the same code. The authors also consider synchronous algorithms apart from the asynchronous options explained in DistBelief, stating that with proper optimizations, synchronous training can be faster than asynchronous techniques.

2.3.3. MXNet

In a similar fashion to TensorFlow, MXNet [10] enables distributed deep learning using a dataflow abstraction to represent models, and a parameter server architecture for parameter exchange. Rather than committing to a declarative approach for defining tensor operations, MXNet allows users to define computations in either a declarative or imperative fashion, more along the lines of other libraries like PyTorch [52]. MXNet also implements its own parameter server: `KVStore`, which implements a classic key-value store API. Users can *push* new parameters and *pull* updates. KVStore takes care of updating the parameters according to an update strategy defined by the user.

MXNet's parameter server implements additional optimizations when compared to traditional PS architectures. Instead of having one layer of parameter servers to which all the workers send their parameters, the authors propose a layered architecture. In each machine, a local server takes care of synchronization between all the local models, while a second layer of parameter servers communicate with the local parameter servers at a cluster-wide scale.

2.3.4. Caffe

Caffe [27] was initially developed within UC Berkeley as a deep learning framework for training deep models in a single machine on one or multiple CPUs or GPUs. Caffe's abstractions remind of the ones in TensorFlow and MXNet. Networks have two main components: *blobs* and *layers*. Blobs store any binary data needed to perform the forward and backward passes, such as images, parameters, or intermediate results between layers. Layers on the other hand, apply operations on these blobs to transform them to obtain an output blob. Caffe provides by default implementations for widely used layers such as convolutional, pooling and nonlinearities.

Caffe's limitations prompted the development of its evolution, Caffe2 [5], which allows for distributed training past a single machine, and incorporates support for additional hardware devices. To synchronize models between worker processes, Caffe2 uses AllReduce algorithms using Nvidia NCCL between GPUs in a single machine, and custom code employing Facebook's Gloo [15] collective communication library between workers in different machines [66].

From among the AllReduce algorithms (see §2.2.3), Caffe2 chooses Ring AllReduce due to its higher bandwidth efficiency [66], this does however, make Caffe2 not tolerant to failures, since a crash of a single worker would cause the ring information to be incomplete, hence impeding synchronization.

2.3.5. Horovod

Horovod [60] is an extension of TensorFlow developed by Uber to address the suboptimal scaling of TensorFlow when radically increasing the number of GPUs. Authors show that using TensorFlow's default synchronization strategy, with some networks like ResNet-101, communication overhead became dominant over computation, causing more than half of the resources not being utilized during training.

To address this, the author's implement a Ring AllReduce algorithm using Nvidia NCCL to replace TensorFlow's communication strategy, using traditional high performance computing tools like MPI [43] to run multiple copies of the TensorFlow program in parallel. Though it increases average GPU utilization throughout a variety of models, by using AllReduce instead of a parameter

server, Horovod becomes intolerant to failures.

2.4. Serverless Computing

In this section, we provide an introduction to Serverless Computing, what its main characteristics and benefits are, as well as it shortcomings.

2.4.1. What is Serverless?

Serverless computing, a term also frequently used interchangeably with *Function-as-a-Service* (FaaS) despite subtle differences [18], is a recently popularized computing paradigm which tries to simplify cloud application development by abstracting away the underlying complexities of deployment and provisioning, and allowing users to focus only on application logic. FaaS applications comprise one or several *functions* that are configured to run after a request is made to a configured trigger. The main benefits of cloud systems designed on top of FaaS include effectively unlimited autoscaling, with more functions being deployed automatically to compensate for an increase in requests, and pay-as-you-use pricing.

Traditionally, to account for variability in the volume of requests and avoid saturation of services users and companies alike had to resort to over-provisioning of resources. Most of these resources were not taken advantage of most of the time, and thus resulted in wasted resources both computationally and economically. FaaS platforms provide a fully managed operational logic by the cloud provider, hiding the deployment details from the user, such as idle virtual machine (VM) management, slow start, VM placement, network routing, provisioning etc. Apart from the operational benefits, FaaS platforms also introduce a brand new payment model compared to traditional cloud platforms. Instead of paying a fixed price for the allocation of a traditional virtual machines, FaaS resources are billed by function utilization, usually by invocation or at a millisecond granularity, meaning that if a function is not invoked, no amount is charged to the owner [18, 25].

This programming model is clearly advantageous for conveniently/embarrassingly parallel applications. In such applications, different functions can proceed in parallel without having mutual data dependencies that require communication and coordination between the functions, such as map-reduce or batch jobs [22]. Some libraries like PyWren [29] were born to facilitate the use of serverless functions for these parallel jobs.

While simple and elegant, the model imposes noticeable limitations when developing applications with more strict communication needs [22]. For starters, serverless functions have most of the time a **limited lifetime**. In cloud platforms' serverless offers, a running function will be suddenly stopped after exceeding this lifetime, resulting in a loss of data that was not saved to external storage. This makes long-running applications complex to implement on top of FaaS, since the internal state of running functions has to be periodically stored to prevent information loss, which also introduces latency into the application.

Serverless functions are **not directly addressable** and cannot be in a server role for communication which in practice means that functions cannot communicate with each other without an intermediate system to exchange information and/or synchronize execution. This introduces the need for external elements to enable communication between functions. Over the years, storage platforms like S3 [67], key-value stores like Redis [56, 28], or application-specific parameter servers [7] have been used in the past to facilitate inter-function communication. However, accessing these external components must be done through the network, which introduces extra latency in the application.

Looking back at the communication algorithms we described in Section 2.2.3, this restricts the usage of AllReduce algorithms since functions do not have access to each other's network address, and thus cannot transfer their model parameters efficiently to each other. This is the reason for most serverless ML platforms to recur to external storage to synchronize between functions. Some works [28] try to replicate AllReduce in a serverless environment, but do so by adapting it to use external storage to deposit the weights from functions instead of transmitting them directly.

An especially sensible feature for deep learning tasks is the usage of special hardware to accelerate the training process of neural networks. Alas, current cloud solutions do not offer any

other resources apart from CPUs, making the training of bigger networks very time consuming when compared to using GPUs or TPUs. This makes many serverless ML solutions lack financial benefits, since training on multiple CPU lambdas takes considerably longer to finish than using a single GPU server, and often results more expensive [28], which limits the applicability of FaaS to this kind of problems.

2.4.2. Current Solutions

Cloud Provider Options

Most of the time, FaaS is directly associated with serverless options offered by cloud providers such as AWS, Google Cloud and Azure, with AWS Lambda being the leading platform for developing FaaS applications [14]. These options are fully managed by cloud platforms, users just need to develop the function code and set events that will trigger the execution of those functions, while the rest of the complexities of managing the infrastructure are completely hidden from them.

In [14] the authors study the usage of different FaaS options and rank them by popularity based on platform. According to their findings, AWS Lambda is used in 80% of FaaS deployment, with Azure Functions comprising just 10% of use cases, and Google Cloud Functions just 3% of the overall solutions.

AWS Lambda Amazon released AWS Lambda[3] in 2014, being the first FaaS system openly available to the public and with time maintaining its position as the leader of the sector.

Lambdas are stateless functions with a limited lifetime, which can be activated using a variety of configurable *triggers*. Some of these include HTTP triggers from an API Gateway, file uploads to S3 buckets, or events from streaming platforms like AWS Kinesis. AWS lambda also offers the widest array of available programming languages or *runtimes* to write functions, including JavaScript, Go, Python, Ruby, Java and C#. Moreover, custom runtimes can be written by users to allow writing functions in extra languages.

Azure Functions Azure Functions[4] is Microsoft's FaaS solution and was introduced in 2016. Unlike AWS Lambda, the code of Azure Functions is open source, and allows for local deployment using Kubernetes. Azure also offers an extension to its traditional serverless functions in the shape of stateful functions called Durable Functions. Azure Functions also offer a variety of languages to run functions with, namely C#, JavaScript, Java and Python, but still does not reach the level of AWS in terms of allowed programming languages.

Google Cloud Functions Google released its own Function-as-a-Service platform in 2016[5] to compete with AWS and Azure, and is as of today the least adopted of the three [14]. When compared to solutions like AWS lambda, a clear shortcoming is the number of languages available to write functions, limited to JavaScript, Python and Go.

Another important factor to keep in mind when addressing the convenience of FaaS is the pricing and the payment scheme. All of the studied cloud platforms follow the same pricing structure, with two sources of cost for the user: price per invocation and price per duration. Price per invocation is the most straightforward, based on the number of functions running, a cost is charged to the user. Current rates are shown in Table 2.1, where the price indicated there shows price per million invocations. Even though Google is more expensive, it offers a more extensive free plan in which the first 400K calls are not charged to the user, therefore reducing the price for the first million invocations, but still remaining more expensive for the consequent calls. The other factor in pricing is related to the memory size of the function, and is charged for the execution time of the function, normally calculated at millisecond granularity. Here we see a comparable offer by both AWS and Azure, with Google offering a much lower price per duration.

All providers offer configurable parameters of functions, with the main parameter being memory. In most cases, the number of CPU cycles assigned to each function is scaled according to

[3] https://aws.amazon.com/lambda/
[4] https://azure.microsoft.com/services/functions/
[5] https://cloud.google.com/functions

			Price	
Platform	**Memory**	**Max. Duration**	**Invocation***	**Duration (GB/s)**
AWS Lambda	128MB - 10GB	15 min	$0.2	$1.67e^{-5}$
Azure Functions	1.5GB**	10 min	$0.169	$1.6e^{-5}$
Google Functions	128MB - 8GB	9 min	$0.4	$2.5e^{-6}$

Table 2.1: Comparison of features of the main Cloud Function Providers
* Price is given per million invocations
** With a Premium plan, memory can be increased until 14GB

the assigned memory, but the function used to determine the dedicated CPU is not clear in some platforms like AWS Lambda. With the goal of accommodating more complex and expensive workloads on serverless, we have seen an increase in the allowed limits both in memory size and duration of functions. AWS increased the maximum amount of memory from 3GB to 10GB recently, as well as increased the maximum duration from 10 to 15 minutes. This is greatly beneficial for running tasks that before were not as good of a fit on serverless, such as machine learning.

Open Source Options

Despite the convenience of cloud provider FaaS solutions, they suffer from issues previously discussed such as limited lifetime and the inevitable vendor lock-in. With this in mind, open source serverless solutions were proposed which, although slightly increasing the operational burden, allow for an increase in flexibility and configurability. These solutions run on top of Kubernetes[6] and implement the serverless paradigm using traditional Kubernetes abstractions such as Pods, Deployments and ReplicaSets.

In this section we will briefly explain the main concept behind Kubernetes and its main abstractions, before diving into the multiple options available to run serverless platforms on Kubernetes, as well as their characteristics and architectures.

Kubernetes Virtualization has played a key part in the development of Cloud Computing, improving efficiency and flexibility over traditional bare-metal servers [70]. Virtualization allows providers to abstract the user's view from the physical machines, and enables shared usage of resources such as CPUs, memory and disk, thus increasing efficiency. With time, virtualization techniques have also evolved and have experienced the birth of containers, providing a smaller and more lightweight environment for running applications than full-fledged VMs.

Containers are also tightly related to the concept of microservice architectures, which try to split an application into its core functionalities, each running as an almost-stateless service. These services communicate with other through events and messages [31]. Containers provide isolation in the form of control groups (`cgroups`) and namespaces, which restrict the kernel features accessible to the user, and the access to system resources respectively. But these microservice applications are composed of hundreds or thousands of instances that need to be managed in terms of scheduling, networking, fault tolerance, monitoring, etc. To address this, Container Orchestration Platforms were created, which automate the deployment and management of containerized applications.

Kubernetes is an open-source container orchestration system, developed by Google, which automates deployment, scaling and management of containerized applications. Kubernetes follows a master-worker architecture, with one server in charge of scheduling tasks and managing resources, and multiple workers that periodically update their state according to the instructions from the master. Instead of determining the low-level application properties such as assignation from containers to servers, Kubernetes follows a declarative approach. The user defines the desired state of the cluster, which is then transparently maintained by the control plane and the master.

Kubernetes offers a wide range of resources and abstractions to build distributed applications, which are defined in the YAML modeling language:

- **Namespaces** define divisions between different applications or workspaces

[6] `http://kubernetes.io`

- **Pods** are the smallest units within clusters. A Pod comprises one or multiple tightly-coupled containers, which share resources such as network and storage.
- **ReplicaSets** are groups of pods all performing the same task, deployed in parallel to increase availability.
- **Deployments** specify the desired state of Pods by creating a ReplicaSet. Deployments also allow to define autoscaling policies for Pods.
- **Services** expose a set of Pods or Deployments as a network service. Services can be configured to expose Pods as a public service through a LoadBalancer or an internal IP within the cluster.

Using these abstractions, open-source serverless platforms are able to orchestrate containers as functions in cloud provider platforms, by allowing containers to execute custom code written by the user. Generally, these solutions allow users to set limits on function resources themselves, instead of having to adhere to enforced limited by cloud providers. Parameters like CPU, memory, duration, and parallelism limits can be tuned or even omitted altogether. Below we list the main options currently available and cover the basics of their architecture.

OpenFaaS OpenFaas[7] follows the same basic structure as AWS Lambda, using a main component serving as the API Gateway performing the fundamental tasks such as exposing an API, monitoring metrics, handling scaling operations and receiving requests to trigger functions. Scaling can be configured to multiple factors such as requests per second or CPU usage, which are monitored through Prometheus and periodically scraped by the Gateway.

Functions are deployed inside Docker containers wrapping the code of the user, which is then triggered with HTTP requests. The process of creating a function involves writing the function code but also a configuration YAML file describing the function configuration, this adds an extra step that is not usual in cloud provider platforms. Function code is wrapped in a programming language-specific docker container, whose details are abstracted away from the user[8].

Kubeless Kubeless[9] follows a similar design choice, using a Controller as the main component handling the API exposure and requests. However, unlike in OpenFaaS, function requests can be configured with different ingress types, either HTTP or events from message brokers like Kafka or NATS. When using HTTP Triggers, functions are not hidden behind a gateway as in OpenFaaS, but are directly exposed using a Kubernetes Service.

In terms of writing functions, Kubeless offers several different *runtimes*, which are Docker containers holding the needed dependencies to write applications in a certain language. Unlike in OpenFaaS, users do not explicitly need to write configuration files, but can simply provide the code and the chosen runtime to be used alongside it. The function code is embedded in the runtime container, and will be run upon a request arrival.

OpenWhisk Apache OpenWhisk[10] is developed by IBM, and follows the same basic structure as previous systems. Its main abstractions are called *actions*, *triggers* and *rules*. An *action* is the name a function is given in OpenWhisk. *Triggers* are endpoints that receive requests and based on the configured *rules*, are connected to one or multiple *actions*. This means that a trigger can prompt the execution of one or more actions, the same way that an action can be launched by multiple triggers.

For internal communication in OpenWhisk, we see a different strategy to other systems. Rather than using plain HTTP, OpenWhisk relies on Apache Kafka for transmitting messages and information to *actions*. Two components act as start and finish points of the invoking process. The *Controller* receives the requests and sends them through Kafka to the *Invoker*. The *Invoker* is

[7] http://openfaas.com
[8] https://docs.openfaas.com/tutorials/first-python-function/
[9] https://kubeless.io/
[10] https://openwhisk.apache.org/

then responsible from taking decisions like spinning a new Docker container to serve the request, or reuse a previous one and avoid cold start.

As with Kubeless, *actions* use runtimes for different programming languages, which are extensible Docker containers containing the user code. This code is embedded in the container at creation time, when the user deploys it using the Whisk CLI.

Fission. Fission[11] provides a more distributed architecture, with the principal components in other systems being divided into smaller services with a single task, following more a microservice design approach. In line with similar systems, the Fission API is exposed by the *controller*, which allows users to create, delete or modify resources. *Triggers* can be configured in the shape of HTTP endpoints that result in the execution of a function. These triggers are registered in another component called the *router*. Upon request, the router queries the function address from the *executor*, which is in charge of creating or deleting function pods according to the configured limits.

To write functions, Fission provides a variety of *environments*, equivalent to runtimes in other systems. These environments hold the dependencies and the function code. A differentiating factor is that environments are not strictly tied to the function code at creation time, instead, the code loading is performed dynamically at runtime. Each function pod holds two containers, the environment we already described and a *fetcher* container. When deployed, the environment is nothing more than a default Docker container with a REST API, but once the pod receives a request, the container is *specialized* by the fetcher, and provided the code it needs to execute. This feature allows multiple functions to run using the same environment, instead of having specialized runtimes for each function.

2.5. Combining Deep Learning & Serverless

This section discusses the previous work about using serverless for large-scale data treatment and machine learning. We discuss data processing frameworks created to exploit the serverless architecture in Section 2.5.1; in Section 2.5.2 we present proposed systems that deal with running ML workflows using serverless. In Section 2.5.3 we summarize the literature discussing the incorporation of GPUs to serverless applications, and in Section 2.5.4 we explore previous work using Kubernetes to to accommodate deep learning tasks.

2.5.1. Serverless Data Processing

As we briefly mentioned in Section 2.4, serverless is an especially good fit for highly parallel applications in which different functions can move forward without having to periodically communicate with each other. Out of the box, cloud platforms offer no universal way of orchestrating functions and combining their output, which limits their applicability to cases such as map-reduce jobs.

There have been some platforms proposed to take advantage of serverless functions as a way to parallelize data processing workflows, with all of them using AWS Lambda as their base platform of choice. PyWren [29] is the main platform used in in literature to coordinate lambdas and easily parallelize local code. PyWren is installed as a Python library and offers a high level API for defining and executing lambda functions. The local device acts as a coordinator of the remote functions, uploading the function code to S3 and invoking lambda functions in parallel executing the same code. Results are output to S3 again, which can be conveniently fetched from the local device. PyWren implements a generalist solution for code execution in lambda functions, so extensions were proposed improving PyWren's performance in particular tasks. Numpywren [61] extends PyWren with optimizations for linear algebra operations, exposing a numpy-like interface for users which can execute numerical workflows with ease. For storage, it also uses S3.

In [56], authors question whether the usage of S3 for all storage during analytics workloads is indeed the most efficient in terms of runtime and cost. They study the performance of shuffle operations in map-reduce jobs running on AWS lambda and the compromises between using slow storage (S3) or fast storage (Redis) deployed on AWS Elasticache as the intermediate storage between stages. Redis offers superior throughput but is almost two orders of magnitude more

[11] `http://fission.io`

expensive, so authors propose the use of Locus, a serverless platform extending PyWren, able to perform a hybrid shuffle using a model to optimize which storage to use to perform communication stages to provide a balance between speed and cost.

2.5.2. Serverless Machine Learning

With serverless only increasing in popularity in recent years, limited work has been done on implementing machine learning workflows using serverless architectures. The main use of serverless in machine learning was at first exploiting serverless' convenient autoscaling and pricing to perform model serving. In these solutions, a previously trained model is exposed behind serverless functions, so that when a certain model is triggered, the function loads the model and the new input datapoints and returns the predictions from the model [26, 2, 3].

In 2018 two papers describe their vision for what running machine learning training tasks on serverless could need to succeed and what the main limitations were at the moment. Carreira et al. [6] describe the main limitations of serverless architectures slowing their applicability. They state that the low memory limits and limited lifetime of functions is a major disadvantage when deploying ML tasks. At the time of writing, lambda functions allowed for a maximum of 3GB of memory and 512MB of disk, and could take up to 2 minutes to start. They list the main challenges going forward as the lack of GPU support and fast intermediate storage to replace S3 and provide performance gains. Feng et al. [16] mentioned that up to their writing efforts there had not been any proposed platform for neural network training using serverless, and envisioned what a possible architecture would be for this problem. They propose two alternative pure-FaaS architectures. One with a single parameter server that is also a FaaS function, and another mimicking tree AllReduce, with workers later acting as parameter servers for each other after completing computation. They do however, not detail how the functions communicate with each other or which storage platform they choose for data exchange.

Cirrus [7] materializes the serverless ML system proposed in [6], and presents a system that uses AWS Lambda to parallelize tasks performed during a variety of stages in the ML pipeline, from data preprocessing and model training to hyperparameter tuning. They build this solution by extending PyWren and opt for a hybrid solution mixing FaaS and traditional VMs. They use S3 for data storage but build a custom-made parameter server to handle model updates during training from the models. They test their solution on a variety of ML tasks using Stochastic Gradient Descent, but do not include deep learning tasks in their experimentation.

A couple of articles tackle the problem of running deep learning tasks on AWS Lambda. Siren [67] also builds on PyWren and opts for a pure-FaaS approach, using S3 for all storage both for data and for intermediate outputs between epochs. Rather than focusing on the actual performance of the platform, the authors place their main focus on accurately predicting the performance on tasks and, using reinforcement learning, choose the optimal parameters for the task (size of lambda function, number of workers) to balance response time and cost. A similar approach is taken by Xu et al. [71], who propose λDNN. Authors do not highlight the convenience of deploying the functions —handled by PyWren in other proposals— and fully concentrate on performance modeling and optimization. Architecture-wise, they use a hybrid approach with a traditional VM acting as a parameter server for functions, and use the ZeroMQ messaging library to communicate the model parameters. They test their system using deep learning tasks and TensorFlow, and compare their resource allocation and prediction to previous approaches such as Siren and Cirrus, improving on both.

Most recently, Jiang et al. [28] proposed LambdaML, a serverless ML platform that enables the comparison between pure-FaaS ML systems and hybrid approaches including traditional VMs. LambdaML allows for performing multiple ML tasks from Logistic Regression to Deep Learning, implemented with PyTorch. It also allows for configurable optimization algorithms including SGD and model averaging. Authors also evaluate the benefits of using different platforms for intermediate storage between functions such as Memcached, S3, Redis and DynamoDB. Focusing on deep learning, the evaluation of the system shows that for small models (MobileNet), using a single GPU instance clearly outperforms all FaaS configurations, reporting 8x faster and 9.5x cheaper performance. This further highlights the need for GPUs in serverless ML to breach this performance and cost gap and make serverless usable and affordable in ML.

2.5.3. Usage of GPU in Serverless

As we have previously described, GPU utilization in serverless is a vital aspect to be implemented in order to favor the convenience of serverless ML and put in on par with Infrastructure-as-a-Service (IaaS) platforms. Alas, limited work exists that is able to exploit these resources from serverless. Given that proprietary cloud platforms do not offer the functionality of incorporating custom hardware like GPUs or TPUs to the runtime, to expose GPU resources, the authors needed to resort to open-source alternatives.

Kim et al. [32] used the open-source platform IronFunctions[12] and Nvidia-Docker[13] to allow serverless containers to access GPU resources. The system works using HTTP requests; when a new request to execute a GPU workload, a new Docker container is started on demand and the function code executed. The authors test their system on a variety of containers, and in terms of deep learning run tasks on TensorFlow, Theano and PyTorch. Despite this work being the first that combines serverless and GPU, it does so by running one workload per container, so it cannot be classified as a distributed machine learning solution.

Naranjo et al. [45] present the OSCAR platform, designed to expose GPU resources in remote servers and allow users to schedule workloads on those GPUs from their local machines as if the GPUs where indeed installed locally. From an architectural perspective, OSCAR uses Kubernetes as an orchestration system, and the open-source OpenFaaS platform as a serverless alternative to run on Kubernetes. Apart from that, they employ rCUDA[14] in their remote GPU server to virtualize GPU resources and provide isolated access to them. They evaluate the system using Tensorflow for image classification, but their experimentation is limited to inference tasks. Nevertheless, the authors argue that model training could also be a future use case of the system since it should benefit from GPU sharing between functions and an increased parallelism.

2.5.4. Deep Learning on Kubernetes

Deep Learning jobs are commonly run on a single machine or cluster made up of VMs, which fix the allocation of resources from the start to the end of the job runtime. To try to make this process more flexible, there has been recent research on scheduling deep learning jobs on Kubernetes, since its design allows for easy allocation and migration of containers and processes throughout the training process. Optimus [53] is a scheduler specifically designed to maximize performance and efficiency of Deep Learning jobs on Kubernetes. The authors envision a deep learning (DL) cluster to which jobs arrive in an online manner, and Optimus is in charge of resizing and migrating containers belonging to different DL jobs in order to accommodate newcomer tasks while still maintaining performance. The authors' focus is on MXNet, a deep learning framework built with the Parameter Server paradigm, in which separate processes behave as workers performing the bulk of the computation, while others are parameter servers (PS) who aggregate the parameters. They design a scheduler which maximizes performance by allocating worker and PS pods in the same servers to minimize response time.

In a similar environment, Mao et al. propose SpeCon [42], a Kubernetes scheduler tailored to deep learning workflows which approximates the convergence of jobs and migrates tasks close to convergence to leave space for more resource-hungry jobs. Converged jobs do not benefit as much from extra computing cycles, so they are migrated together to avoid their interference on newer jobs which would benefit more from added CPU cycles.

A more interesting system due to its relation with our own KubeML is Kube-Knots [65], which focuses on the drawbacks of GPU allocation in Kubernetes clusters. In line with our experience, the authors state that unlike CPU and memory, which can be scheduled and guaranteed with ease by clusters, GPU resources are not properly modeled or scheduled by Kubernetes. On top of that, virtualization of GPU resources is currently not supported in general by cloud providers.

Because GPU resources are exposed using PCIe passthrough, bypassing the hypervisor, no fine-grained control can be performed by the scheduler of limits and requests for GPUs. This results in Kubernetes treating GPUs as exclusive resources, and allocating them statically to a single pod. This often leads to GPU underutilization, which is not a desired outcome given the

[12] https://open.iron.io
[13] https://github.com/NVIDIA/nvidia-docker
[14] http://www.rcuda.net

cost of these resources. Kube-Knots modifies the Nvidia k8s-device-plugin[15] used for exposing GPUs to Kubernetes in order to allow GPU sharing between multiple pods. GPUs are time-shared in terms of compute, and space-shared in terms of memory. Kube-Knots uses a bin-packing algorithm and real-time monitoring to safely schedule multiple pods per GPU which can coexist without causing crashes due to excessive allocation of resources.

Complimentary to this work on GPU sharing on Kubernetes, Yeung et al. [72] propose a model able to predict GPU usage by deep learning models, which helps to decide which models can be co-located in the same GPU in order to increase utilization without causing errors and slowdown as a result of over-allocation.

2.6. Conclusion

In this chapter, we have explained the basics of deep learning and how we can translate the sequential process of training a deep neural network into a distributed setting using different communication methods and architectures. We have also introduced the concept of serverless computing and analyzed its benefits and shortcomings. This has lead us to discussing possibilities of merging both technologies, enabling elastic neural network training on serverless architectures, and we presented current applications making use of serverless and deep learning in conjunction.

As shown by the currently available literature, one of the main shortcomings preventing FaaS from evolving into a real alternative for training ML models is the lack of GPU support. This obstacle points us in the direction of open source serverless platforms. However, these platforms have a higher operational cost for users, since apart from defining the logic of their ML code, they have to still allocate a cluster and configure its components. For this reason, another objective to be fulfilled in our work should be to breach this gap between cloud and open-source alternatives, providing an easy to use interface to deploy functions and start training without the need for low-level knowledge of cluster management and the underlying system.

Another clear idea to exploit is the one proposed in [45]. GPU resources are often under-utilized when training deep learning models, especially with smaller networks. With GPUs being such expensive resources, there needs to be a way to multiplex tasks on GPUs, favoring resource utilization and achieving higher parallelism and better performance without investing on extra resources.

In terms of system architecture, previous work highlights the benefits of using fast storage as an intermediate step for model weights and gradient aggregation. One of the main deterrents for its usage is the increased price of cloud fast storage when compared to slower solutions like S3 [56]. Using a containerized system like Kubernetes, we could deploy these fast storage resources co-located with the cluster at no extra cost, thus reducing the impediments for its usage and increasing performance. On the other hand, [28] also highlighted that for models whose communication step is costly such as deep ResNets, standard synchronization algorithms such as S-SGD slow the convergence of the network since the model parameters need to be sent through the network, with its obvious effect on latency. We shall overcome this problem to make serverless deep learning more performant by implementing alternative algorithms studied in Section 2.2.2 that are able to relax the synchronization frequency but still have good convergence properties.

[15] https://github.com/NVIDIA/k8s-device-plugin

3

Design of KubeML

In this chapter, we analyze the requirements that must be fulfilled by KubeML[1] in terms of performance and usability. We describe which features of KubeML enable those characteristics, and provide an overall look of the system, its components and their inner-workings.

3.1. Overview

In line with the work studied in previous chapters, KubeML should implement features that accommodate both convenience and performance. We translate these objectives into functional and non-functional requirements.

3.2. Requirements

3.2.1. Functional Requirements

(FR1) **Offer an Efficient Communication Model**: In any distributed machine learning environment, communication overhead is a vital aspect to optimize. Models need to be synchronized periodically to maintain a common view of the weights and biases of the network. This is done either by sharing the model gradients or the model weights. This is even more important in a scenario where not only are models distributed among different GPUs, but also across multiple machines in a cluster. This implies that the model parameters need to be sent across the network, which entails an increase in latency, and further highlights the need to optimize the training scheme.

(FR2) **Allow Execution of Concurrent Jobs**: We specifically tackle the problem of making KubeML a potential replacement for shared serverless clusters on which multiple users can utilize resources in isolation from other concurrent tasks. This means that users' jobs should not be affected by failures of other tasks during the training process.

(FR3) **Enable Dynamic Scaling of Tasks**: As in any shared environment, usage and available resources fluctuate according to the number of tasks running at each moment and their characteristics, as well as due to potential failures in the system. KubeML must therefore adapt to these circumstances by scaling up and down the task parallelism during the training process transparently and reliably.

(FR4) **Utilize both CPU and GPU Resources**: As we have explored in the state-of-the-art chapter, a major inconvenience of current serverless platforms for deep learning is the lack of GPU support, which makes the training slow and inefficient. A key property of KubeML should be to exploit GPU resources to accelerate training. Moreover, increasing parallelism using the same amount of resources has previously been attempted with good results [34, 65], so we include the possible usage of GPUs by multiple processes concurrently in the desired properties.

[1]The source code of KubeML is available at `https://github.com/DiegoStock12/kubeml`

(FR5) Offer a Unified Solution for Training and Inference: Instead of having to prepare different deployments and setups for training and deploying the model for inference, a single function is enough to perform the training and inference thereafter.

3.2.2. Non-Functional Requirements

(NFR1) Minimal Setup and Configuration Overhead: In current distributed deep learning solutions, the process of preparing the task to be distributed can imply lengthy configurations or changes to the local code, which can undesirably complicate the development and deployment process. KubeML should allow users to prepare and setup the networks with minimal effort and without extra configuration files.

(NFR2) Convenient Transition from Local to Cloud Environment: Transitioning from a local development environment to a distributed setting should be almost transparent. KubeML should offer tools to developers to easily convert local code into a code that can be distributed across a Kubernetes cluster.

(NFR3) Enable Easy Testing: One of the inconveniences of traditional serverless platforms is that testing the functions entails deploying them to the proprietary cloud platform, be it AWS Lambda, Cloud Functions, Azure Functions, etc. KubeML, being built on top of Kubernetes, allows testing wherever a Kubernetes cluster is available. That has two positive consequences: (1) Functions can be tested in on-premise clusters or even single computers with tools such as Minikube[2] and Kind[3]. (2) There is no vendor lock-in. The same code and setup will work regardless of the cloud provider, making migrations or multi-cloud deployments easier.

3.3. Choosing a Serverless Platform

KubeML offers neural network training and inference, along with autoscaling capabilities. For this, we rely on existing solutions for hosting containerized applications as well as an open-source serverless platform.

For our base platform of choice we opt for Kubernetes, given its convenience to develop highly scalable containerized applications and its flexibility. As a second step, we had to choose an open-source serverless platform to deploy the deep learning functions. Several platforms are available built on top with Kubernetes as we reviewed in Section 2.4. To decide on a serverless platform from those available on Kubernetes, we analyzed different factors that would help clear our decision.

F1. Thorough Documentation: Is there a comprehensive amount of documentation and examples that make getting started easier?

F2. Focus on Performance: Is performance one of the platform's main focus points?

F3. Extensible Architecture: Does the architecture favor the development of new components that interact with its own?

F4. Component Isolation: Are the components of the platform separate and are there clear isolated responsibilities? This means that the failure of one component would not result in a general failure of the system. We choose this factor since the overall concept of KubeML also tries to follow this component isolation principle.

From among the pool of candidate platforms, the chosen ones for evaluation were Kubeless [36], OpenFaas [50], OpenWhisk [51] and Fission [17]. From the point of view of extending and interacting with the platform, OpenWhisk is the least favorable, given that it is written in Scala. All others allow us to interact easily with their API as well as with the Kubernetes API natively in Go. For performance measurements, we rely on previous studies comparing the three platforms [44]. In this paper, the authors benchmark the response time of the different platforms, with Kubeless

[2] https://minikube.sigs.k8s.io/docs/
[3] https://kind.sigs.k8s.io/

reporting the more predictable behavior, however Fission beats the other two by a wide margin in terms of median response time.

Moreover, Fission fully satisfies the requirements for thorough documentation and examples available, while the other solutions fall a bit short on documentation explaining the inner-workings of system components, which hinders the extensible architecture requirements. Lastly, Fission is the only one of the alternatives which separates its components into multiple containers, as opposed to having a single *controller* managing the entire platform.

3.3.1. Fission Architecture

We choose Fission[4] as our serverless platform because of its extensibility and its focus on performance [44]. Fission runs atop Kubernetes and implements the serverless paradigm using Kubernetes primitives.

Serverless Pods Serverless functions are translated into the minimum entity of Kubernetes clusters, *Pods*. Pods can be described as a self-contained group of containers, all sharing common storage and network resources. in Figure 3.1 we show the process of calling a function in Fission.

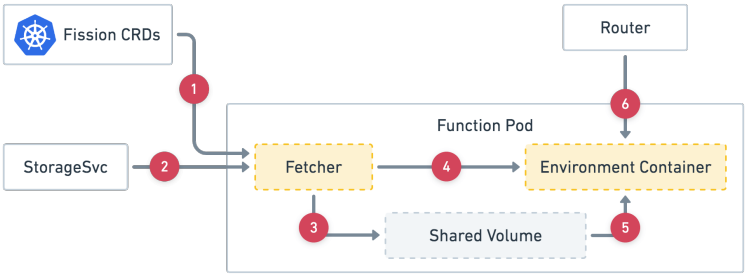


Figure 3.1: Schema of how a Fission function is invoked [17]. The *fetcher* loads the information and the function code from storage (1, 2) which is put in a shared volume so it can be accessed by the environment container (3). The *fetcher* then *specializes* the environment and makes it load the user code and prepare for execution (4, 5). When ready, the function serves the request from the router (6).

The *environment* is the container in which the function code will be executed. Initially the environment is nothing else than a generic Docker container with dependencies installed and a REST API. When a function is invoked, the container is *specialized* to serve a particular function, and will be used for that function alone until it is idle and its resources can be returned. The *fetcher* is in charge of loading the appropriate user code when a function is invoked through a trigger and *specializing* the environment, which loads the code provided and executes the main function. Generally, multiple instances of these pods are deployed simultaneously to serve requests in parallel and in isolation for a specific function type.

Request Ingress. Similar to traditional serverless platforms, requests to execute the user code are processed by a *router*. This component holds an index of the pods defined to serve each type of request and picks one of the free pods to by triggering it and handling the response once the pod is done processing the request. The router interacts with the *executor* component, which manages the available pods and sends its choice of the optimal pod to use to the router. We show this pod selection progress in Figure 3.2.

Scalability One of the main benefits of the serverless paradigm over traditional cloud computing applications is the apparent infinite scaling based on the rate of requests to a specific function. In Fission there are two approaches to scaling the number of pods of a function. We can use the `PoolManager` executor to set a minimum number of free pods to be maintained, so that when a new request arrives, a new serverless pod is created to keep a constant number of free resources. Another option is to use the `NewDeploy` executor, which uses a *Horizontal Pod Autoscaler* (HPA)

[4] `https://fission.io/`

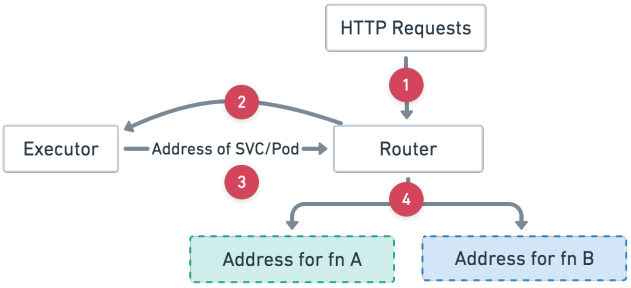


Figure 3.2: How the router sends the request to a function [17]. The Executor is responsible for maintaining the pools of available pods. The router simply queries the executor, which returns the address of a free pod. The choice of the pod will direct the requests preferably to a pod that has already been *specialized* and loaded the code and dependencies. This makes pod reuse feasible and reduces the effect of cold start.

to create pods based on resource utilization such as CPU after a set limit. For deep learning, and as a result long-running tasks, a single pod should handle the training on one replica of the network, so using the `PoolManager` executor makes more sense as it guarantees that each of the requests will be served by a separate container.

3.4. KubeML Architecture

KubeML interacts with the Fission components explained in Section 3.3.1 to train neural networks in a distributed manner. The overall system is divided into different Kubernetes Namespaces for isolation, as shown in Figure 3.3.

1. **Fission.** Where the Fission components are deployed. The main components KubeML will interact with to perform its duties are the *controller* and the *router*. The controller exposes the Fission API, so requests to create, delete or modify resources such as functions and environments will be directed to this component. The router, on the other side, behaves similarly to an API Gateway in AWS. It registers HTTP triggers, each associated with a certain function. Once triggered, the router will obtain the address of an available function pod from the executor, and forward the request to said function.

2. **Fission Function.** This is the namespace where function pods are created. Each pod includes a *fetcher* which retrieves the function code when invoked, and an environment container, consisting of a docker container with a REST endpoint, which handles requests from the router. We extend this environment container to support deep learning functions by re-building it with custom libraries such as PyTorch, Numpy, Pandas, etc.

3. **KubeML.** Where the KubeML components for exposing the API, handling the training process, and storage reside. These will be explained more in detail in the coming subsections.

4. **Monitoring.** As with most applications, observability and performance measurement is a vital aspect of KubeML. For this purpose, we use Prometheus as a metrics time series database. This allows us to track the performance of Fission components such as function response time or failures, as well as router requests. Moreover, we also include Grafana in the stack for visualizing the metrics. KubeML also exposes the metrics of training jobs in real time, so typical performance metrics such as accuracy, loss and duration is easily followed during training.

As shown in Figure 3.3, KubeML relies on different components to perform functions ranging from management operations to storage and tracking the training process. In this section we provide an in-depth explanation of each component and its task. To enable straightforward interaction with both the Fission and Kubernetes APIs, all the components except for the storage service are written in Go. In addition, similar to Fission and the Kubernetes API, all components expose a REST API, and communicate with each other using HTTP and JSON.

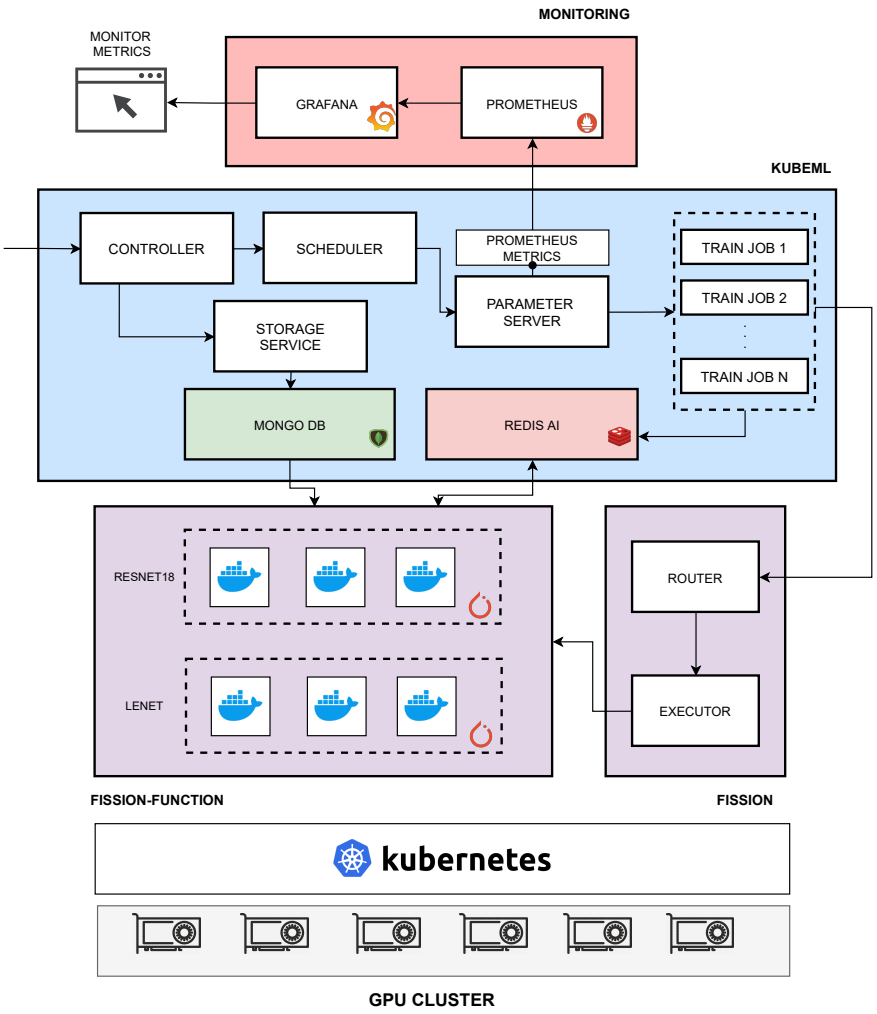


Figure 3.3: Schema of the KubeML Components

3.4.1. Controller

Similar to the concept found in Fission, the controller acts as a gateway to the system, exposes the KubeML API and acts as a proxy towards other components in requests. The Controller API exposes several endpoints to create, delete and modify several resource types in the system:

- `/train` Receives the specifications for a training task. Each train task can be configured with parameters such as the batch size, learning rate, dataset, function name, target accuracy, etc. These requests are then validated and forwarded to the scheduler.
- `/infer` Receives an inference request, holding the function name and the data to be used as input to the network. This request is also forwarded to the scheduler.
- `/dataset` Handles dataset creation and deletion. The dataset upload is done through a multipart upload to handle big datasets efficiently. In these cases, the controller acts as a reverse proxy and forwards the request to the storage service, which processes and stores the data in a predefined format.
- `/history` Retrieves the history of a training job, that is, the summary of the performance of the job throughout the training process. This includes accuracy and loss per epoch, parallelism, duration, etc. for all epochs of the job.
- `/task` Allows the user to interact with the running jobs of the system, such as obtaining information, listing or stopping them.

3.4.2. Scheduler

Similarly to the Kubernetes Scheduler, the KubeML scheduler receives requests to start a training job or perform an inference request. Once tasks arrive at the scheduler, they are queued and wait to be processed. Once the task is popped from the queue, the scheduler executes the appropriate actions based on the task type and the scheduler *policy*.

- **Inference Tasks.** In the case of inference tasks, the scheduler directly invokes the function with the parameters passed alongside the requests, thus acting as a proxy towards the function pods. Since inference will mostly be allocated to CPU resources, which will be less utilized than GPU, the scheduler simply invokes the appropriate function in the router in forwards the results.
- **Train Tasks.** In the case of train tasks, the scheduler's purpose is to scale up and down the parallelism of tasks based on the rules specified in the policy being used. The policy is defined as an interface that can be extended to determine the conditions for scaling.

  The current policy is a reactive one based on throughput and response time. The scheduler will allocate a start parallelism to a task, and after every epoch, the task will send its parameters such as elapsed time and throughput to the scheduler. The scheduler will optimistically increase the parallelism until the throughput of a task stops improving. In case the throughput of the task suddenly drops under a certain threshold, the parallelism is scaled down. This is to address the possibility of multiple tasks running concurrently on the system and having to reduce resource usage to fit all of them. This means that we can reactively scale up or down the number of functions running in parallel at runtime without any changes to the code, thus fulfilling **FR3**.

3.4.3. Parameter Server

Receives the specification of training tasks from the scheduler and starts a new Train Job for each. Instead of sharding parameters by name, it shards them by task, so each has its own dedicated pod to update its parameters. It forwards the updates from the scheduler to the appropriate train job, as well as manages the lifetime of each job, from creation once the task is received to deletion once finished to release resources.

This component is also in charge of exposing the metrics from all the jobs in the system via a Prometheus endpoint. The jobs update the parameter server periodically with their latest metrics, and these are updated in the endpoint, and thus gathered in the Prometheus server.

- **Train Job Management.** Given that KubeML should be able to host concurrent jobs from different users, one key property of the platform should be job isolation. The failure of one job should not affect the overall usability of the platform.

  To achieve this, the Parameter Server creates the Train Jobs in a new pod isolated from the rest of the components of the system to satisfy **FR2**. At creation time, using the Kubernetes API and a custom ClusterRole, the Parameter Server creates a new pod for the job, as well as a new Kubernetes service. The creation of the service facilitates referencing the job from the serverless functions by providing the pod a DNS name and entry in the cluster DNS.

  During the training process, the Parameter Server will act as an intermediary between the scheduler and all the jobs, handling the new parameters such as the parallelism degree sent from the scheduler.

  Lastly, once the job finishes, the parameter server cleans up the job resources such as the pod and service, and clears the metrics from the Prometheus endpoint.

- **Metric Collection.** Apart from managing the pods and services, the Parameter Server also serves as a single publishing point for the metrics of all the jobs running in the system. After each iteration, the jobs communicate with the parameter server their latest measurements of the defined metrics. The default metrics reported to the server are validation loss, accuracy, train loss, parallelism and epoch duration.

  The metrics are then exposed in a custom Prometheus endpoint to be fetched periodically by Prometheus. These metrics can be then queried and manipulated, as well as analyzed with tools with Grafana, to follow the training progress of jobs.

3.4.4. Train Jobs

The Coordination of the deployment of new serverless functions to run during the next epoch is based on the abstraction of Training Jobs. Jobs maintain the reference model, the common view of the network weights and parameters. In a Data Parallel approach, different replicas of the same network are trained on disjoint subsets of the dataset, thus fitting only a subset of the overall distribution of datapoints. Periodically, during the training job, functions will synchronize their current views of the model resulting in an average model combining the views from all networks, which will be used as the starting point for the next iteration.

Function Invocation Jobs invoke a number of functions at the start of each epoch, this number is defined by the job *parallelism* level. During the epoch, the job will listen for requests sent by the functions to its API updating the partial models and requesting a model average. The job effectively acts as a synchronization point between the functions, and concurrently merges the models and notifies functions to continue with the next iteration.

Weight Aggregation Running the training process in a serverless environment adds extra restrictions, since functions cannot communicate directly with each other, thus requiring an intermediate component that performs the aggregation of the model weights or gradients. This role could be performed by a traditional parameter server [7] or with an intermediate storage [61, 28].

In KubeML, the same function pod is used for the entire duration of a training epoch, which can have multiple synchronization points. In each synchronization point, functions notify the train job, which proceeds to average the models. The train job communicates with the functions only through the *router*, so it does not effectively know the exact addresses of the currently running functions.

For this reason, we decide to rely on RedisAI[5] as a high performance storage for the reference model and the function models during training, used to communicate the weights from the functions to the job and vice versa. We will describe in more detail in Section 3.5 how the weights are communicated and combined into the reference model.

3.4.5. Storage Service

In order to provide a transparent experience, users upload their datasets in the same format as they would use locally to the KubeML dataset storage service. The storage service is in charge of preprocessing the datasets uploaded by the users to optimize the training process and make it easier to distribute the subsets to different functions.

Users can use the KubeML CLI[6] to upload a dataset to KubeML. The storage service is the only component written in Python so it can modify the uploaded data. The storage service accepts files in the most common formats such as `.npy` and `.pkl`. Once uploaded, the service divides the dataset in fixed size subsets of a manageable size like 64 datapoints, which are then saved in multiple documents to the storage backend. This allows to easily distribute the datasets among functions by loading just the subsets that a function needs to train on. These subsets pertaining to each of the functions is easily calculated using the function id and the number of functions in the system. A detailed explanation of the function inner-workings will be provided in the following sections.

KubeML relies on two different storage platforms to maintain both datasets uploaded by the user and model parameters of the different deep learning functions during a job. We opt for a hybrid storage solution using a document store for dataset storage due to its convenience for organizing labels and targets of a dataset in a single entity, and a fast key-value store for intermediate storage given the advantages that it has shown in other work performance-wise [56].

3.4.6. Dataset & Model Storage

KubeML relies on two different storage platforms to maintain both datasets uploaded by the user and model parameters of the different deep learning functions during a job.

[5] `https://oss.redislabs.com/redisai/`
[6] We will cover in more detail the usage of the KubeML CLI in Section 3.8.1

- **Dataset Storage.** The dataset storage holds the datasets uploaded by the user and processed by the storage service. The datasets are stored divided in multiple subsets of a fixed size to allow easy access to the corresponding batches of data by each of the deep learning functions.

  When selecting the storage platform for datasets, three main requirements were considered:

  1. The platform must allow for straightforward storage of binary data. The storage service divides the dataset in multiple subsets and saves in each subset the arrays representing the features and labels in binary format. The database must allow for easy storage and retrieval and also filtering by range of subset id.
  2. High read throughput. The usage of the dataset storage consists of predominantly read operations, thus it should offer fast read performance.
  3. Collocation with functions. The storage platform should be available close to the functions to maximize read performance.

  Given these conditions, we settled on MongoDB as the main dataset storage solution, given its performance on read operations and the easy containerized deployment in the same Kubernetes cluster as the functions. In the future, other storage backend solutions could be implemented, such as MinIO[7], or S3 in case the Kubernetes cluster is deployed in AWS.

  Each dataset is saved as a MongoDB database comprising two collections, the `train` and `test` collections. Each collection holds several documents, each consisting of an id, and a data and labels field, containing the binary representation of a subset of features and labels of the dataset.

- **Model Storage.** Each deep learning function trains a replica of the model in a subset of the data, these models have to be combined to produce the reference model periodically by sending the gradients or weights to the parameter server. In KubeML this task is performed by the Train Job. In KubeML, rather than directly communicating with each of the functions and sending the model N times through the network, the Train Job and the functions used a high performance storage platform, RedisAI[8] as an intermediary to communicate each of their models.

  RedisAI is a module for the Redis key-value store which makes interacting with tensors much easier by abstracting away conversions between programming languages and libraries. RedisAI also brings the benefit of high read and write throughput, which improves the performance of the model merging part of the training.

  The choice of RedisAI allows us to introduce optimizations to improve tensor loading in the Train Job, decreasing latency and increasing concurrency. We employ connection pooling, which allows us to maintain a pool of ready connections which saves initialization costs when wanting to access the tensor storage. Moreover, using multiple connections allows us to increase concurrency, by having *P* threads load the model weights concurrently from the functions with a parallelism degree *P*. On top of that, transmitting the tensors one by one is a costly operation. Especially when transmitting hundreds of small tensors, communication overhead starts playing a dominating role compared to computation. We resort to *pipelining* to perform the tensor fetch calls asynchronously, so we are able to perform multiple commands in a row without having to wait for previous results, consequently increasing throughput.

3.4.7. Deep Learning Functions

KubeML uses a data-parallel approach to accelerate the training process of neural networks. This means that a replica of the network is initialized with the same weights as the reference model in each iteration, and trained on a different subset of the data. The functions then synchronize the models again and continue the training process.

[7] `https://min.io/`
[8] `https://oss.redislabs.com/redisai/`

KubeML functions are written in Python, using PyTorch. Users can reuse the same code they would use to train the network locally and upload it to the Kubernetes cluster with the help of the CLI. PyTorch code is wrapped using the KubeML Python library, which provides a simple interface allowing using the same local code. More details on how the module works will be provided in Section 3.7.

We take advantage of PyTorch only using as much GPU memory as needed for training to allow multiple serverless functions to be allocated in the same GPU. With small models, a big part of the GPU resources remain unused, so this approach will increase resource utilization as well as improve performance without additional hardware resources. We follow a similar approach to [65] by configuring Nvidia Docker to allow multiple pods to be scheduled in the same GPU. We do not reach the same level of optimization as in that work, since that would involve the development of a new Kubernetes Scheduler and a new Fission Executor that could decide which serverless pod to use for deep learning functions to minimize the chance of over-allocation errors. Using this approach we address **FR4** by incorporating GPUs into the serverless deep learning ecosystem.

Function Lifecycle A key aspect when designing a serverless application is to account for the function slow start time. Slow start is a characteristic of serverless platforms, which occurs after a request arrives for a function without an available container ready to serve it. In these cases, the platform needs to start a new VM or container to serve that function, which results in a delay in function execution.

In KubeML, a single invocation of a function will perform the training for a whole epoch. After this, the statistics for that specific function are returned as the HTTP response, and the pod is returned to the pool of available ones. In order to maximize performance and reduce the chance of slow start taking place, we configure Fission so that requests performed soon after the pod is freed can still reuse the same container. This approach circumvents the process of importing libraries, initializing connections and importing the user code.

Fission's *executor* component keeps the state of the pods being used and performs sticky routing, meaning that if a pod is already set to serve requests for a given endpoint, new requests will be directed to these running pods instead of initializing new ones.

Sticky GPU Choice As explained above, reusing function pods before its resources are returned results in considerable time savings at invocation time. When using GPUs, another time-consuming operation is creating the model and allocating it to the GPU. This is done PyTorch through the `.cuda()` method, and can be a lengthy operation for larger networks. On top of that, we also want to minimize the different allocations performed by functions when training on GPU. This means that the device used by a function should be the same in subsequent invocations.

To solve this, we use the function id (f_{id}) parameter passed to each pod as a tool to define the GPU to be used. In each server, with a number of GPUs equal to g, we choose the GPU to be used by that specific function by calculating the modulo of the id with respect to the number of devices such as $gpu_{id} = f_{id} \mod g$. However, in the subsequent invocations of that pod, the function *id* could differ, resulting in allocating memory in another GPU. To avoid this, we set an environment variable which is read at the start of each invocation, which makes each function reuse the memory allocation used in previous iterations, thus increasing initialization speed.

Throughout the training process of a network, several steps must be performed. The first time a network is created, its weights and biases can be initialized in a predetermined way to help convergence. Later, during the training process, the iterative process of training and evaluating the network on the train and validation data is performed until the target number of epochs is reached. Lastly, after the network is fully trained, it is commonplace to perform inference by having the network predict new data. In KubeML, the same function code can be run to perform each of these steps. The Train Job, with each call to the functions, passes certain arguments along with the HTTP request that provide the function with context for the request, and makes it execute a certain task.

- **Init Function.** The init function's task is to create and initialize the weights of the network, and save the initial model to the model storage. This first model is then used as the base for

creating the reference model in the Train Job. This function is invoked once in each train job before the training process starts.

- **Train Function.** Performs the training of the function on a subset of the data. Depending on the parallelism of the job, each function calculates the number and id of the subsets that it must load from the dataset storage, and loads the reference model from the model storage. During training, functions might synchronize the model multiple times per epoch, marked by a *k* parameter, which indicates the number of batches before sync. This is done via an HTTP request to the Train Job service. More details of the training algorithm and its benefits will be covered in Section 3.5.
- **Validation Function.** This function tests the model on the validation dataset and returns the validation loss and the accuracy of the network on the unseen data.
- **Inference Function.** Given a new set of datapoints, the inference function loads the trained model and returns the outputs for those new datapoints.

3.5. Training Algorithm

There have been many proposed algorithms to handle the distributed training of neural networks. The most well-known of them might be Distributed Synchronous SGD (S-SGD) [8]. In S-SGD, multiple models are trained on different batches of the data, and after each forward pass, all of the models are synchronized before the next iteration. This approach mimics the behavior of sequentially training the network given that all workers have a common view of the reference model at the start of each iteration. This frequent syncing, although reporting good final accuracy, makes the training process slower and more sensitive to stragglers.

To mitigate these problems and provide the workers some leeway in terms of syncing, asynchronous methods were proposed, which allow workers to continue to the next iteration after syncing without waiting for all workers to be finished. This maintains the same communication overhead, however it improves the handling of straggler workers. Cases of these include Elastic Averaging SGD (EASGD) [75], bounded staleness methods [79], and lock-free optimistic approaches [58]. Although these methods generally increase performance in terms of time, they also make it harder for the model to converge. Most recently, it has been argued that Async-SGD methods lose benefits once used with bigger models, given their limited speed improvement and worse accuracy [69].

3.5.1. Implementing Local SGD

In KubeML, functions are deployed in multiple containers spread across machines in a cluster, having to communicate through the network. This further underlines the need to minimize communication to reduce latency. Moreover, we also want the benefits of synchronous SGD. To allow this, we use a synchronous algorithm which combines characteristics from EASGD (synchronizing after multiple batches) with strong convergence properties from synchronous methods.

We use Local SGD [80, 81, 63, 41] , also known as K-AVG SGD or Parallel SGD. In Local SGD, each worker trains for multiple epochs before syncing with the reference model. The number of epochs before sync is commonly referred to as *k*, hence the name K-AVG. This parameter *k* represents a balance between exploration (each worker exploring a concrete region of the loss space) with big *k* and exploitation (all workers exploring the same region of the loss space) with small *k*. In our case, we define the maximim value for *k* to be one epoch, which we represent as $k = \infty$. For merging the models into a single reference model, instead of aggregating the gradients like in other solutions, we take the approach of performing model averaging [34]. During the sync step, the models from all of the workers are averaged to obtain the new reference model, following Equation (3.1).

$$\tilde{\boldsymbol{w}}^{(t+1)} = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{w}_i^{(t)} \tag{3.1}$$

Where $\tilde{\boldsymbol{w}}$ represents the new reference model weights, and N is the number of workers, being w_t^i each of the workers' weights. An intuitive representation of the operations performed during

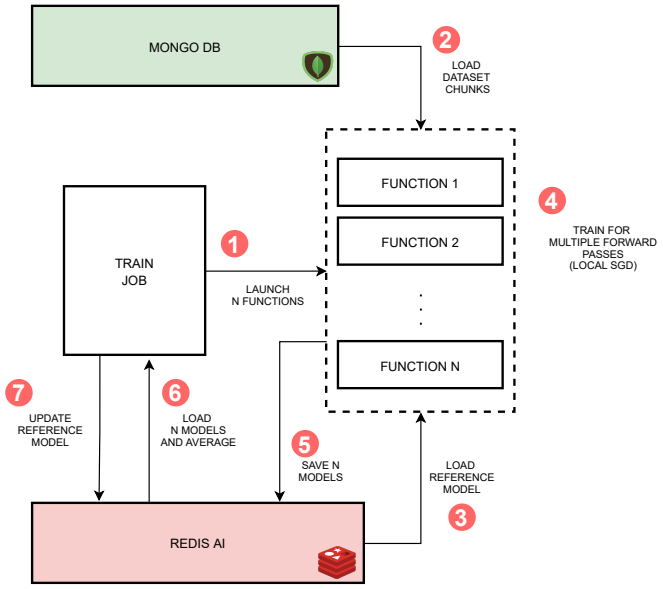


Figure 3.4: Overview of the Local SGD Training Algorithm

the training process can be seen in Figure 3.4. Each of the functions trains for k batches, and saves its model to the model storage. The Train Job waits for all functions to be ready, and reads and averages all the models from the functions. Finally, the train job publishes the model weights and the functions proceed to the next iteration. This effectively reduces communication overhead by a factor of k compared to synchronous SGD.

One important decision is the choice of the k parameter. Interestingly, in [80], authors prove that Local SGD scales better than methods like Async-SGD and EASGD with the number of workers, but that the maximum accuracy is often reached with a k parameter bigger than 1, which would be equal to S-SGD, thus highlighting the benefits of exploration of different loss spaces by the different models. We experiment with the values of k and their influence in performance for different networks in Chapter 4. By using Local SGD, we offer the choice of prioritizing communication efficiency by increasing the k parameters, which results in a linear decrease in communication overhead. This addresses the requirement for efficient communication (**FR1**) needed to tackle the excessive latency introduced by sending the gradients over the network.

3.6. Implementation Details

We implement the core components of KubeML in Go, to enable simple integration with the Kubernetes and Fission APIs and to favor performance. The storage service and the KubeML library are implemented in Python to enable interaction with common deep learning libraries. The summary of the components and their code size can be seen in Table 3.1.

Component	Written In	Lines of Code
KubeML Core	Go	4.5K
Storage Service	Python	100
KubeML Library	Python	440

Table 3.1: Lines of Code for each of the components

3.7. The KubeML Module

As mentioned in the motivation for the system, one of the focus points should be to make distributed deep learning straightforward and accessible, avoiding the need to perform lengthy configuration of algorithms or servers. We also want to let users reuse as much as their code developed for local testing without having to rewrite considerable parts of the training logic. For this

reason, we developed the KubeML[9] Python module, which lets users convert their PyTorch code into distributed code with minimal overhead, addressing **NFR2**.

3.7.1. The KubeML Classes

As part of the KubeML library, the users are provided with two classes that serve as abstractions for KubeML objects and transparently distribute the learning task once deployed in a Kubernetes cluster.

1. **KubeDataset.** Responsible from abstracting the process of loading the appropriate dataset chunks for the function from the dataset storage. The *KubeDataset* class extends the PyTorch *Dataset* class and implements the KubeML logic. This allows it to be used with all other PyTorch objects such as a *DataLoader*, thus minimizing the needed code changes.

2. **KubeModel.** Wraps a PyTorch model and abstracts away the process of synchronizing with the train job periodically, as well as loading and saving the model in the model storage during the training process. The *KubeModel* is defined as an abstract class in which users define the `train, validate, infer` and `init` methods according to their preference. These methods are then invoked interleaved with the distributed logic, allowing the user to use the same code as locally. This means that the same function can be reused for all of the operations from training to inference without the need to separate the inference logic, thus satisfying **FR5**.

3.7.2. Detailed Component Interaction

As described in the previous section, we develop the KubeML module to accommodate the same local code without any significant changes and make it scale to multiple GPUs or nodes in a completely transparent way. In this section, we explain the basic inner-workings of the library, and how the data and iterations are handled without the user having to know about what is happening under the hood.

Remember that all communication in both Fission and KubeML is handled through HTTP, this makes it convenient to embed information in requests to define the behavior of the serverless functions. This is how the component in charge of invoking the functions, the Train Job, transfers the parameters needed by the functions at the start of each epoch. We define a helper class `KubeArgs` that parses the arguments encoded in the request URL, which are used by the `KubeModel` to prepare the model and data at the start of the epoch.

KubeDataset. Acts as a wrapper that transparently handles loading the appropriate subsets of data that will be needed to train the network before the next synchronization point. With Local SGD, a function might need to synchronize periodically after *k* forward passes with the Train Job. To save memory, the `KubeDataset` performs a lazy loading of the data. Rather than fetching all the datapoints when instantiated, a `KubeDataset` object exposes two methods that given the first and last dataset subset to load, performs a range query to the database, loads the data and exposes it through the `data` and `labels` member variables. This operation is triggered by the `KubeModel`, which calculates the needed subsets and indicates whether to load validation or train data.

KubeModel. At creation time, the `KubeModel` receives three main parameters: the torch module representing the network, a `KubeDataset` object that will hold the data used for training, and a flag of whether the training will use GPUs. After the components are initialized, the `KubeModel` exposes a `start()` method, which under the hood triggers the request arguments to be parsed, and depending on the `task` flag in the request, it results in the execution of a given hook provided by the user (Listing 1).

Each of the private methods executed as a result of the `start()` method act as wrappers to the user hooks. The `infer` method parses the JSON attached to the request body holding the new feature arrays into a `numpy` array and executes the forward pass of the network, returning the predictions. The `init` method simply applies the user init hook to the model weights, before

[9] `https://pypi.org/project/kubeml/`

publishing the model weights to the model storage. The behavior of the `train` and `validate` methods is more complex, since it needs to handle all the logic from the Local SGD algorithm and synchronization with the Train Job.

```
def start(self)
    """
    Start executes the function invoked by the user
    based on the request context flags
    """
    # parse arguments and initialize network logger
    self._read_args()
    self._get_logger()

    # depending on the task flag, execute one of the
    # registered hooks by the user
    if self.task == "init":
        layers = self.__initialize()
        return jsonify(layers), 200

    elif self.task == "train":
        loss = self.__train()
        return jsonify(loss=loss), 200

    elif self.task == "val":
        acc, loss, length = self.__validate()
        return jsonify(loss=loss, accuracy=acc, length=length), 200

    elif self.task == "infer":
        preds = self.__infer()
        return jsonify(predictions=preds), 200

    else:
        self._redis_client.close()
        raise KubeMLException(f"Task {self.task} not recognized", 400)
```

Listing 1: Underlying code of the `KubeModel` to trigger the multiple actions.

Taking the `train` method as an example, the `KubeModel` first calculates the ids of the subsets that the network will train on before the next synchronization step. This is easily calculated using the total number of functions or parallelism N and the function id $f_{id} \in \{0, 1, \dots, N - 1\}$. Once the assigned subsets are defined, the function needs to know how many subsets it should train on before the next synchronization step, to minimize memory usage by only loading those. This is easily calculated since the k parameter and the batch size (B) are passed in the URL arguments following Equation (3.2).

$$N_{subsets} = \left\lceil \frac{B \times k}{S} \right\rceil \tag{3.2}$$

Where S represents the subset size in the storage, or how many datapoints are stored in each document in the database. This amount is set to 64 by default, to try to balance the number of requests needed to load data but also limit the size of each document in the database.

With the number of subsets per iteration and the interval of subsets assigned to each function, the network proceeds to the training stage. At the start of each iteration, the `KubeModel` loads the appropriate subsets into the `KubeDataset` and the latest model weights from the model storage. Then trains the model iteratively using the user-provided hook until reaching the next synchronization point. At this stage, it saves the model to storage and requests the Train Job to average the models and publish the new reference model. In the meantime, the model keeps track of the intermediate metrics of the training, and aggregates them and returns them after an entire epoch is complete. A simplified version of the code for Local SGD training is shown in Listing 2.

These abstractions allow users to use the same code that they would use to train the network locally while transparently distributing the training of the network across any number of CPUs, GPUs, nodes, or both. In the following sections, we will provide examples of how simple it is to

```
    for i in intervals:

        # load the appropriate data batches from the dataset storage
        self._dataset._load_train_data(i)
        loader = DataLoader(self._dataset, batch_size=self.batch_size)


       # load the reference model and reset
       # optimizer state
        self._on_iteration_start()

        for idx, batch in enumerate(loader):
            # send the batch to the appropriate device, preventing the user
            # from having to know which physical device the model is running on
            batch = self._batch_to_device(batch)

            # call the train hook defined by the user with the batch already
            # moved to the GPU or CPU, based on the network location
            loss += self.train(batch, idx)

        # publish the local model to be averaged
        self._on_iteration_end()

        # in intermediate sync points, request average to
        # the train job
        if i != intervals[-1]:
            self.__send_finish_signal()


    self._on_train_end()
    return loss / num_iterations
```

Listing 2: Simplified code depicting the logic behind the train process.

write functions in KubeML, following one of the main benefits of serverless as is the convenience that it provides to users compared to alternative paradigms.

3.7.3. Writing KubeML Functions

When wanting to transition to a distributed training job on Kubernetes, a user needs to write a serverless function and embed their local code in it. The steps to be followed to write a distributed network are:

Define the Dataset object. The user must create a dataset object extending the *KubeDataset* class, same as would be done with a traditional PyTorch dataset. There, the user defines the name of the dataset as uploaded to the dataset storage and defines custom functionality like dataset transformations. The dataset exposes the `data` and `labels` member variables with the features and labels loaded from the dataset storage. An example of a Dataset object can be seen in Listing 3.

Create the KubeModel. As with the dataset class, the user must extend the *KubeModel* class and implement the network's initialization, training and inference functionality. This is done as shown in Listing 4. As can be seen in the example, the code used to define the training behavior of the model is the same that could be used to train locally, which takes away the complication of using alternate classes and algorithms to train distributedly.

Write the Main Function. After defining the dataset and model classes, the user must define the `main` method run when invoking the functions. An example of this is given in Listing 5. There, a PyTorch model is created, alongside the dataset created in Listing 3. These are then fed to the network defined in Listing 4, and the `start` method is invoked to start the training.

```python
class MnistDataset(KubeDataset):

    def __init__(self):
        # Provide KubeML dataset name
        super().__init__("mnist")
        self.transf = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,),
                                 (0.3081,))
        ])

    def __getitem__(self, index):
        x = self.data[index]
        y = self.labels[index]

        return self.transf(x), y.astype('int64')

    def __len__(self):
        return len(self.data)
```

Listing 3: Python code to define a KubeDataset, which takes care of loading the appropriate data samples for each iteration during the training and validation phase. The user can define custom transformations and return any number of objects in the ___getitem___ method, since passing the batch arguments to the desired backend (CPU or GPU) will be transparently handled by KubeML.

```python
class KubeLeNet(KubeModel):

    def __init__(self, network, dataset):
        super().__init__(network, dataset, gpu=True)

    def configure_optimizers(self):
        sgd = SGD(self.parameters(),
                  lr=self.lr,
                  weight_decay=1e-4)
        return sgd

    def train(self, batch, batch_index):
        x, y = batch
        self.optimizer.zero_grad()

        output = self(x)
        loss = F.cross_entropy(output, y)
        loss.backward()
        self.optimizer.step()

        return loss.item()

    def validate(self, batch, batch_index):
        x, y = batch

        output = self(x)
        _, predicted = torch.max(output.data, 1)
        test_loss = F.cross_entropy(output, y).item()
        correct = predicted.eq(y).sum().item()

        accuracy = correct * 100 / self.batch_size
        return accuracy, test_loss
```

Listing 4: Example definition of a KubeModel. Boilerplate code is avoided and users just need to define the forward pass.

3.8. Deploying Jobs to KubeML

Just like the process of writing the distributed functions, the process of deploying the functions to the cluster should be as straightforward as possible. In the first place, we take advantage of Kubernetes' convenience and create Helm[10] charts for KubeML. Helm is a package manager for

[10] https://helm.sh/

```python
def main():
    # create a PyTorch network and the dataset
    lenet = LeNet()
    dataset = MnistDataset()

    # initialize the KubeModel and start
    kubenet = KubeLeNet(lenet, dataset)
    return kubenet.start()
```

Listing 5: Main method run by the serverless functions

Kubernetes, which allows deploying all of the components of a containerized application to a Kubernetes cluster using a single command. This streamlines the installation of KubeML in both cloud and local clusters, and satisfies **NFR3**, since KubeML can be deployed in any Kubernetes cluster for testing.

As for deploying the functions written in the previous sections, we wanted to avoid having to write YAML files as is common with other Kubernetes applications, as we saw in Section 2.4 with OpenFaaS. To solve this, we created the KubeML CLI. This command line interface allows to deploy and start a training job with a couple of commands without the need of writing configuration files or any having to know low-level implementation details about Kubernetes or the system itself. With this, we also satisfy the requirement for straightforward deployment and interaction with serverless functions (**NFR1**), thus completing all of the requirements summarized in Section 3.2.

3.8.1. The KubeML CLI

Like with other serverless systems, the KubeML CLI is the easiest way to create and manage resources such as functions, datasets and train jobs. Additionally, KubeML also offers a Go API with which users can interact, exposed by the controller both inside and outside the cluster.

The commands available are summarized in Figure 3.5. Tasks like uploading a function and dataset are completely automated and can be performed with just a single command, as well as starting a train task. Throughout this section, we will provide a brief guide of how a user can utilize the CLI to quickly set up a new deep learning job without in-depth knowledge of either Kubernetes, Fission or KubeML's implementation, following the serverless principle of simplicity.

3.8.2. Deploying Functions

Once the function code is written as explained in Section 3.7.3, users can deploy them to the cluster with a single command by referencing the Python file. The CLI interacts with the Fission components of the cluster to package the source code and create the needed HTTP triggers for the function to be accessible. An example of deploying a function is given in Listing 6 By default, the CLI will create a new code package using Fission's storage API, so that the code can be loaded at runtime by the *fetcher*.

KubeML by default creates two HTTP triggers to enable both training and inference. A GET endpoint will be used when triggering the function's `init`, `train` or `validate` methods, and the POST endpoint will be used when uploading the new datapoints for the `infer` method. Likewise, users can delete functions with one command, which will clear all the created resources.

```
$ kubeml fn create --name lenet --code lenet.py
```

Listing 6: Command to deploy a function to a cluster

3.8.3. Uploading a Dataset

In the same way that functions can be transparently uploaded, the CLI enables uploading datasets to the cluster in a single command. The user needs to divide the dataset in features and labels both for train and validation data, before using the `dataset` subcommand referencing the data as shown in Listing 7. The data can be saved in common formats like `.npy` or `.pkl`.

```
kubeml
├── function (Create/Delete functions)
│   ├── create
│   ├── delete
│   └── list
├── dataset (Create/ Delete datasets)
│   ├── create
│   ├── delete
│   └── list
├── history (Inspect histories of past jobs)
│   ├── get
│   ├── delete
│   ├── list
│   └── prune
├── train (Create a train job)
├── infer (Send an inference job)
├── task (Inspect currently running tasks)
│   ├── list
│   ├── prune
│   └── stop
└── logs (Access logs of train jobs)
```

Figure 3.5: Commands and functionality available in the KubeML CLI

```
$ kubeml dataset create --name mnist
                        --traindata x-train.npy
                        --trainlabels y-train.npy
                        --testdata x-test.npy
                        --testlabels y-test.npy
```

Listing 7: Command to upload a dataset to KubeML

After the command is executed, the CLI performs a multipart upload, so bigger datasets can also be handled efficiently. Internally in the cluster, the controller proxies the handling of the dataset upload to the Storage Service, which divides the data in subsets so they can be easily handled by the functions at train time.

3.8.4. Starting the Training Process

After the function has been deployed and the dataset uploaded as shown in the previous sections, the training job can be started using the `train` command of the CLI as shown in Listing 8. The train job is run asynchronously, the CLI returns the ID assigned to the train job, which will be used to index the job during training and fetch the results once the job is finished.

```
$ kubeml train --function lenet
               --dataset mnist
               --batch-size 64
               --lr 0.01
               --epochs 10
```

Listing 8: Command to start a training job

Apart from these basic parameters, other extra options are provided to customize the training process, which are shown in Table 3.2.

Option	Type	Description
`--parallelism`	`int`	Initial parallelism level
`--static`	`bool`	Whether to keep parallelism fixed
`--validate-every`	`int`	Period of validation
`--K`	`int`	Define synchronization period
`--target-accuracy`	`float`	Target Accuracy to be reached

Table 3.2: Options for the training job

3.8.5. Monitoring Function Performance

KubeML runs train jobs asynchronously since deep learning jobs can take hours to days to finish for some more complex networks and datasets. Consequently, it must offer a way to follow the progress of the job and metrics such as loss and accuracy. Users can track which tasks are running at a single point in time using the `kubeml task list` command. They can also interrupt a task before it has finished using the `kubeml task stop --id <id>` command.

For a more detailed summary, however, KubeML utilized Prometheus and Grafana. As was described when introducing the parameter server component in Section 3.4.3, train jobs periodically communicate their statistics to the PS which exposes them using a Prometheus endpoint. These metrics are gathered by the Prometheus server located in the `Monitoring` namespace, and can be visualized with Grafana using a custom-made dashboard.

3.8.6. Examining Function Results

Once the train job has finished, the results from the training job such as the job parameters and the metrics gathered during training are stored in the job history and saved to storage. These histories summing up the job behavior can be fetched with the help of the CLI to analyze the results with the `history` command.

```
$ kubeml history get --id b71b1d11
```

Listing 9: Command to fetch a train history

An example of a train history saved to storage can be seen in Listing 10. There, the epoch-by-epoch metrics of the network as well as the job parameters and options can be quickly analyzed.

3.9. Conclusion

In Chapter 2 we went through the basics of distributed deep learning and the serverless paradigm. From there, and with the help of previous work on merging both of these technologies, we devised a series of objectives to be solved by KubeML, such as incorporating GPUs into the system, implementing an efficient communication protocol, and allowing straightforward creation and deployment of deep learning tasks.

In this chapter, we have presented KubeML, which tries to solve all of the issues present in previous work and highlighted by the authors. KubeML builds on top of Fission, an open-source serverless platform focused on extensibility and performance, which itself runs atop Kubernetes. Serverless functions run in the form of Kubernetes Pods holding Docker containers. This allows to create a custom container image with dependencies for running PyTorch applications and exploit GPUs. KubeML uses Nvidia Docker to expose GPU resources of cluster nodes so that the serverless pods can take advantage of these resources. On top of that, following the approach of previous work [45, 65], we allow multiple pods to use the same GPU, with the aim of improving utilization and efficiency.

To solve the communication overhead problem, we opt for an algorithm which lets us adjust the communication period such as Local SGD. The Train Job component is in charge of invoking, synchronizing, and publishing the functions and its results. Functions run for an entire epoch in each invocation, possibly synchronizing at multiple intermediate points to average their partial models, specified by the *k* parameter.

Lastly, we develop the KubeML library in Python, as well as the KubeML CLI to ease the process of developing and deploying functions to the cluster. Users can plug in their PyTorch code used to train locally and, with minor adjustments, convert it into a distributed training program, able to scale to multiple GPUs, nodes, or both transparently without configuring extra daemons or parameters. Once the function code is ready, users can create serverless functions and upload datasets with a single command using the CLI. Finally, starting the training can be done with a single command, and parameters such as parallelism and autoscaling can be configured at deploy time.


```
{
  "id": "b71b1d11",
  "task": {
    "model_type": "lenet",
    "batch_size": 128,
    "epochs": 5,
    "dataset": "mnist",
    "lr": 0.01,
    "function_name": "lenet",
    "options": {
      "default_parallelism": 5,
      "static_parallelism": true,
      "validate_every": 0,
      "k": 10,
      "goal_accuracy": 90
    }
  },
  "data": {
    "validation_loss": [
      0.2873407344818115
    ],
    "accuracy": [
      91.4
    ],
    "train_loss": [
      3.2260020107030867,
      1.1191881984472274,
      0.6412935853004456,
      0.533321401849389,
      0.43479404859244825
    ],
    "parallelism": [
      5,
      5,
      5,
      5,
      5
    ],
    "epoch_duration": [
      5.311429519,
      2.815807104,
      2.785033174,
      3.161798912,
      2.781011169
    ]
    }
}
```


Listing 10: Example of a train history

4

Experiments

In this section, we evaluate the performance of KubeML training on different environments and models. Our evaluation focuses on three main aspects: performance, efficiency and convenience. First, we test the performance during training tasks against TensorFlow using smal, medium and big networks on a variety of datasets. With these experiments we intend to study the viability of using Local SGD to train networks of different parameter sizes, and whether the increased parallelism of the training process can have a degrading effect on the convergence of the network. We also evaluate the efficiency reached by KubeML in comparison to TensorFlow by analyzing the usage of resources of both systems. Lastly, we demonstrate the superior convenience of KubeML by showing how the same code can be reused with minimal changes to migrate the training settings from single to multiple machines with GPUs or CPUs.

4.1. Experimental Setup

Platform For our multi-GPU server, we use a machine configuration with 2 NVIDIA RTX 2080 Ti and two 32-core AMD EPYC2 CPUs with SMT-2 enabled (128 hardware threads in total). As a system baseline, we compare the performance of KubeML against TensorFlow. We use the version 2.2 of TensorFlow and in all experiments use the `MirroredStrategy` to distribute the training load between both GPUs. The `MirroredStrategy` is an implementation of Data Parallel synchronous training, where the global batch is divided among the GPUs, which synchronize after each forward pass to keep a common view of parameters.

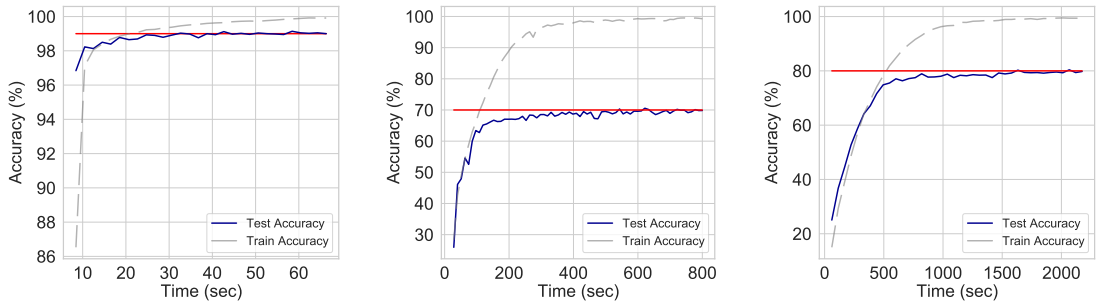
Network	Num. Parameters	Dataset
LeNet5	44K	MNIST
ResNet34	21M	CIFAR10
VGG16	134M	CIFAR10
ResNet32	0.46M	CIFAR10

Table 4.1: Networks and Datasets Used in the Evaluation

Models & Datasets Throughout the experiments we use several networks representing common baselines and a variety of network sizes and architectures. A summary of the models, their size and related datasets used can be found in Table 4.1. We use the LeNet5 [37] as an example of a small network and train it using the MNIST dataset for written character recognition. The MNIST dataset holds 60K train and 10K test images of hand-written numerical characters in black and white, each image of dimensions 28x28.

To test KubeML's performance on bigger and deeper networks, we use the Resnet34[1] network proposed in [21]. This network, with its more than 20 million parameters, should portray better

[1]We use the `torchvision.models` implementation in PyTorch and the one in `https://github.com/raghakot/keras-resnet` for TensorFlow.



(a) LeNet (b) ResNet34 (c) VGG16

Figure 4.1: TensorFlow convergence over time

the benefits of the Local SGD algorithm to reduce communication overhead as a result of less frequent synchronizations. Alongside the ResNet34, we use the CIFAR10 dataset [35], consisting of 50K train and 10K test images with 10 classes. Each image has 3 color channels and a size of 32x32.

Finally, to explore the performance on very big networks we use the VGG16[2] network first described in [62]. With 134 million parameters, this network will show how well Local SGD scales with the number of parameters in terms of communication overhead and accuracy. We also use the CIFAR10 dataset in this case.

Additionally to the experiments comparing KubeML's performance against TensorFlow, we also train the ResNet32[3] network [21] to assess the performance of KubeML on longer training tasks and the effect of the parameter k on the convergence and the final results achieved.

Network	Optimizer	Momentum	Learning Rate	Weight Decay
LeNet5	SGD	0.9	0.01	1*e*-4
ResNet34	SGD	0.9	0.1	1*e*-4
VGG16	SGD	0.9	0.01	5*e*-4
ResNet32	SGD	-	0.1*	1*e*-4

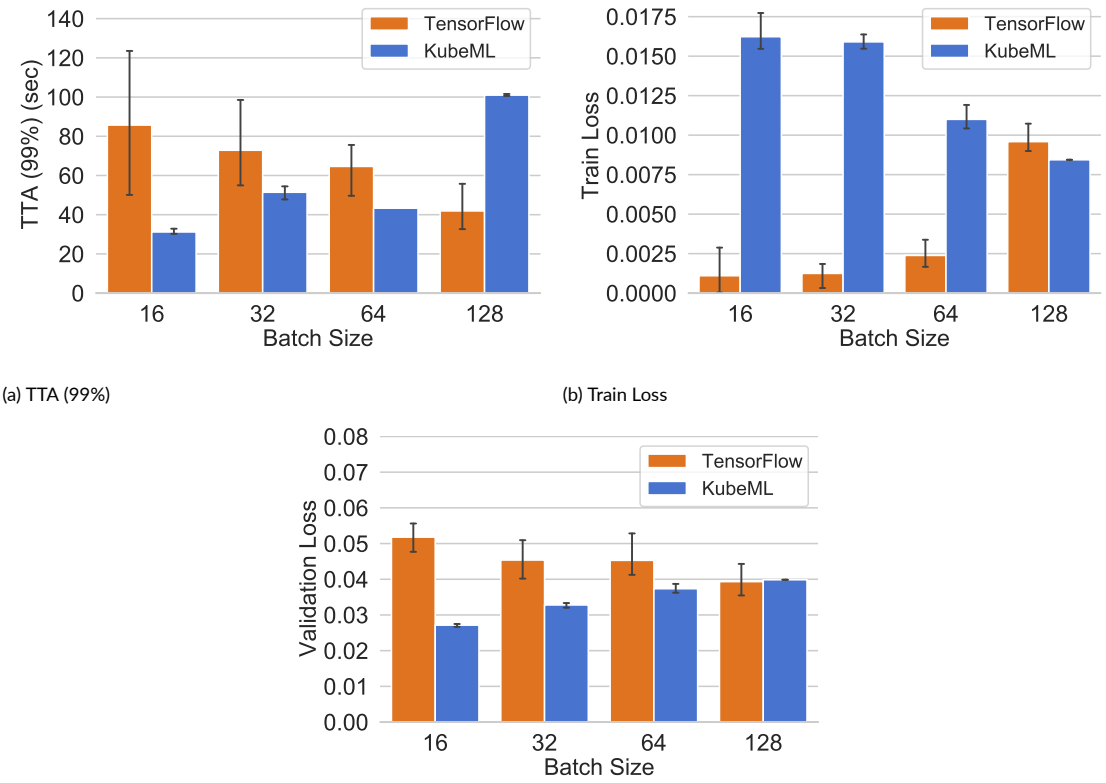
Table 4.2: Networks and Datasets Used in the Evaluation
*The starting LR is 0.1 but we use a Learning Rate Scheduler as in the original paper [21].

Hyperparameters For the comparison with TensorFlow, we want to maintain the configuration as straightforward as possible to avoid differences in implementation between classes in both deep learning frameworks, PyTorch, used in KubeML, and TensorFlow. To assure this, we perform only basic data transformation operations on both systems, such as feature-wise standardization of samples, and keep hyperparameters fixed during training. All our comparisons will be based on the per-worker or local batch (b) for fairness, since KubeML and TensorFlow declare the batch size in different ways.

In TensorFlow, with multiple devices, the provided batch size is interpreted as the global batch. Hence, a batch of 64 using two GPUs will result in each GPU training on 32 datapoints at a time in between synchronization points. In contrast, in KubeML the specified batch size will be used as the local batch, so a batch of 64 means that each worker trains locally on 64 datapoints at a time.

When training the LeNet or VGG16, the learning rate is fixed at 0.01, while for the ResNet34 we fix it at 0.1 and vary the weight decay according to literature. For the training algorithm, we use SGD. An overview of the parameters used can be seen in Table 4.2.

[2]For the PyTorch implementation, we also use the model in `torchvision.models` but changing the last layer to produce 10 outputs rather than 1000. For TensorFlow we follow the description of the paper and make sure both have the exact same number of parameters.
[3]The implementation used is available at `https://github.com/akamaster/pytorch_resnet_cifar10`



(a) TTA (99%)

(b) Train Loss

(c) Validation Loss

Figure 4.2: Training Performance Comparison of LeNet

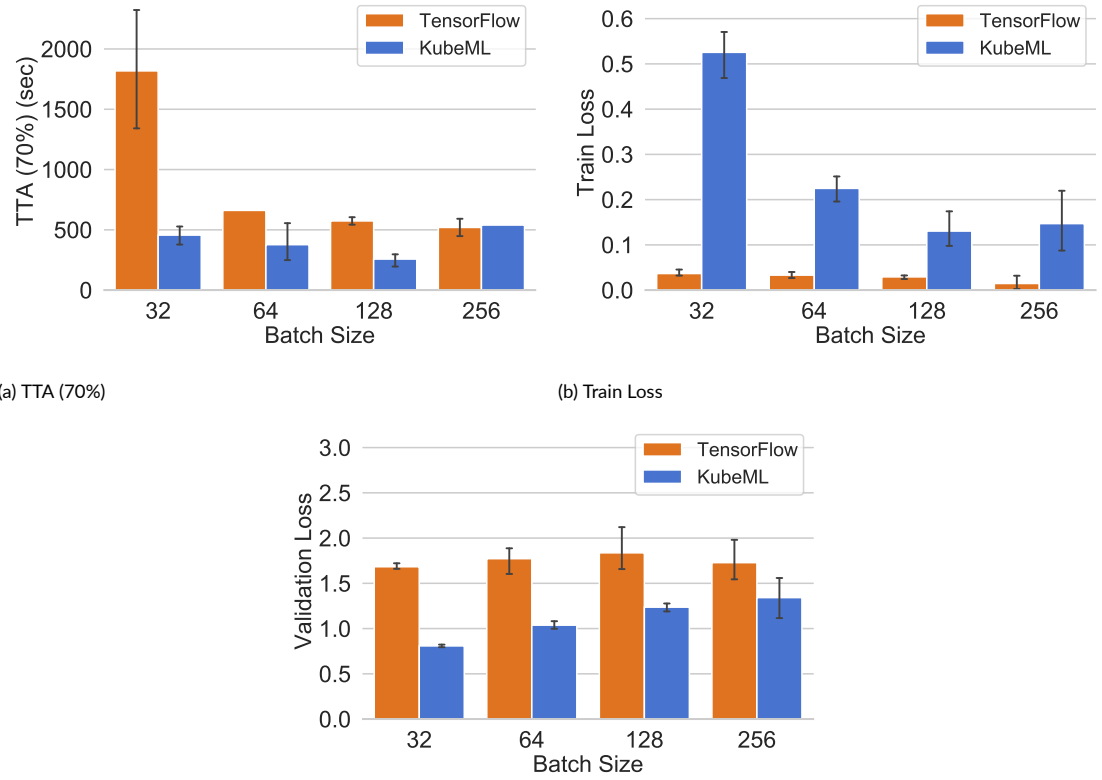
Metrics In our comparison against TensorFlow, our main metric is *Time-To-Accuracy (TTA)*, defined as the time it takes for the validation accuracy to reach a certain amount. To settle on the target accuracy that will serve as objective for the comparison, we train all the networks used as comparison on Tensorflow until consistently reaching a plateau in terms of validation accuracy. We show these results in Figure 4.1, with the baselines for each network being: 99% for LeNet, 70% for the ResNet34 and 80% for VGG16. We also compare both systems by the final train and validation loss reached during training. In terms of efficiency, we record the average usage of GPUs for both systems, and analyze the ability KubeML to extract extra performance from GPU resources by multiplexing tasks on GPU.

4.2. Comparison with TensorFlow

Small Networks In Figure 4.2 we compare the results obtained when training on the LeNet, all batches reported correspond to the local batch size of each worker. For KubeML, we provide the best results achieved with the best parameter combination in terms of parallelism and k.

As a first particularity of the results, all of the optimal parameter combinations for KubeML use a $k = \infty$, meaning that functions train locally for the entire duration of an epoch and only synchronize once before continuing the training process. As can be seen in Figure 4.2a, the performance improvement has a tight relationship with the batch size. With bigger local batch sizes TensorFlow performs better than KubeML in terms of TTA. With smaller batches however, we see that KubeML outperforms TensorFlow and is 1.41x faster with a batch of 32, and 2.75x faster to the target accuracy with a local batch of 16. In these two cases, the best result is achieved with a parallelism of 8, that is, 4 models per GPU, showing that Local SGD is able to converge faster and without a loss in accuracy even with multiple workers scheduled per GPU.

Another relevant insight is the relationship between the train and the validation loss summarized in Figures 4.2b and 4.2c. We can see that even though KubeML consistently performs



(a) TTA (70%)

(b) Train Loss

(c) Validation Loss

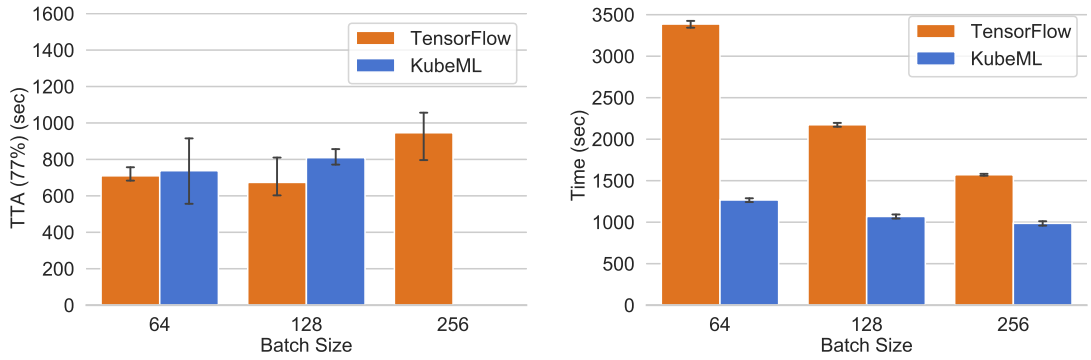
Figure 4.3: Training Performance Comparison of ResNet34

worse that TensorFlow in terms of train loss, it often results in a lower validation loss. This finding could corroborate the findings of [41], where the authors discuss that the local updates of Local SGD inject noise to the training dynamics, resulting in a convergence to flatter minima than traditional SGD. These *flat minima* are characterized by a better generalization than the *sharp minima* reached with other methods [30]. We will explore this in more detail in the following section.

Medium Networks The benefits of using Local SGD should be accentuated when training bigger networks whose communication step takes a considerable amount of time when compared to the computing step, and this is the exact trend observed using the ResNet34 as shown in Figure 4.3. With regards to the TTA, KubeML is consistently equal or better performing than TensorFlow, with the same improvement we saw with the LeNet taking place with small batches. With a local batch of 32, KubeML is 3.98x faster to 70% (Figure 4.3a). Additionally, if we take into consideration only the best results from each system (b = 128 for KubeML and b = 256 for TensorFlow), KubeML is still 2.02x faster to the target accuracy.

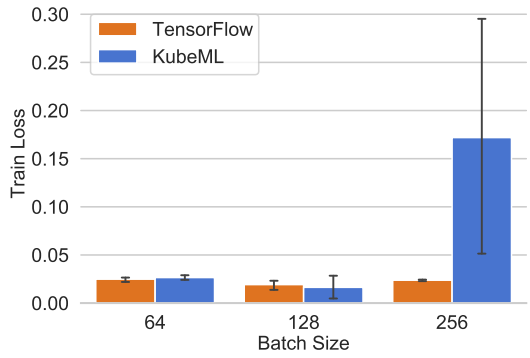
Analyzing the losses of Figures 4.3b and 4.3c we reach the same conclusion as before. KubeML overfits less and generalizes better than TensorFlow, which translates into lower validation loss despite bigger train loss.

Big Networks The weakest point of KubeML and other serverless systems is the communication overhead and latency. In these experiments, TensorFlow uses the `MirroredStrategy` to exchange model gradients. This makes communication between GPUs very fast since it uses NCCL to optimize throughput by using PCIe lanes to efficiently send the parameters. This performance advantage really shows when training on an extremely big network such as the VGG16. Unlike the ResNet, the VGG uses more fully connected layers, translating into a higher parameter count. The first fully connected layer alone has over 102M parameters, which means that

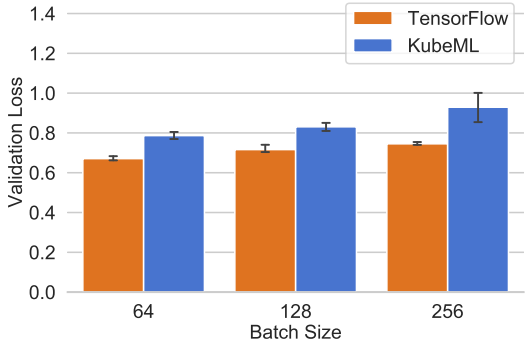


(a) TTA (77%)

(b) Time to complete 40 epochs ($k = \infty$ for KubeML)



(c) Train Loss



(d) Validation Loss

Figure 4.4: Training Performance Comparison of VGG16

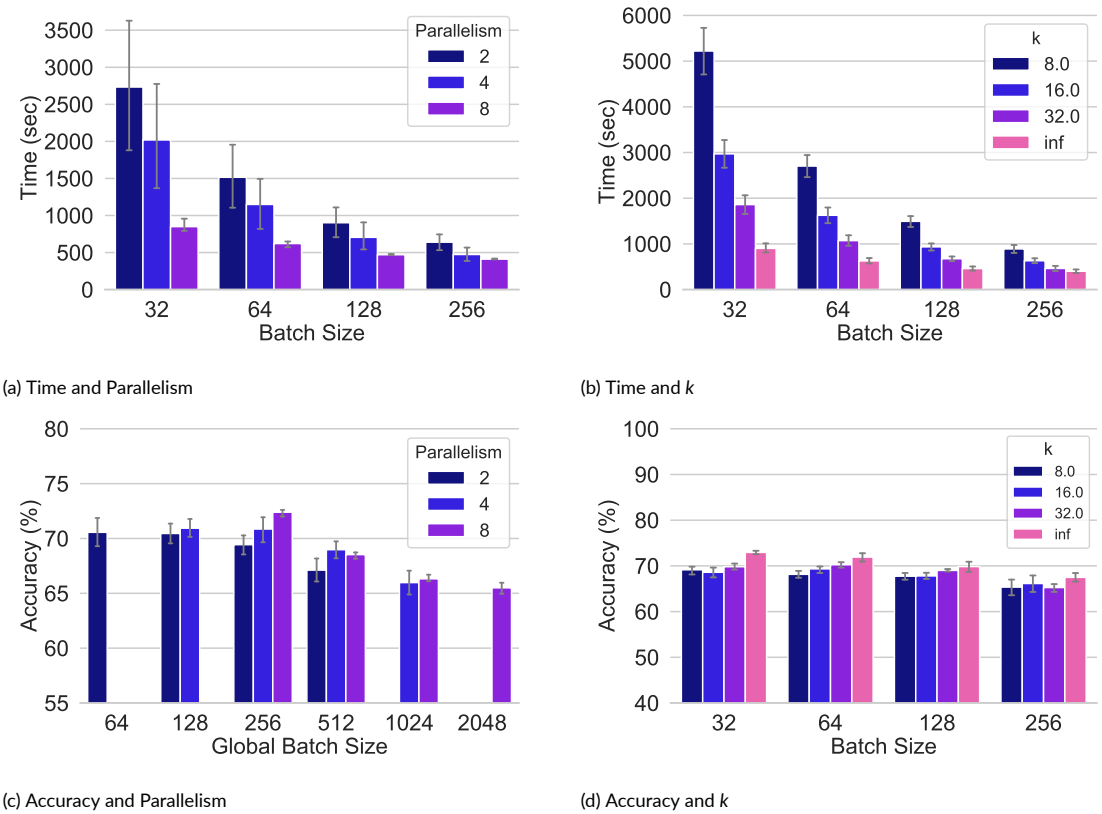
sending it implies sending more than 400MB of data for that layer alone.

We show the results of training the VGG16 network in Figure 4.4. Unlike with the other network sizes, we observe that KubeML is not able to exploit Local SGD in this problem to accelerate convergence. In fact, as we can see in Figure 4.4a, KubeML is not able to reach the baseline accuracy of 80% objective described in Figure 4.1c, so we lower the objective to 77% accuracy. Despite this, we see that TensorFlow is able to achieve the target accuracy faster on average than the best configuration of KubeML.

Looking at Figure 4.4b we see that despite the synchronization steps being lengthier, at almost 3 seconds to synchronize all the layers, KubeML is 1.5x to 2.7x faster to finish 40 epochs, however with such a sparse averaging strategy, the network does not converge as well as seen with previous networks, as we can appreciate in Figures 4.4c and 4.4d. KubeML with the faster spec of $k = \infty$ reports a higher train loss, but is not able to generalize better as indicated by a higher validation loss. We will analyze the results with the VGG16 network further in Section 4.3 to dig deeper in the behavior with bigger networks.

4.3. Speedup Analysis of Hyperparameters

KubeML uses two main hyperparameters apart from the traditional deep learning ones, parallelism and k. Parallelism sets the number of workers that will receive part of the dataset and train a replica of the network in parallel, be it in the same GPU as other workers or in different GPUs across multiple servers. On the other side, k defines the synchronization interval between functions, sync-SGD would be equivalent to running Local SGD with $k = 1$. In this section we analyze the impact of these hyperparameters on the performance regarding the time and accuracy achieved at the end of training. We focus on two networks in particular, the ResNet34 and the VGG16, since the LeNet behaves similarly to the ResNet34 as was seen in the previous section.



(a) Time and Parallelism

(b) Time and k

(c) Accuracy and Parallelism

(d) Accuracy and k

Figure 4.5: Influence of the communication delay K and the parallelism on the resulting time and accuracy. In our experiments, KubeML scales well with the number of workers both in terms of time (showing close to linear speedup) and accuracy. This experiment uses ResNet34.

ResNet34

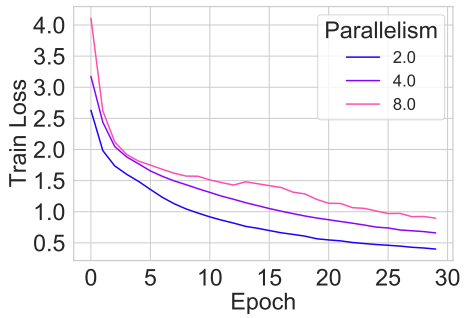
The influence of the hyperparameters for the ResNet34 model is depicted in Figure 4.5. In Figures 4.5a and 4.5b we can see that both the parallelism and the k can provide near-linear speedup in training tasks when increased, with smaller batches obviously benefiting more from the increased parallelism or synchronization period. With bigger parallelism and k KubeML allows for small-batch training, usually preferred for statistical efficiency, while still maintaining the speed improvements of training with larger batches.

However, a distributed training algorithm's performance should not be measured solely based on the speedup resulting from additional workers or sparser synchronization. Speedup without good convergence properties is often not enough to effectively train neural networks in a distributed setting, this is a drawback which affects async-SGD and its variants [69].

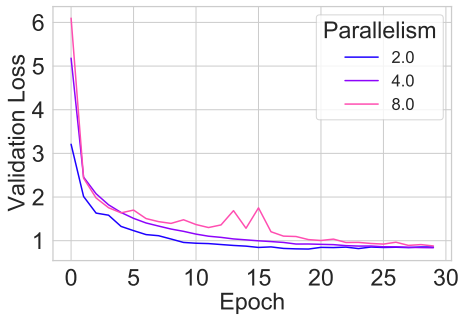
KubeML and its Local SGD algorithm take advantage of the sparse communication and extra workers also in the convergence aspect. As shown in Figure 4.5c, for the same global batch (resulting from multiplying the number of workers time the local batch b), adding extra workers does not seriously degrade the final accuracy reached within a certain number of epochs, and often results in a better generalization. Moreover, in our experiments, a bigger k consistently yields similar or better generalization than more frequent synchronization, as described by Figure 4.5d. This effectively means that KubeML is able to scale out and reduce communication while maintaining the same level of performance as synchronous SGD variants.

In the previous section, we saw that one of the most surprising benefits of using KubeML and Local SGD is the superior generalization performance of the models. KubeML is able to achieve a better validation performance with a sometimes considerably higher train error. Moreover, all of the best results achieved with KubeML and shown in Figures 4.2 and 4.3 where achieved with sparse averaging or $k = \infty$. We study this phenomenon in more detail in Figure 4.6.

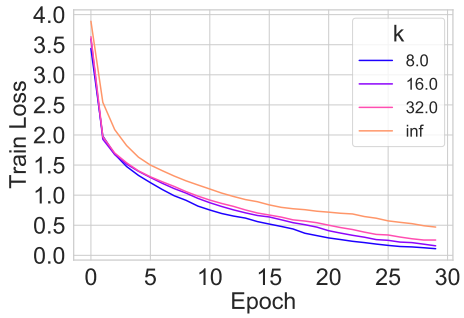
Looking at Figures 4.6a and 4.6b we can see that generally, with higher parallelism the train



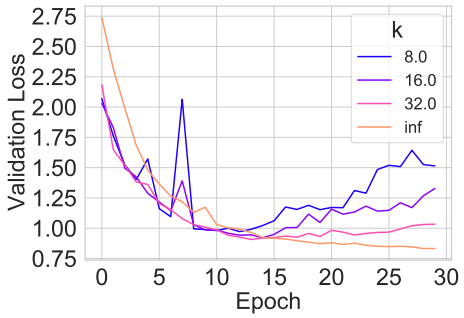
(a) Train Loss varying parallelism, k is fixed at ∞.



(b) Validation Loss varying parallelism, k is fixed at ∞.



(c) Train Loss varying k, parallelism is fixed to 4.



(d) Validation Loss varying k, parallelism is fixed to 4.

Figure 4.6: Progress of the train and validation loss with different levels of parallelism and k. Parallelism does not have a strong effect on generalization, however a larger k prevents overfitting by the network. These plots use the ResNet34 with b = 128

loss tends to be worse than with smaller parallelism. With a higher degree of parallelism, there are more functions that explore their own area of the loss space, leading to a higher inconsistency in model updates. However, despite reporting higher train loss, the validation loss is the same for all parallelisms, with negligible differences.

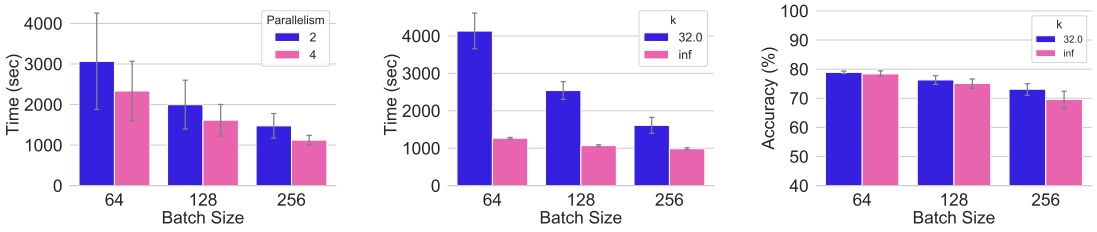
We observe a more drastic difference when analyzing the influence of k on the model generalization. In Figure 4.6c we can see that as the value of k increases, the train loss does too. However, in Figure 4.6d we see the superior generalization performance of sparse averaging. Smaller values of k, after a certain point, start increasing in validation loss, or overfitting to the training data, and the effect is also inversely proportional to the value of k. In fact, sparse averaging is the only setting in which the loss never increases, and overall ends up being the lowest loss overall.

In general, these results confirm that KubeML can exploit speedup both in terms of adding additional workers and reducing communication frequency without degrading performance, and even improving generalization as a result.

VGG16

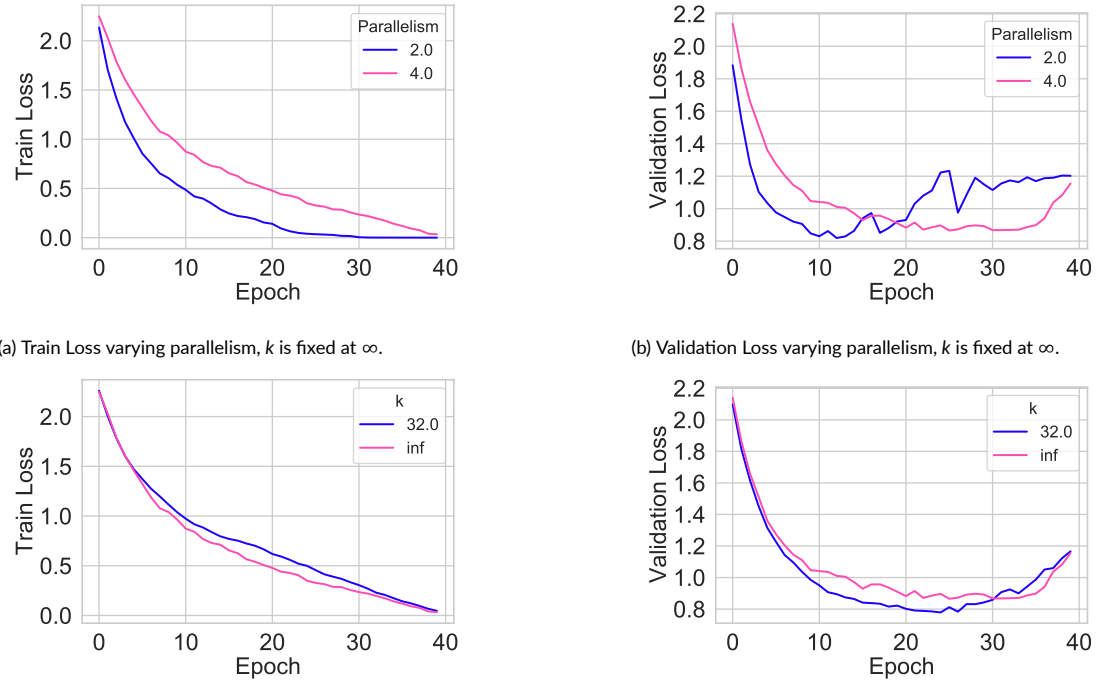
With a considerably higher parameter count, we saw that the VGG16, unlike other networks, was not easily trained using Local SGD to reach better results than with TensorFlow. With the VGG, communication overhead is higher than the ResNet34, and plays a bigger part in the total runtime. We can see that looking at Figure 4.7a. Doubling the number of workers no longer represents as big of an improvement as with the ResNet, but diminishing the synchronization steps provides an even bigger speedup as shown by Figure 4.7b, since communication is the main bottleneck with a network as big as the VGG16.

Also in contrast with the other tested networks, the VGG16 is the only network in which a smaller communication period actually translates into a higher final accuracy, as shown in Figure 4.7c, where averaging every 32 iterations consistently ends up providing a higher accuracy than once per epoch. We cannot clearly state that the increase in parameters is the root cause of the generalization gap. Previous work has been done trying to understand the generalization



(a) Time and Parallelism (b) Time and k (c) Accuracy and Parallelism

Figure 4.7: Influence of the communication delay K and the parallelism on the resulting time and accuracy using the VGG16 network. Unlike with other networks, the VGG no longer benefits with sparse averaging and reports better accuracy with more frequent synchronization. With the VGG16 being a much bigger network, increasing communication frequency greatly affects the time to finish the 40 epochs as shown in (b).



(a) Train Loss varying parallelism, k is fixed at ∞.

(b) Validation Loss varying parallelism, k is fixed at ∞.

(c) Train Loss varying k, parallelism is fixed to 4.

(d) Validation Loss varying k, parallelism is fixed to 4.

Figure 4.8: Progress of the train and validation loss with different levels of parallelism and k using the VGG16 network and $b = 128$. Parallelism or k do not have a strong effect on generalization, but in this case $k = \infty$ no longer behaves best and overfits

behavior and its relationship with the number of parameters in neural networks [76, 47, 48], finding no clear relationship that guarantees a more complex model to overfit more to the data. Also, some generalization analysis of Local SGD [41] shows the superior generalization on a variety of tasks, but the authors do not use models as big as the VGG16.

Looking at the lines showing the progression of the loss in Figure 4.8, we can observe a couple of differences compared to the other networks. As seen in Figures 4.8a and 4.8b, bigger parallelism ends up resulting in the same train loss, but generally results in a better validation loss. Nonetheless, both of the configurations cause the validation loss to increase at the end of training, a behavior that we did not see with the other models.

We see a similar behavior when analyzing the performance with varying k in Figures 4.8c and 4.8d. Unlike with other networks, we see that the best validation error is achieved by the more frequent communication, as was already shown by Figure 4.7. Moreover, we stop seeing the property of sparse averaging which made the validation loss not increase unlike more frequent synchronization, since both show an increase in validation error at the end of the training process.

4.4. Training Resnet32

To assess performance on longer optimization tasks we use a common benchmark using the ResNet32 network on the CIFAR10 dataset. We use a weight decay of $1e^{-4}$ and a local batch of 32 with 4 workers, which translates to the usual global batch of 128 used when training the ResNet32 on the CIFAR10 dataset. We apply the usual transformations to CIFAR10 introduced in [21], during training, which include random cropping with padding and random horizontal flips to the data, before standardizing each of the image channels. It should be noted that adding these transformations to the `KubeDataset` class is trivial, since it is fully interoperable with PyTorch transformations, and allows to set up different transformations during training abd validation. As for the learning rate, we apply the common scheduling practices, we start with a learning rate of 0.1, divide it by 10 at epoch 100 and epoch 150, and finish training at 200 epochs. For this experiment we use SGD without momentum.

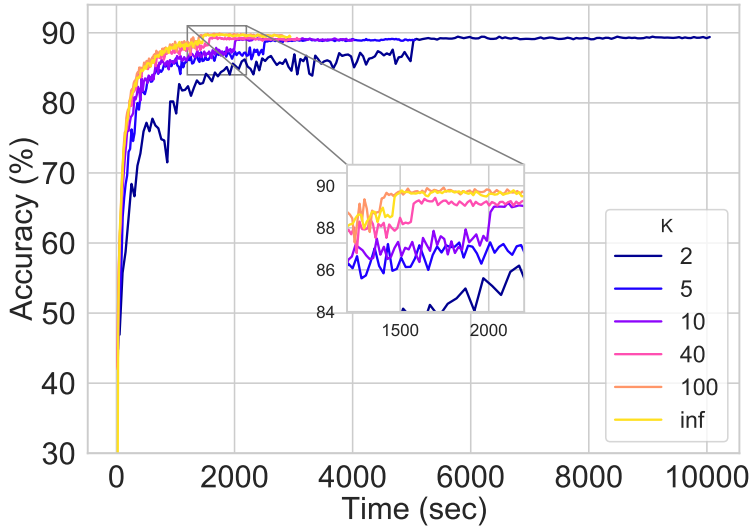


Figure 4.9: Performance on ResNet32 when varying k, the number of forward passes between synchronizations

The results can be seen in Figure 4.9, where we plot the convergence against the time with different values of the synchronization period k. As was discussed in previous sections with small and medium networks, we again see the pattern of better results being obtained when synchronizing once per epoch ($k = \infty$), which not only accelerates convergence but also makes it more stable with fewer oscillations of the accuracy value, and a higher final accuracy of roughly 90%. More frequent synchronization results in a noisier convergence and a lower final accuracy.

These results are in line with best results in literature obtained training the ResNet32 on CIFAR10 with standard SGD, and are also in line with the results we obtained performing the training in a single device and thread, at around 89% accuracy. KubeML does not reach state-of-the-art performance as listed in the original paper [21], since distributed training has to deal with some issues when using momentum and other stateful optimizers, which have been shown to not only accelerate convergence, but render a higher accuracy than standard SGD [73]. We discuss these issues in the coming section.

4.5. The Issue with Optimizer State

In these experiments with ResNet32 we use SGD without momentum, despite common practice when training on a single machine including a momentum value of 0.9. This is due to a common problem when training in a distributed environment using stateful optimizers. Unlike traditional SGD, which simply performs a step towards the direction of the negative gradient based on the statistics of a single iteration, other optimizers have their own state, which holds statistics of updates in previous iterations with the aim of accelerating the convergence. This is the case of more complex optimizers such as Adam [33] and its variants, but also SGD with momentum.

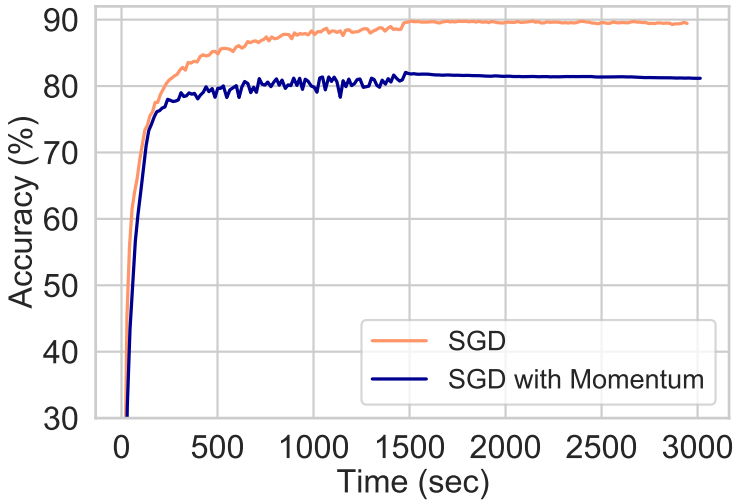


Figure 4.10: Effect of Momentum when Training with SGD, In our case, when using momentum, we opt for the simpler approach of resetting the optimizer state after each model loading step.

With the optimizer having its own state to help convergence, training in a distributed setting has to take care not only of the model weights being carried to be updated and used in the next iteration, but also the optimizer state should be carried to mimic local training. The problem of optimizer state has been studied previously and some solutions have been proposed, such as using block momentum [9, 68] which divides momentum in a global part and a local part that is reset at the start of each iteration. An alternative solution involves averaging the optimizer state in conjunction with the model weight [73].

None of these solutions were, however, studied in a serverless setting. With function containers or VMs not guaranteed to be the same as used in previous epochs, carrying a global momentum and saving the current state for the next iteration proves difficult to manage.

The simplest approach of resetting the optimizer state, or saving the previous state suffer from similar issues. Being at a different point in the loss space after model averaging, it often leads to taking steps in a suboptimal direction. We see this in Figure 4.10. Far from improving convergence, using momentum results in noisier updates after merging the models and a much lower final accuracy, which makes stateful optimizers not advantageous in general without extra optimizations in serverless environments.

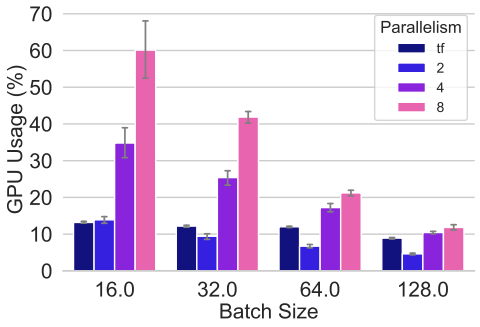
4.6. Resource Utilization

Another important factor when training neural networks on GPUs is the utilization of resources. GPUs are expensive resources, and fitting one model per GPU often results in poor utilization, leading to a waste of both hardware and economic resources [34]. KubeML makes multiple functions share the same GPU by means of the *parallelism* setting to improve resource usage.

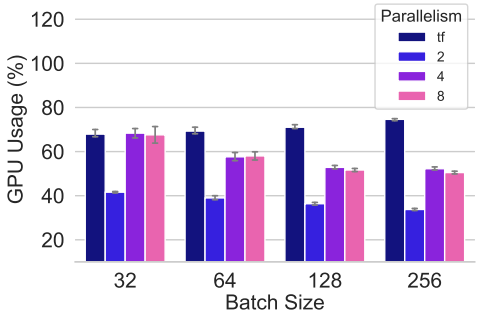
We evaluate how KubeML affects the usage of resources for the networks used in previous sections. The results are plotted in Figure 4.11.

As can be seen, with more workers per GPU, the performance increases for LeNet, especially with smaller batches (Figure 4.11a). This could be due to being easier interleaving small tasks on the GPU, reaching 6x better utilization than TensorFlow. With bigger networks however, although observing that same improvement with small batches, KubeML stays on the same utilization level as TensorFlow (Figures 4.11b and 4.11c). A cause for this could be the added communication time resulting from a bigger model counteracting the throughput gains with more workers, since the addition of extra workers also adds extra communication overhead [64] (the sync time for LeNet is less than 10ms, while for the ResNet34 and VGG16 it takes in the order of a few seconds).

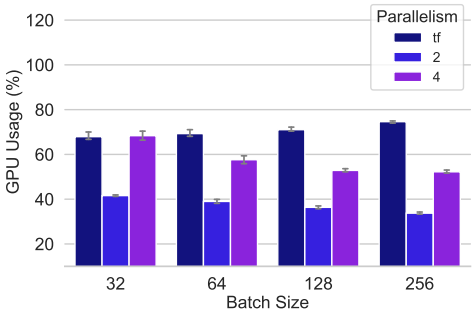
We investigate this issue further in Figure 4.12, where we plot the GPU usage as a function of time, to see more clearly the evolution of GPU usage as well as its variability during the com-



(a) LeNet

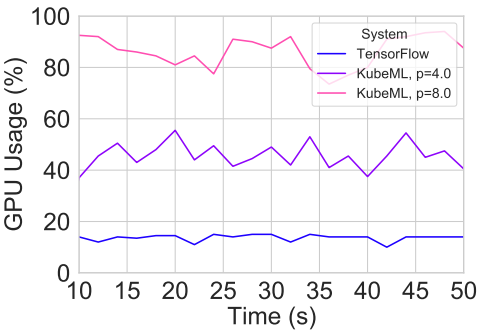


(b) ResNet34

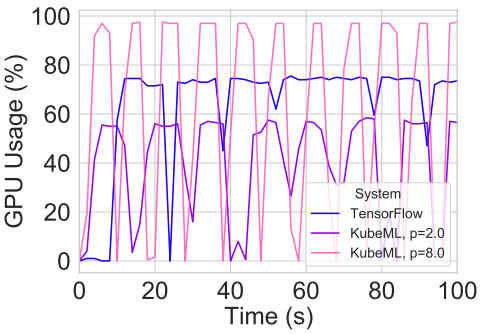


(c) VGG16

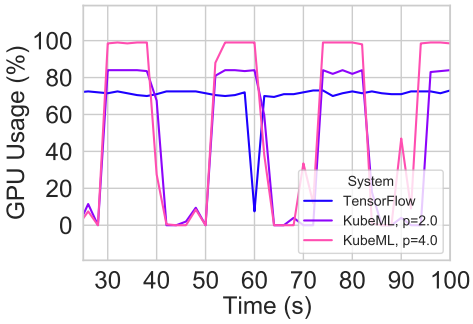
Figure 4.11: GPU utilization comparison between TensorFlow and KubeML with different number of workers



(a) GPU usage of LeNet using local batch $b = 16$



(b) GPU usage of ResNet34 when using local batch $b = 64$



(c) GPU usage of VGG16 when using local batch $b = 64$

Figure 4.12: GPU utilization comparison between TensorFlow and KubeML with different number of workers. In both cases k is fixed at ∞

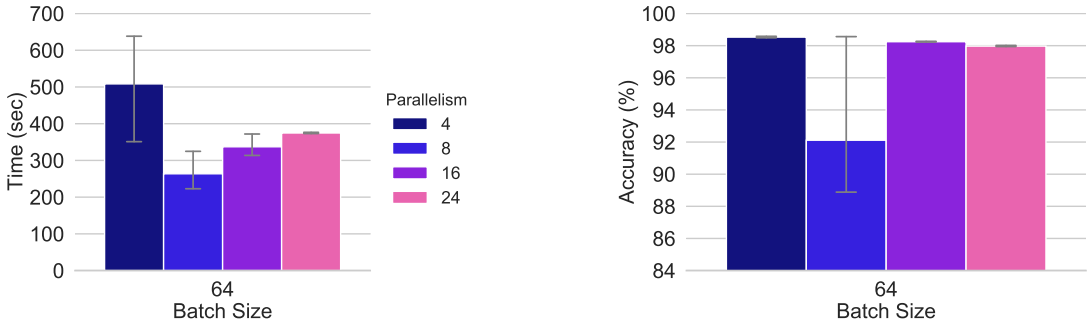
munication periods of the functions. Figure 4.12a confirms that KubeML is able to consistently maintain a higher utilization of resources with small networks such as the LeNet. GPU utilization scales linearly with the number of workers, doubling from the two models used by TensorFlow to KubeML with $p = 4$, and close to doubles again with $p = 8$. With small number of parameters, communication finishes in matter of milliseconds and does not affect the mean utilization.

In Figure 4.12b we can see why in Figure 4.11b the GPU utilization stops improving with more workers. Despite KubeML reaching almost 100% utilization of resources with $P = 8$, the communication cost with extra workers and the sheer size of the network parameters make the synchronization counteract the increase in utilization during the computation phase. TensorFlow, using NCCL AllReduce to synchronize models in the same device, is able to scale better with larger models than the parameter server used in KubeML. Still, this should be tested again with multiple nodes in the cluster, and TensorFlow having to employ a different algorithm to Ring AllReduce, as the results could be more similar in terms of communication cost.

Lastly, we see a similar behavior with the VGG16 model in Figure 4.12c. KubeML is able to achieve a greater maximum utilization of the GPUs compared to TensorFlow, but due to the increased communication overhead, the average utilization remains the same or lower. In the case of VGG16, we can see that also TensoFlow's synchronization steps temporarily reduce the utilization of the GPUs, however the parameter exchange is much faster than in KubeML.

4.7. Multi-Node Training

To show the convenience of KubeML, we deploy it on a 6-node Kubernetes cluster running on Google Cloud Platform (GCP) and train the LeNet for 30 epochs with the same hyperparameters as before. Each node has 8 vCPUs and 8GB memory, and we limit each function to use 1.5 vCPUs. We highlight that the code is the same as was used before, just setting `gpu=False` in Listing 4, and the same code can be used to train on any number of workers, which will be transparently divided among nodes.



(a) Time to complete

(b) Final Accuracy

Figure 4.13: Results after training on a multi-node cluster

We show the results in Figure 4.13, where we see that the network is able to be trained effectively with multiple workers, with limited loss in accuracy (4.13b). In Figure 4.13a we see that from 4 to 8 workers we see linear scaling, but with more workers the results start to get worse. Since we used a small network, there is this tipping point at which stragglers start to become more prevalent and the increasing communication cost starts to dominate the total runtime [64].

4.8. Cost Comparison

Apart from performance, another key aspect that KubeML should cover compared to traditional serverless platforms is deployment cost. In this section we compare the monetary cost of running a serverless application to train a neural network on KubeML and AWS Lambda using two theoretical scenarios. We analyze two different settings with two different training tasks varying in network complexity and communication cost. As in previous sections, we use the LeNet5 to analyze performance on small networks, and the ResNet34 and VGG16 to analyze performance

on bigger networks, whose computational cost should mark the difference between training on CPUs and GPUs.

In our experiments, we used GCP to deploy a Kubernetes cluster and train the networks using CPU nodes, while we used an on-premises server to run the GPU workloads. To compare the pricing in a fair way, we need to translate these testbeds into equivalent cloud resources. In our CPU experiments, we limited each of the running pods to use 1.5 vCPUs, which, with lambda functions receiving between 1 and 2 vCPUs up until 3GB of memory[4], we consider adequate to translate the performance obtained in the GCP CPU experiments to AWS Lambda in terms of runtime. To choose the size of lambda to be used, which also affects the final price, we observe that functions during training use around 1GB and 2.5GB memory each when training the LeNet and ResNet/VGG16 respectively, so we settle for functions with 1.5GB and 3GB memory each.

For the KubeML configuration, we analyze the CPU usage while running the chosen task, and obtain an average CPU usage of 8% CPU for the LeNet and 15% for the ResNet, and memory usage of 10% for both. That translates to an equivalent 5 and 10 CPU cores respectively and 13GB of memory during training. From among the suite of N1 instances in GCP, the only ones suppoting GPUs, we opt for the `n1-standard-8` with 8 vCPUs and 30GB of memory for the LeNet and the `n1-standard-16` with 16 vCPUs and 60GB of memory for the ResNet and VGG. To these instances we incorporate 2 NVIDIA K80 GPUs, which have the same amount of GPU memory as our 2080 Ti's and a slightly higher count of CUDA cores. A summary of the systems used in this comparison are provided in each section.

4.8.1. Small Networks

We first analyze the theoretical results using a small network such as the LeNet. We use the results of training the LeNet on the MNIST dataset for 30 epochs with both systems and compare performance and cost. We choose results with a high parallelism to accelerate training on CPUs, and use $k = \infty$ and batch size of 64 due to the good performance we've observed in other experiments of sparse averaging. As already mentioned, we use the `n1-standard-8` with two Tesla K80 GPUs as the KubeML testbed, and 8x 1.5GB lambdas as the AWS Lambda testbed.

Resource Cost	**KubeML**	**AWS Lambda**
Number of Resources	1	8
Price per invocation	-	$\$2e^{-7}$
Price per duration	\$0.38/h	$\$2.5e^{-6}$/s
GPU Price	2 × \$0.45/h	-
Additional Cost	\$0.1/h	-
Runtime (s)	**69.3**	263.56
Total Cost (10^{-3}\$)	26.76	**5.27**

Table 4.3: Final cost comparison after running 30 epochs training the LeNet on the MNIST dataset

The results of the experiments are summarized in Table 4.3. The lambda price is calculated considering that we invoke a new function at the start of each epoch, amounting to $8 \times 30 = 240$ invocations and a runtime of 263.56 seconds, which is multiplied by the parallelism to obtain the function runtime that will be charged to us: $8 \times 263.56 = 2108.48$ seconds. That equates to a total of \$0.00527 for the training. The invocation cost at this scale is almost negligible at \$0.000048.

In contrast, with KubeML we deploy a single instance with two Nvidia Tesla K80s, and a cluster hourly administrative cost of \$0.1. The usage of GPUs makes KubeML finish much faster than the CPU lambdas, however with the LeNet being a lightweight network, the speedup does not fully compensate for the increased cost of provisioning GPUs, and makes KubeML more expensive for training small networks. It should be highlighted, that for communication to be as efficient in the AWS Lambda system as in KubeML, we would need to provision also a high-throughput storage additionally to the functions, which would add to the overall cost of the lambda deployment.

[4] `https://gist.github.com/saidsef/882647c6589e868b81f0cced4041b132`

4.8.2. Medium Networks

We now compare the performance of both alternatives using a considerably-sized network such as the ResNet34. This network has 477x more parameters than the LeNet, translating into a higher computational cost. Hence, this could boost the benefit of using GPUs further than in the previous experiment. The increase in network and dataset size makes us increase the cost of both deployments. We now use an `n1-standard-16` with twice the CPUs and memory of the previous KubeML deployment, and switch to 3GB lambdas.

In both cases, we train the ResNet34 network on the CIFAR10 dataset for 40 epochs, using again a parallelism of 8 and $k = \infty$, and this time with a batch size of 128.

Resource Cost	**KubeML**	**AWS Lambda**
Number of Resources	1	8
Price per invocation	-	$\$2e^{-7}$
Price per duration	\$0.76/h	$\$5e^{-6}$/s
GPU Price	$2 \times$ \$0.45/h	-
Additional Cost	\$0.1/h	-
Runtime (s)	**470.77**	17,326
Total Cost (\$)	**0.23**	0.69

Table 4.4: Final cost after training for 40 epochs the ResNet34 on the CIFAR10 dataset

The results of the experiments are summarized in Table 4.4. The lambda price is calculated considering that we invoke a new function at the start of each epoch, amounting to $8 \times 40 = 320$ invocations and a runtime of 17,326 seconds, which is multiplied by the parallelism to obtain the function runtime that will be charged to us: $8 \times 17,326 = 138,611$ seconds. That equates to a total of \$0.69 for the training. Again the invocation cost at this scale is negligible at just \$0.00006.

As can be seen, with bigger networks, the use of GPUs makes KubeML finish much faster than the CPU lambdas, which more than compensates for the increase in cost due to GPUs. In the end, KubeML finishes 36x faster and costs 3x less than AWS Lambda.

4.8.3. Big Networks

Lastly, we compare the performance of both alternatives using a big network such as the VGG16. We maintain the same configuration as with the ResNet34, using 3GB lambdas and the same `n1-standard-16` instance given the memory requirements of the network.

In both cases, we train the VGG16 network on the CIFAR10 dataset for 40 epochs, using this time a parallelism of 4 and $k = \infty$, since the VGG uses approximately 4GB of GPU memory, preventing us from fitting more models per GPU. We also use a batch size of 128.

Resource Cost	**KubeML**	**AWS Lambda**
Number of Resources	1	4
Price per invocation	-	$\$2e^{-7}$
Price per duration	\$0.76/h	$\$5e^{-6}$/s
GPU Price	$2 \times$ \$0.45/h	-
Additional Cost	\$0.1/h	-
Runtime (s)	**1035**	38,409
Total Cost (\$)	**0.506**	0.768

Table 4.5: Final cost after training for 40 epochs the ResNet34 on the CIFAR10 dataset

The results of the experiments are summarized in Table 4.5. The total invocations are now $4 \times 40 = 160$ and the runtime runtime of 38,409 seconds, which with 4 functions, translates into $4 \times 38,409 = 153,636$ total billable seconds. That equates to a total of \$0.768 for the training.

Similarly to the previous examples, the usage of GPUs in KubeML shows a great benefit in terms of runtime, with a 37x faster completion of the training. In terms of cost, we also see a reduction, though not as significant as with the ResNet given the reduced parallelism which cuts the Lambda costs in half.

4.9. Conclusion

In this section we have analyzed the performance of KubeML on a wide variety of networks of different sizes and depth. We benchmarked the performance both in terms of convergence speed, final accuracy and resource utilization by comparing it with a state-of-the-art deep learning platform such as TensorFlow. Moreover, we analyze the cost of deploying KubeML in the cloud and compare it with AWS Lambda by performing a theoretical analysis based on the performance on GPU and CPU on different testbeds of all networks.

Overall, we see that KubeML clearly outperforms TensorFlow using small and medium sized networks in terms of TTA, shining especially with small batches, where KubeML is able to improve runtime, generalization and resource usage. In contrast, TensorFlow performs better than KubeML with huge networks such as the VGG16. Even though KubeML is able to reduce total runtime using Local SGD, this does not translate into an increase in accuracy as seen with smaller models. The cause of this decrease in performance is yet to be confirmed, since studies about the generalization of Local SGD do not cover models as big as the VGG16. Moreover, KubeML's lack of low-latency synchronization methods also causes GPU utilization to suffer with bigger models, while TensorFlow is able to take advantage of NCCL and AllReduce to speed up parameter exchanges, KubeML has to rely on slower transmission of data through the network due to the restrictions of serverless architectures. We could see a decrease in communication efficiency in TensorFlow when using multiple servers and having to send the parameters using a technique different to NCCL, which could level the efficiency of both platforms.

Lastly, we compared the cost of running KubeML using a Kubernetes cluster running on a GPU instance on Google Cloud Platform against running the same models on CPU using AWS Lambda. In our theoretical calculations, we reach the conclusion that, despite using costly GPUs, a GPU KubeML setup would still be better both from a cost and a runtime standpoint compared to CPU Lambdas. For smaller network as the LeNet, CPU functions are still able of training the network fast enough to justify the usage of Lambda as it is much cheaper than using a GPU instance.

5

Discussion & Future Work

5.1. Discussion

In this work we have introduced KubeML as a possible alternative for democratizing distributed deep learning by using a serverless architecture. We have adapted the design of KubeML to the restrictions and requirements from the communication standpoint, and measured its performance against state-of-the-art ML systems with good results. We are now ready to summarize our conclusions with regards to the research questions introduced in Chapter 1:

(RQ1) How can we design a serverless system specialized for deep learning tasks?

We have introduced common deep learning practices in Section 2.2 and the downsides of serverless platforms in Section 2.4. From our research we concluded that the key properties of a serverless platform specialized for deep learning should be its ease of use, performance and efficient communication model. With serverless restricting communication between functions, we resorted to a well-known and widely utilized paradigm for distributed ML such as the parameter server architecture. Also learning from previous work from Section 2.5, we opted for a hybrid storage approach, using a flexible but slower storage for datasets, and a faster key-value cache for communicating results between the functions and the train jobs.

(RQ2) How can we take advantage of GPU resources in a serverless environment?

Previous work like [6] and [28] showed that the lack of GPU resources was a key obstacle stopping a wider adoption of serverless in distributed deep learning. Studying the traditional cloud provider options, we found that we would have to rely on providers enabling this extension to additional hardware platforms, so instead we opted for open-source alternatives such as Kubernetes and Fission.

By default, Kubernetes assigns GPUs as exclusive resources, meaning that only one pod can access the GPU at the same time. This limits the parallelism we can incorporate into KubeML, and was already studied in previous work [65]. To make up for this, we configure Nvidia Docker so that we can bypass the restrictive allocation and allow multiple pods to run deep learning jobs concurrently. This results in a higher hardware efficiency as shown in Section 4.6, since we can fit multiple smaller networks that alone would greatly underutilize the GPUs, and extract extra performance from the same hardware.

The addition of GPUs and the ability of multiplexing multiple tasks per GPU allows us to outperform state-of-the-art systems like TensorFlow in terms of performance consistently, and report higher maximum utilization, although suffering to maintain it with bigger networks.

(RQ3) How can we optimize communication to improve latency in a serverless environment?

A popular way of optimizing communication in distributed deep learning is the use of high performance collective communication libraries like MPI [43] and algorithms like AllReduce. However, the restriction imposed by serverless does not allow us to communicate between workers directly. Same goes for using GPU to GPU communication libraries such as NCCL. At each epoch, the physical location of each of the function pods might vary, so we cannot rely on static addressing of resources.

Using the parameter server architecture we overcome the communication problem. We create a DNS entry for each train job and the RedisAI storage so the functions can communicate with them without low-level details. From among the multiple available algorithms, we wanted to focus on synchronous alternatives given the latest trend being using synchronous approaches given their convergence benefits [69]. However, in serverless when training on multiple nodes, sending the weights through the network at each iteration could hinder performance by adding considerable latency. To solve this, we relax communication by using Local SGD, which trains for multiple iterations k before synchronizing model weights, as had already been seen in explained algorithms such as Downpour SGD [11] and EASGD [78].

Later in our experiments in Section 4.3, we found out that against what could be expected, a bigger synchronization period, far from worsening performance, is in many cases beneficial, since it allows for a wider exploration by the workers and consistently improves generalization.

(RQ4) How can we minimize configuration overhead and changes to local code to create deep learning tasks?

We also wanted KubeML to incorporate the benefits of cloud provider serverless options by allowing easy testing and transition into a highly distributed environment. Some previous work had already focused on offering an amicable interface to write serverless applications [7], so we wanted to go in that line by offering a familiar interface that required minimal changes to the local code.

To address this we developed the KubeML Python module. Following the structure of PyTorch classes, KubeML classes can fully interact with the interfaces that the users are used to when creating deep learning jobs. Writing a distributed job no longer requires configuring daemon processes in each of the servers as is the case in TensorFlow, or writing lengthy YAML files that slow the deployment process as in tools like KubeFlow. Users simply define hooks for the different operations done during training, and can reuse the same code to create the `KubeDataset` and `KubeModel` classes.

In addition, the KubeML CLI further eases the deployment process by handling all the complexities in a transparent way, and offering users to start their distributed jobs only with a couple of intuitive commands.

(RQ5) How well does the proposed system perform when compared to state-of-the-art distributed deep learning systems?

In Chapter 4 we compared the performance of KubeML against a state-of-the-art system like TensorFlow. In our experiments, we observe that KubeML is in most cases able to beat TensorFlow in terms of Time-to-Accuracy (TTA) for small and medium networks, seeing an even more pronounced gain when training with small batches. KubeML generalizes better thanks to its Local SGD algorithm, and is able to better utilize resources thanks to scheduling multiple jobs per GPU unlike its rival.

5.2. Future Work

In this work we were able to present a new approach to serverless deep learning, for the first time incorporating techniques such as Local SGD and the usage of GPUs. However, throughout this work we came across areas in which deeper research is needed in order to provide a more well-rounded and holistic solution for serverless deep learning.

1. Despite KubeML being able to incorporate GPUs into serverless, our solution at this point does not provide clever scheduling of GPUs, let alone handling multiple jobs accessing GPUs safely. As we have shown, an effective and fair scheduler for these cases is still an open problem [65]. To implement a clever scheduler like this, we would need to build a custom Kubernetes scheduler that created the serverless pods having GPUs and their resources into account, as well as a custom version of the Fission Executor, which distributed the requests equally between GPUs rather than randomly like it is done now.

2. Also in line with the previous point, KubeML does offer flexible scaling at runtime using the Scheduler, however this also has to be done in a more intelligent way than the reactive policy implemented currently. A new policy should monitor the resources available in real time as done in [65], and scale up and down the tasks accordingly. Still, preventing crashes from over-allocation becomes increasingly difficult with more jobs running concurrently.

3. In terms of performance, we saw in Section 4.6 that once we train bigger networks with more parameters, it is difficult to fully utilize the GPUs by multiplexing tasks since communication becomes so dominant that it makes the GPUs idle for long periods of time. To fix this, we should investigate extra methods to make communication more efficient. From the training algorithm point of view, methods like quantization or sparsification or weights before transmission could help reduce the communication overhead [64]. We could also try a new variation of Local SGD in which we give workers a little more leeway allowing a certain degree of staleness. In the PS architecture, this is beneficial since we could avoid the congestion from all workers communicating at the same time.

4. We would also like to increase the flexibility of both the Go and Python part of the libraries. To enable experimentation with different optimization algorithms like proposed in point 3, we must make the API more flexible and allow common interfaces that make including extra features easy without an in-depth knowledge of the entire system. In the same line, the Python library is mostly addressed towards a certain subset of metrics and tackling mostly supervised learning problems, we should also offer solutions for a wider range of problems.

5. Lastly, we would have liked to perform bigger experiments with potentially tens or hundreds of GPU nodes to truly analyze the scaling properties of KubeML and the PS architecture. This could also help in coming up with new points that could be improved in future work.

Bibliography

[1] Martín Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] Ahsan Ali et al. “BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching”. en. In: (), p. 15.

[3] A. Bhattacharjee et al. “BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services”. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. June 2019, pp. 23–33. DOI: 10.1109/IC2E.2019.00-10.

[4] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. In: *CoRR* abs/1604.07316 (2016). arXiv: 1604.07316. URL: http://arxiv.org/abs/1604.07316.

[5] *Caffe2*. 2019. URL: https://caffe2.ai/.

[6] Joao Carreira et al. “A case for serverless machine learning”. In: *Workshop on Systems for ML and Open Source Software at NeurIPS*. Vol. 2018. 2018.

[7] Joao Carreira et al. “Cirrus: A serverless framework for end-to-end ml workflows”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 13–24.

[8] Jianmin Chen et al. “Revisiting Distributed Synchronous SGD”. In: *arXiv:1604.00981 [cs]* (Mar. 2017). arXiv: 1604.00981. URL: http://arxiv.org/abs/1604.00981 (visited on 11/18/2020).

[9] Kai Chen and Qiang Huo. “Scalable training of deep learning machines by incremental block training with intra-block parallel optimization and blockwise model-update filtering”. In: *2016 ieee international conference on acoustics, speech and signal processing (icassp)*. IEEE. 2016, pp. 5880–5884.

[10] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[11] Jeffrey Dean et al. “Large scale distributed deep networks”. In: (2012).

[12] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization.” In: *Journal of machine learning research* 12.7 (2011).

[13] Sanghamitra Dutta et al. “Slow and Stale Gradients Can Win the Race: Error-Runtime Tradeoffs in Distributed SGD”. In: *arXiv:1803.01113 [cs, stat]* (May 2018). arXiv: 1803.01113. URL: http://arxiv.org/abs/1803.01113 (visited on 12/22/2020).

[14] Simon Eismann et al. “A Review of Serverless Use Cases and their Characteristics”. In: *arXiv:2008.11110 [cs]* (Aug. 2020). arXiv: 2008.11110. URL: http://arxiv.org/abs/2008.11110 (visited on 10/29/2020).

[15] *Facebook Gloo*. 2017. URL: https://github.com/facebookincubator/gloo.

[16] Lang Feng et al. “Exploring serverless computing for neural network training”. In: *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE. 2018, pp. 334–341.

[17] *Fission*. 2021. URL: https://fission.io/.

[18] Geoffrey C Fox et al. “Status of serverless computing and function-as-a-service (faas) in industry and research”. In: *arXiv preprint arXiv:1708.08028* (2017).

[19] Farzin Haddadpour et al. “Local SGD with Periodic Averaging: Tighter Analysis and Adaptive Synchronization”. en. In: *arXiv:1910.13598 [cs, stat]* (May 2020). arXiv: 1910.13598. URL: http://arxiv.org/abs/1910.13598 (visited on 04/26/2021).

[20] Farzin Haddadpour et al. “Trading Redundancy for Communication: Speeding up Distributed SGD for Non-convex Optimization”. en. In: *convex Optimization* (), p. 10.

[21] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[22] Joseph M. Hellerstein et al. “Serverless Computing: One Step Forward, Two Steps Back”. en. In: *arXiv:1812.03651 [cs]* (Dec. 2018). arXiv: 1812.03651. URL: http://arxiv.org/abs/1812.03651 (visited on 12/22/2020).

[23] Qirong Ho et al. “More effective distributed ml via a stale synchronous parallel parameter server”. In: *Advances in neural information processing systems* 2013 (2013), p. 1223.

[24] Yanping Huang et al. “Gpipe: Efficient training of giant neural networks using pipeline parallelism”. In: *arXiv preprint arXiv:1811.06965* (2018).

[25] Amazon Inc. *AWS Lambda Pricing*. 2021. URL: https://aws.amazon.com/es/lambda/pricing/.

[26] V. Ishakian, V. Muthusamy, and A. Slominski. “Serving Deep Learning Models in a Serverless Platform”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Apr. 2018, pp. 257–262. DOI: 10.1109/IC2E.2018.00052.

[27] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[28] Jiawei Jiang et al. “Towards Demystifying Serverless Machine Learning Training”. In: (2021).

[29] Eric Jonas et al. “Occupy the cloud: Distributed computing for the 99%”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 445–451.

[30] Nitish Shirish Keskar et al. “On large-batch training for deep learning: Generalization gap and sharp minima”. In: *arXiv preprint arXiv:1609.04836* (2016).

[31] Asif Khan. “Key characteristics of a container orchestration platform to enable a modern application”. In: *IEEE cloud Computing* 4.5 (2017), pp. 42–48.

[32] Jaewook Kim et al. “GPU enabled serverless computing framework”. In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 533–540.

[33] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *CoRR* abs/1412.6980 (2015).

[34] Alexandros Koliousis et al. “CROSSBOW: scaling deep learning with small batch sizes on multi-gpu servers”. In: *arXiv preprint arXiv:1901.02244* (2019).

[35] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[36] *Kubeless*. 2021. URL: https://kubeless.io/.

[37] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[38] Mu Li et al. “Communication Efficient Distributed Machine Learning with the Parameter Server.” In: *NIPS*. Vol. 2. 2014, pp. 1–4.

[39] Mu Li et al. “Parameter server for distributed machine learning”. In: *Big Learning NIPS Workshop*. Vol. 6. 2013, p. 2.

[40] Mu Li et al. “Scaling distributed machine learning with the parameter server”. In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 583–598.

[41] Tao Lin et al. “Don’t Use Large Mini-Batches, Use Local SGD”. In: *arXiv:1808.07217 [cs, stat]* (Feb. 2020). arXiv: 1808.07217. URL: http://arxiv.org/abs/1808.07217 (visited on 12/22/2020).

[42] Ying Mao et al. “Speculative Container Scheduling for Deep Learning Applications in a Kubernetes Cluster”. In: *arXiv:2010.11307 [cs]* (Oct. 2020). arXiv: 2010.11307. URL: http://arxiv.org/abs/2010.11307 (visited on 10/29/2020).

[43] *Message Passing Interface (MPI)*. 2020. URL: https://www.mpi-forum.org/.

[44] Sunil Kumar Mohanty, Gopika Premsankar, and Mario di Francesco. "An Evaluation of Open Source Serverless Computing Frameworks". en. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Nicosia: IEEE, Dec. 2018, pp. 115–120. ISBN: 978-1-5386-7899-2. DOI: 10.1109/CloudCom2018.2018.00033. URL: https://ieeexplore.ieee.org/document/8591002/ (visited on 11/09/2020).

[45] Diana M Naranjo et al. "Accelerated serverless computing based on GPU virtualization". In: *Journal of Parallel and Distributed Computing* 139 (2020), pp. 32–42.

[46] Deepak Narayanan et al. "PipeDream: generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 1–15.

[47] Behnam Neyshabur et al. "Exploring generalization in deep learning". In: *arXiv preprint arXiv:1706.08947* (2017).

[48] Roman Novak et al. "Sensitivity and generalization in neural networks: an empirical study". In: *arXiv preprint arXiv:1802.08760* (2018).

[49] Beng Chin Ooi et al. "SINGA: A distributed deep learning platform". In: *Proceedings of the 23rd ACM international conference on Multimedia*. 2015, pp. 685–688.

[50] *OpenFaas*. 2021. URL: https://www.openfaas.com/.

[51] *OpenWhisk*. 2016. URL: https://openwhisk.apache.org/.

[52] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[53] Yanghua Peng et al. "Optimus: an efficient dynamic resource scheduler for deep learning clusters". en. In: *Proceedings of the Thirteenth EuroSys Conference*. Porto Portugal: ACM, Apr. 2018, pp. 1–14. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190517. URL: https://dl.acm.org/doi/10.1145/3190508.3190517 (visited on 10/29/2020).

[54] Boris T Polyak. "New stochastic approximation type procedures". In: *Automat. i Telemekh* 7.98-107 (1990), p. 2.

[55] Boris T Polyak and Anatoli B Juditsky. "Acceleration of stochastic approximation by averaging". In: *SIAM journal on control and optimization* 30.4 (1992), pp. 838–855.

[56] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, fast and slow: Scalable analytics on serverless infrastructure". In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 193–206.

[57] Maithra Raghu et al. "On the expressive power of deep neural networks". In: *international conference on machine learning*. PMLR. 2017, pp. 2847–2854.

[58] Benjamin Recht et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". en. In: (), p. 9.

[59] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[60] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[61] Vaishaal Shankar et al. "Numpywren: Serverless linear algebra". In: *arXiv preprint arXiv:1810.09679* (2018).

[62] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[63] Sebastian U. Stich. "Local SGD Converges Fast and Communicates Little". In: *arXiv:1805.09767 [cs, math]* (May 2019). arXiv: 1805.09767. URL: http://arxiv.org/abs/1805.09767 (visited on 12/22/2020).

[64] Zhenheng Tang et al. "Communication-Efficient Distributed Deep Learning: A Comprehensive Survey". en. In: *arXiv:2003.06307 [cs, eess]* (Mar. 2020). arXiv: 2003.06307. URL: http://arxiv.org/abs/2003.06307 (visited on 04/26/2021).

[65] Prashanth Thinakaran et al. "Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters". In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2019, pp. 1–13.

[66] Joost Verbraeken et al. “A Survey on Distributed Machine Learning”. In: *ACM Comput. Surv.* 53.2 (Mar. 2020). ISSN: 0360-0300. DOI: 10.1145/3377454. URL: https://doi.org/10.1145/3377454.

[67] Hao Wang, Di Niu, and Baochun Li. “Distributed machine learning with a serverless architecture”. In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE. 2019, pp. 1288–1296.

[68] Jianyu Wang and Gauri Joshi. “Adaptive Communication Strategies to Achieve the Best Error-Runtime Trade-off in Local-Update SGD”. en. In: *arXiv:1810.08313 [cs, stat]* (Mar. 2019). arXiv: 1810.08313. URL: http://arxiv.org/abs/1810.08313 (visited on 04/26/2021).

[69] Arissa Wongpanich, Yang You, and James Demmel. “Rethinking the Value of Asynchronous Solvers for Distributed Deep Learning”. en. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. Fukuoka Japan: ACM, Jan. 2020, pp. 52–60. ISBN: 978-1-4503-7236-7. DOI: 10.1145/3368474.3368498. URL: https://dl.acm.org/doi/10.1145/3368474.3368498 (visited on 03/25/2021).

[70] Yuping Xing and Yongzhao Zhan. “Virtualization and cloud computing”. In: *Future Wireless Networks and Information Systems*. Springer, 2012, pp. 305–312.

[71] Fei Xu et al. “\lambda DNN: Achieving Predictable Distributed DNN Training with Serverless Architectures”. In: *IEEE Transactions on Computers* (2021).

[72] Gingfung Yeung et al. “Towards {GPU} Utilization Prediction for Cloud Deep Learning”. In: *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

[73] Hao Yu, Rong Jin, and Sen Yang. “On the Linear Speedup Analysis of Communication Efficient Momentum SGD for Distributed Non-Convex Optimization”. en. In: *Convex Optimization* (), p. 10.

[74] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai. 2020.

[75] Chengliang Zhang et al. “Stay Fresh: Speculative Synchronization for Fast Distributed Machine Learning”. en. In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. Vienna: IEEE, July 2018, pp. 99–109. ISBN: 978-1-5386-6871-9. DOI: 10.1109/ICDCS.2018.00020. URL: https://ieeexplore.ieee.org/document/8416283/ (visited on 03/25/2021).

[76] Chiyuan Zhang et al. “Understanding deep learning requires rethinking generalization”. In: *arXiv preprint arXiv:1611.03530* (2016).

[77] Jian Zhang et al. “Parallel SGD: When does averaging help?” In: *arXiv:1606.07365 [cs, stat]* (June 2016). arXiv: 1606.07365. URL: http://arxiv.org/abs/1606.07365 (visited on 12/21/2020).

[78] Sixin Zhang, Anna Choromanska, and Yann LeCun. “Deep learning with elastic averaging SGD”. In: *arXiv preprint arXiv:1412.6651* (2014).

[79] Wei Zhang et al. “Staleness-Aware Async-SGD for Distributed Deep Learning”. en. In: (), p. 7.

[80] Fan Zhou and Guojing Cong. “On the convergence properties of a K-step averaging stochastic gradient descent algorithm for nonconvex optimization”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (July 2018). arXiv: 1708.01012, pp. 3219–3227. DOI: 10.24963/ijcai.2018/447. URL: http://arxiv.org/abs/1708.01012 (visited on 12/22/2020).

[81] Martin Zinkevich et al. “Parallelized Stochastic Gradient Descent”. en. In: (), p. 9.