THESIS

# TOWARDS FEDERATED LEARNING OVER LARGE-SCALE STREAMING DATA

Submitted by

Aaron Pereira

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2020

Master's Committee:

Advisor: Dr. Sangmi Pallickara

Dr. Shrideep Pallickara
Dr. Sammy Zahran

# ABSTRACT

## TOWARDS FEDERATED LEARNING OVER LARGE-SCALE STREAMING DATA

Distributed Stream Processing Engines (DSPEs) have seen significant deployment growth along with an increase in streaming data sources such as sensor networks. These DSPEs enable processing large amounts of streaming data in a cluster of commodity machines to extract knowledge and insights in real-time. Due to fluctuating data arrival rates in real-world applications, modern DSPEs often provide auto-scaling. However, the existing designs of advanced analytical frameworks are not effectively aligned with scalable streaming computing environments. We have designed and developed ORCA, a federated learning architecture that supports the training of traditional Artificial Neural Networks as well as Convolutional Neural Networks and Long Short-term Memory Network based models while ensuring resiliency during scaling. ORCA also introduces dynamic adjustment of the 'elasticity' hyper-parameter for rescaled computing environments. We estimate this elasticity hyper-parameter using reinforcement learning. Our empirical benchmarks show that ORCA is capable of achieving an MSE of 0.038 over real-world streaming datasets.

# ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sangmi Pallickara, for her constant guidance and encouragement, without which this thesis work would not be possible. I would also like to thank my committee members, Dr. Shrideep Pallickara and Dr. Sammy Zahran, for their constructive advice and helpful tips. I would also like to thank Mountain Data Group and Dell EMC for providing me with the opportunity to gain industry experience as well as providing me with industry data to evaluate several sections of this thesis. Special thanks to my research lab-mate, Kartik Khurana, for his constant insight and suggestions. I would also like to thank my parents, James and Maria Pereira, and my best friend Manjiri Shete, for their constant support throughout the course of my Master's degree.

# DEDICATION

*I would like to dedicate this thesis to my Mom and Dad.*

## TABLE OF CONTENTS

# LIST OF FIGURES

# Chapter 1

# Introduction

Major advances in computational hardware have resulted in a tremendous increase in the scale at which neural networks are being used today. These advances are enabling researchers to push further the boundaries of what is possible using neural networks. Deep learning, which consists of neural networks with a large number of hidden layers, continuously seems to outperform traditional machine learning algorithms that were previously considered state-of-the-art. Alexnet [1] and ResNet [2] are prime examples of this. However, the data traditionally used to train these deep learning networks underwent a massive transformation in recent times. With the advent of *'smart'* appliances, the user demand and amount of data produced by these appliances are ever-increasing. Most of the data collected by these devices is streamed to central servers for processing. Big data processing frameworks like Apache Hadoop [3] and Apache Spark [4] gained widespread popularity for batch processing of data powered by a few specifically developed machine learning libraries [5]. Performing deep learning in a distributed setting poses a multitude of synchronization, communication, and computation problems. Federated Learning [6] is a novel approach that tries to address these issues. However, such *'smart'* sensor data rarely exists in batched forms. These sensors usually report time-separated observations, often separated by milliseconds. This nature of streaming data sources gave rise to the need for truly distributed stream processing engines with high throughput and low latency.

Apache Storm [7] is a real-time distributed computation engine that is suitable for real-time processing of boundless streams of data. It provides support to define computational topologies that can assist with the sequential processing of stream data. However, Apache storm does not provide native support for machine learning. Furthermore, once a topological structure is defined, it cannot be modified. Hence, Storm is not capable of reacting to a change in velocity of the data stream. If we go ahead with the assumption that Storm can scale its topological structure in response to a change in stream velocity, designing and implementing a deep learning architecture that is capable of scaling itself based on a change in the underlying Storm topology structure, is a non-trivial problem. Ideally, a change in stream velocity should trigger a change in topology structure and the deep learning architecture structure while resisting a significant drop in throughput and an increase in latency.

To address the challenges mentioned above, we have developed ORCA, an adaptive and scalable deep learning framework built on top of Apache Storm. ORCA is based on Federated Learning principles, which roughly consist of multiple worker nodes, which process data in a data-parallel manner and one single centralized parameter server. Learning performed on the workers is periodically aggregated by the parameter server. ORCA is unaffected by worker nodes being added or removed on-the-fly, which helps it achieve high throughput and low latency.

## 1.1 Research Questions

This work explores the following research questions:

- **[RQ1]** How can we deploy a federated learning model on a large scale DSPE with the real-time streaming data?
- **[RQ2]** How can a federated model with real-time streaming data be trained effectively over an auto-scalable DSPE? Can we tune hyper-parameters of the model for the changing number of nodes?

## 1.2 Overview of Approach

Storm's topological structure consists of spouts (serve as data sources and are responsible for sending data to the other components) and bolts (the actual units of execution) placed one after the other in a particular order based on the computations that have to be performed [8]. At a minimum, ORCA will create one spout that would be responsible for sending data into the topology and one processing bolt that is responsible for performing deep learning tasks in parallel. ORCA uses DL4J [9] as its underlying deep learning processing engine which takes care of creating and instantiating multi-layer neural networks and provides APIs to configure, train, and test them. ORCA also has a single centralized parameter server that is implemented as a Django web application [10]. ORCA also contains an API to extract and restore model weights. ORCA periodically extracts these weights and sends them to the parameter server where they are aggregated based on one of ORCA's predefined federated learning strategies. ORCA contains two strategies out of the box, but additional strategies can be user-defined too. If a worker is newly provisioned, it obtains the latest weights from the parameter server and instantiates itself using these weights. ORCA also contains a reinforcement learning module. This module aims to improve model accuracy by tuning the 'Elasticity' hyper-parameter used by the Elastic Averaging Stochastic Gradient Descent [11] strategies. This module is implemented with the help of the parameter server using TensorFlow [12]. All communication between the workers and the parameter server is performed using Rest-API calls and with the help of Apache's HTTPClient. ORCA also contains API calls to track accuracy metrics for the underlying DL4J model periodically.

## 1.3 Research Contributions

Through this research, we present our approach to designing a distributed deep learning architecture for streaming workflows. This approach also ensures that the architecture is immune to sudden changes in the underlying topology structure. It ensures that throughput, latency, and model accuracy remain mostly unaffected.

(1) We have deployed a distributed federated learning architecture for streaming workflows. Our architecture is built specifically to test and evaluate performance on highly volatile topology structures where worker nodes and data arrival rates continuously change.

(2) We have implemented two federated learning strategies and have also tested their performance on highly volatile topology structures using the popular MNIST and CIFAR-10 datasets. In addition to this, we have also evaluated their performance on industry data provided to us by Dell EMC and Mountain Data Group.

(3) Our system predicts the 'Elasticity' hyper-parameter used in the two federated learning strategies in a bid to effectively reduce convergence time taken by the architecture.

(4) The architecture is built on Apache Spark with the help of DL4J and Django. Users also have the provision for designing custom federated learning strategies or deep learning models as well as evaluating their performance.

## 1.4 Organization

The organization of the rest of this document is as follows. Chapter 2 describes work that is similar or is a necessary pre-requisite to our approach. Chapter 3 describes ORCA's design choices as well as provides a high-level overview of ORCA's structure and chapter 4 further builds upon this and provides more specific implementation details. Chapter 5 goes over our experiments and tries to support answers to our main research questions. Lastly, our conclusions are described in 6.

# Chapter 2

# Related Work

## 2.1 Federated Learning

Stream process engines [13–15] have been used in the context of real time sensor data processing [16, 17] and multimedia processing [18]. Optimal stream scheduling is NP-Hard; algorithms such as the prediction rings algorithm [19] account for the interference between computations to ensure collocation of computations to ensure high throughput. There have also been several efforts to manage voluminous scientific data [20–22]. Several of these efforts have focused on effective metadata management [23] and subsetting [24]. In cases where the data has a geospatial component, the spatial dimension is used to manage storage workloads [25–27], support queries [28–30], perform anomaly detection [31], and leverage sketching [32–34] to reduce data volumes. Efforts have also focused on visualization [35] and building models [36] at scale. Distributing deep learning across a cluster of computers while computing and updating gradients in parallel is an extremely non-trivial process. Stochastic gradient descent or *'SGD'* [37] involves a computationally expensive backpropagation procedure. Attempts to distribute computations across a cluster primarily fall into one of two categories, *data parallelism* and *model parallelism* [38]. Model parallelism is ideal for extensive networks as the individual neurons that constitute the network are stored across multiple nodes. However, our problem setting makes the vital assumption that the underlying DSPE is scalable. This means that whenever a new node is added to the system, a reconfiguration of the distribution of neurons would be required, which would pose a significant computational overhead in addition to a distribution policy dilemma. Hence, Model parallel approaches would not seem to be a good fit for our problem setting. Hence, we restricted our literature study to data-parallel approaches. Data-parallel approaches are better suited to smaller networks as, at any given time, each worker is required to contain a copy of the entire model within itself. Hence, larger models would require greater computational resources per node. In this approach, different workers perform their gradient descent and periodically aggregate their learning based on certain pre-defined policies. This approach seems to be the best fit for our problem setting. ’Federated Learning’ is a category of data-parallel approaches that rely on centralized aggregation of learning on a central node, mostly referred to as a ’Parameter Server.’ Figure 2.1 shows a basic overview of a Federated Learning approach. It is also important to note that federated learning techniques could also vulnerable to differential attacks [39] and their security concerns need to analyzed in further detail. Secure aggregation techniques [40] have also been implemented to tackle privacy concerns of the aggregated gradient.

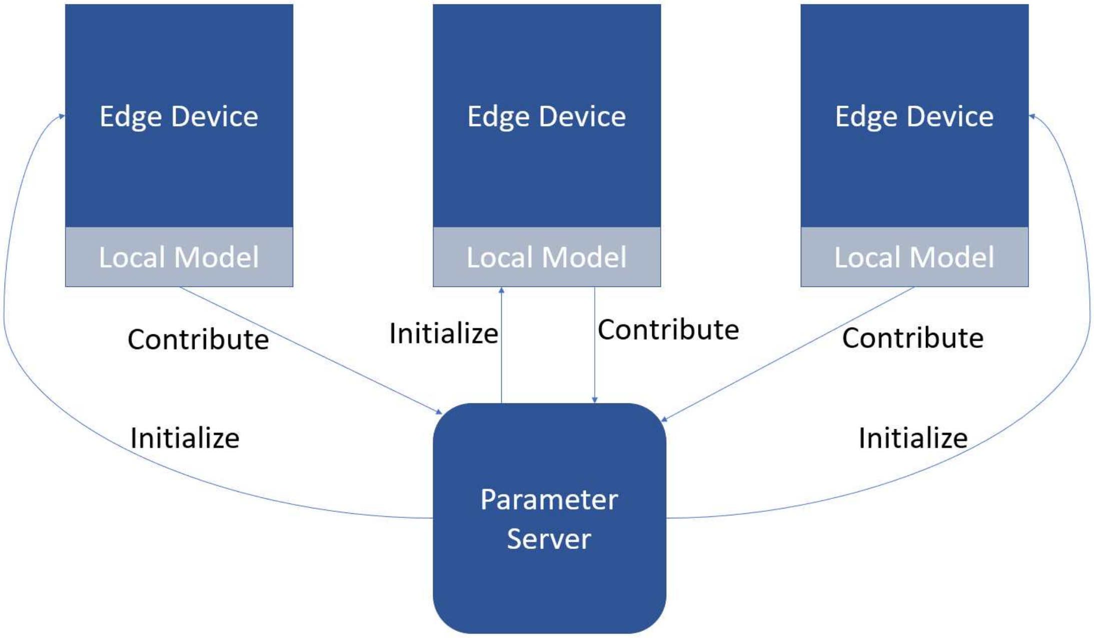

**Figure 2.1:** Overview of Federated Learning

The research conducted as part of [6] coined the term ’Federated Learning.’ This paper introduced an algorithm named ’FedAvg’ and compared it to the traditional federated learning approach named ’FedSGD.’ The primary difference between the two is that FedAvg uses multiple local training batches on worker nodes before it aggregates data as compared to FedSGD, which aggregates after a single training pass. This paper also experimented with IID and non-IID data distribution and its implications on training. The ’FedAvg’ algorithm focused on reducing the communication overhead by increasing the number of training passes a worker made over its local dataset and analyzed the same. The number of contributing nodes, along with batch sizes for training data, were also experimented with. Experiments were performed on CIFAR-10, MNIST, and LSTMs for next-word predictions. The research presented in [41] also provides an overview of the possible techniques that are currently widely used as part of Federated Learning.

*Distbelif* [42] focuses on deep distributed learning via data and model parallelism. This paper put forth an approach that attempted to utilize data and model parallel approaches together. It consisted of a data-parallel approach that used a parameter server, which, at any given time, represented the combined learning performed by *Distbelif*. This server stored a sharded version of the most recent model. *Distbelif* also consisted of multiple workers that themselves were sharded over multiple nodes, which provided for model parallelism. It consisted of two techniques to perform learning, Downpour and Sandblaster L-BFGS. Sandblaster consisted of a co-ordinator process that helped with making the learning process more communication efficient for bandwidth-limited applications. Communication constraints pose a challenge for federated learning during the training phase and [43] tries to address some of the issues with sending large amounts of data through the network using compression techniques. This paper reported two key findings. Firstly, asynchronous SGD works well for training deep neural networks, especially for non-convex problems. Secondly, distributed approaches such as this can be applied to train extensive models and achieve a speed-up in model training times.

Communication-Mitigated Federated Learning (CMFL) [44] tries to reduce the communication overhead associated with synchronous variants of federated learning. It does this by attempting to identify updates that might not help towards global convergence before they are sent to the parameter server. It computes this alignment to the global variable by computing the percentage of parameters that have different signs as compared to the global variable. For each worker, after a local update is done, its variables are compared to the global aggregation of the previous iteration. This approach was shown to have a 13.97-time reduction in data transmitted as compared to vanilla federated learning and still managed to reach the same approximate accuracy. GAIA [45] was an

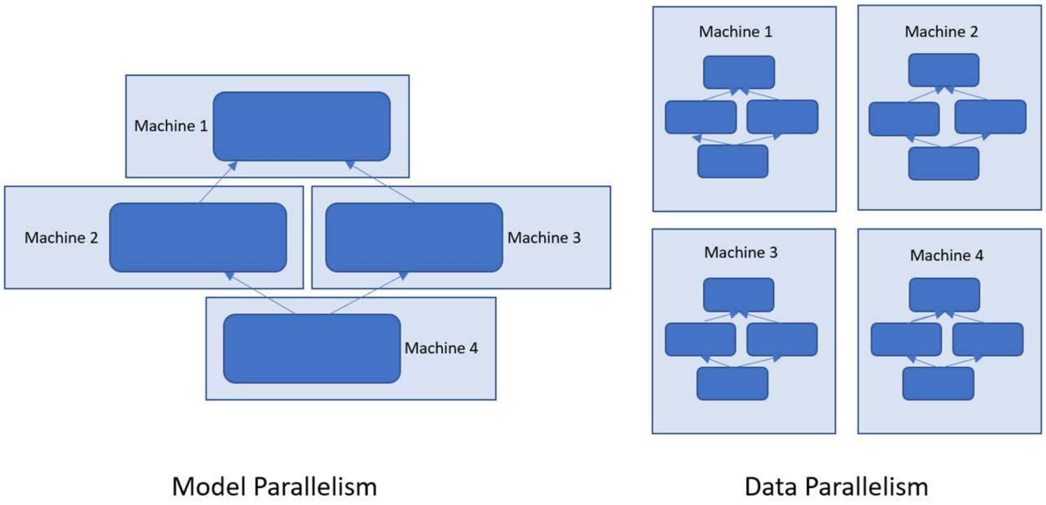

**Figure 2.2:** Model Parallelism vs Data Parallelism

approach similar to CMFL that was applied to geospatial data and used only the magnitude of gradient updates. CMFL was also more efficient than GAIA by a factor of 11. ORCA does not address the communication overhead posed by a large number of updates to the parameter server, and an approach similar to CMFL would be a helpful addition. Federated learning is also known to become biased towards data collected from specific workers and approaches like [46] to mitigate this problem.

GoSGD [47] or Gossip SGD differs from standard Federated Learning. In the absence of a Parameter Server, 'gossip averaging' is leveraged to combine learning from various workers. Here information is exchanged in peer-to-peer fashion between workers, thereby reducing the communication overhead as well as relaxing communication constraints. Each worker is made to draw a random Bernoulli variable, which decides if the worker is to share its learning with another worker that would be chosen uniformly, among others. Once learning is shared, it is stored in the receiving worker. When the receiving worker is to load this learning from its queue, it uses a simple sum-weight gossip protocol to merge its learning with the received learning. Hence, learning for each worker is asynchronous, and no worker is ever blocked. In their experimental analysis, the authors showed this approach to outperform EASGD [11] in terms of training time and accuracy for the CIFAR-10 and dataset. However, the distribution policy used here is unsuitable for Apache Storm, whose topological structure restricts distributed bolts from sharing information. It would take a design overhaul to accommodate this approach within a storm topology, which could provide some compelling results.

Elastic Averaging Stochastic Gradient Descent or EASGD [11] draws attention to the fact that naïve Federated Learning approaches suffer from one fundamental problem: Workers need to communicate with the central parameter server frequently. These frequent communications could lead to increased bottlenecks at the parameter server and reduce the throughput of any potentially scalable architecture. To counter this problem, they introduce a technique called Elastic Averaging SGD. Instead of the naïve aggregation put forth in approaches like Downpour, EASGD computes an elastic force between the worker and the global variable. They both exert an equal and opposite force on each other. The magnitude of this force can be adjusted by tweaking the value of the ’Elasticity’ hyper-parameter. The authors also demonstrate that, as compared to traditional techniques, the communication duration (number of training iterations performed by the worker before combination with the parameter server) can be relaxed, and the algorithm would still converge. Hence, EASGD requires few combinations with the parameter server and still achieves acceptable accuracy. Furthermore, they also present asynchronous and Nesterov momentum powered variants of this algorithm. The asynchronous variant provides higher throughput but also results in more erratic movements of the global variable. They perform experimental analysis in the form of testing done on the CIFAR-10 and Imagenet datasets. Performance is compared to various variants of traditional federated learning approaches. It is shown that EASGD outperforms their counterparts whilst requiring a significantly smaller number of combinations. ORCA uses this approach as its core federated learning approach, as it seems to fit best with our scalable setting.

## 2.2 Machine Learning on DSPEs

The research presented in [48] serves as an introduction to Machine Learning in Apache Storm. It briefly describes Apache Storm and its various components and its computing model. It then describes the key differences between vanilla Storm and *Trident* [49], which supports stateful stream processing with low latency and high throughput. It then briefly describes Trident-ML or the machine learning version of Trident. Trident ML currently supports linear classification, linear regression, k-means clustering, feature normalization, text feature extraction, stream statistics, and pre-trained Twitter sentiment classifier. It then describes Kafka briefly and talks about how Kafka can be integrated and used with Trident-ML. Lastly, this paper describes the distributed K-means Algorithm supported by Trident-ML.

Apache Spark [4] has an in-built machine learning library called MlLib. As of now, Classification, Regression, Decision trees, Random forests, Gradient boosted trees, Alternating least squares (ALS), Clustering algorithms, Latent Dirichlet allocation (LDA), Frequent itemsets, Association rules, and Sequential pattern mining are supported. However, deep learning with Artificial Neural Networks is not supported. Spark also supports streaming workflows via micro-batching. Apache Flink [50] supports Support vector machines, Multiple linear regression, K-Nearest neighbors, Alternating Least Squares (ALS), Distance Metrics, and Cross-Validation. Again, Apache Flink does not support Artificial Neural Networks for Deep Learning. A survey article [51] briefly describes a few independent projects that enable machine learning on apache spark using python libraries like Keras [52], Pytorch [53], Tensorflow [12] and Pyspark [54]. However, these approaches are not officially supported and are not suited for scalable execution.

## 2.3 Parameter Server Implementations

Most of the federated learning approaches described earlier rely on a centralized 'Parameter Server.' Once a node achieves a certain amount of gradient descent, it combines its learning with that of the parameter server. The basic requirements that this server must satisfy is that it must provide concurrently accessible storage for the globally aggregated parameters and must do so using simple push and pull directives whilst following an acceptable consistency model to deal with concurrent requests. The parameter server is also expected to provide its services via an interface like REST, through which packets of data can be transmitted. As the data transmitted is often model parameters, which could be in the order if millions, the parameter server is expected to process such large packets of data seamlessly. *Distbelif* [42] used an application-specific distributed architecture as it evaluated its results under extremely distributed settings. However, other approaches, like EASGD [11], relied on a more naive implementation of a parameter server. Significant effort is applied towards trying to de-centralize this architecture where [55] is an attempt at trying to do the same but the scope of this work assumes a naive parameter server implementation. Naive implementations also suffer for long wait times and [56] proposes an approach to using stale values in a bid to reduce wait times.

The parameter server implementation described in [57] is the most comprehensive approach that was encountered throughout the course of this work. It supports distribution over multiple workers and is also capable of handling multiple concurrent model instances enabled by namespace differentiation. Communication between the server nodes and the client is also done in the form of 'push' and 'pull' requests. These requests also aim to reduce the amount of network traffic transmitted by sending metadata about only those vector values that have changed and then updating only those values. There is also support for multiple consistency models that can be applied based on the application. This framework also provides elastic scalability and fault tolerance, and the framework does not require a restart after new nodes are added. Data is stored in the form of key-value vectors, and vector clocks are associated with each key-value pair to assist with fault tolerance.

## 2.4 Reinforcement Learning

Reinforcement learning [58] is an area of machine learning where agents need to take actions on their environment to maximize the reward or profit gained from choosing the best possible action from a set of actions. The agent eventually learns how to navigate its problem setting by taking into account the positive or negative reinforcement obtained by choosing a specific action on a state. It eventually learns which set actions it must pick to reach its respective goal. Q-learning [59] is a sub-domain of reinforcement learning that seeks to learn specific policies that eventually maximize profit. It learns from actions that might be outside of the scope of the current policy, like taking random actions. Q-learning often involves a Q-table that serves as a reference to pick the best possible action that maximizes the Q-value.

Deep Q-Learning (DQL) [60] introduced the concept of a Q-learning agent that was powered by deep learning. For applications like games, the state-action-reward space is too large. Instead of using a Q-table lookup, DQL proposed using a deep learning network that would predict actions. Traditionally, in Reinforcement Learning, the distribution of data changes as learning progresses, which can pose a problem for deep learning because it assumes a fixed distribution. To counter this problem, random samples from previous transitions were regularly re-trained. This process was also called 'experience replay.' For this paper, an agent was trained to play a total of 7 Atari 2600 games, where the only inputs would be a video frame and the corresponding reward. The resulting architecture outperformed the previous best methods according to literature, namely 'Contingency' and 'Sarsa' in all games and even a human expert in a few games in terms of the total score. It also used an epsilon-greedy approach to train. The experience replay memory helped the network remember state-action transitions that it may have forgotten. The research presented in [61] tries to predict certain hyper-parameters using Deep-Q Learning.

# Chapter 3

# Methodology

ORCA is designed to work as an Apache Storm [7] application on top of an Apache Storm cluster. It is designed specifically to work in a scalable environment, and hence it makes a primary assumption that the underlying Storm cluster is scalable. These scalability requirements would need the cluster to be capable of dynamically increasing the number of tasks in a given topology as and when the underlying JVMs start to run out of resources. ORCA uses Elastic Averaging Stochastic Gradient Descent [11] as its underlying federated learning approach. Each worker is parallelized as an Apache Storm Bolt [8] and a single Spout [8] is responsible for reading data into ORCA. A parameter server is also designed and implemented in Django [10]. This server provides the basic data push, pull, and aggregate operations required for federated learning. The Apache Storm application communicates with the server utilizing REST-API [62] calls. The parameter server also provides the application with a reinforcement learning module built over Keras [52] and Tensorflow [12] that assists with hyper-parameter tuning. The parameter server uses an underlying PostgreSQL [63] database that it uses to store model and reinforcement learning information.

## 3.1 Federated Learning on Apache Storm (RQ1)

To perform federated learning on Apache Storm, we first selected a machine learning library that is compatible with Storm's Java-based architecture. In this thesis, we focus on neural networks such as deep convolutional neural networks, long short-term memory networks and simple artificial neural networks. Many machine learning frameworks like Weka [64] and MOA [65] exist, but these neural network capabilities. We selected DL4J [9], a popular, open-source, and well-supported deep learning library as the underlying deep-learning framework to power ORCA. In addition to Artificial Neural Networks, DL4J also supports Convolutional Neural Networks, Recurrent Neural Networks, and Long Short-term Memory Networks, which made it ideal for implementing, testing and benchmarking our planned Federated Learning approaches. In addition to providing an interface to instantiate, train, and test these networks, DL4J also provides suitable interfaces to format and load data into these networks effortlessly. However, DL4J does not provide any built-in support for the distributed training of these networks.

To perform the federated learning over multiple computing nodes in a DSPE cluster with autoscaling feature, the requirement of this strategy was to be able to react to scalability-driven actions by the underlying framework and do so with the least amount of down-time possible. Model parallel techniques were not considered as part of this work as they would place too much processing overhead on these scalability-driven actions. It was also necessary to avoid data partitioning of the input stream as this would lead to hot-spots. Ensemble Learning [66] is a traditionally accepted approach that runs a similar model on every processing node. In Ensemble Learning, each model performs independent learning, and predictions are made by aggregating each prediction from every model by a secondary selection scheme. A brief description of modern ensemble learning techniques is provided as part if the work done in [67]. Most ensemble learning techniques use homogenous base learners. This would seem like a good fit for our problem setting. However, when new worker nodes are added, the base learner on that node would need to start training from scratch. This could severely hamper the secondary selection scheme. Furthermore, this scheme should also be able to adjust to additional nodes being added. Some ensemble learning techniques like bagging and stacking involve using slightly different models on each worker. Because each model is slightly different, ideally, data has to be partitioned in a specific scheme that ensures each model learns essential characteristics from the data. If data partitioning is done, then selecting these partitioning schemes, on-the-fly as scalability-driven actions occur is a non-trivial problem. Each scalability-driven action would have to be predefined, and this places curtailing restrictions on the seamlessness and fluidity of these actions. In addition to this, if slightly differing models were to be used, selecting what model change would best fit a newly added worker would be a non-trivial problem. Each newly-added worker would also have to start training from scratch, which could cause an imbalance in the accuracy of the secondary aggregation scheme. Hence, owing to these challenges, Ensemble Learning did not seem to be a correct fit for our problem setting.

Data-parallel federated learning approaches seem to be the best fit for our problem setting. Federated learning requires workers to work in a data-parallel setting where they each process similar batches of data. Once a certain threshold in terms of gradient descent is reached, they combine their learning by aggregating their weights on a centralized parameter server. A few federated learning approaches have been discussed in section 2.1 that give a brief overview of the different kinds of approaches that could have been used to solve this problem. EASGD [11] stood out as a strategy that satisfied all our basic requirements, as well as provided a novel solution to both, communication restrictions, and unified convergence. It consists of the same underlying client-server federated learning architecture, as mentioned before, along with a novel aggregation criteria. After certain gradient descent has occurred and aggregation has to be performed, EASGD calculates an elastic force between the client and the server. Instead of naively aggregating the client and server weights as a weighted average, it applies this elastic force in an equal and opposite direction to both the client and server with respect to their original positions in hyper-space. This elastic force helps keep workers, and the central parameter server variable in check. This force's magnitude can be varied. Further, it was also proven that this approach guarantees convergence even if the communication period, restrictions are relaxed. Traditional naive aggregation methods suffered if the number of successive gradient descent steps are increased. Relaxing this constraint would mean reducing the effective bottleneck on the parameter server because of a reduction in the total number of combinations performed.

## 3.2 Elastic Averaging Stochastic Gradient Descent (RQ1)

EASGD comes in two variants based on the averaging performed at the parameter server and worker nodes, i.e., Synchronous EASGD and Asynchronous EASGD. ORCA provides support for both these variants. For synchronous EASGD, workers have to wait until all other workers have finished performing their gradient descents. Once every worker has performed their descents, their weights are sent to the parameter server for aggregation. The parameter server computes the elastic force based on the weights received from the workers taking its own weights into account too. Once the force is computed, it is sent back to each individual worker who applies this force to its own weights. Once elastic forces are sent to every worker, the parameter server applies the cumulative elastic force to its weights and updates itself accordingly. This signals the end of a single training step. Once this step is complete, the parameter server permits the workers to proceed and perform gradient descent for the next step. In our implementation, each worker regularly polls the parameter server and requests for the aforementioned permission. Figure 3.1 demonstrates the sequence of steps performed as part of EASGD.

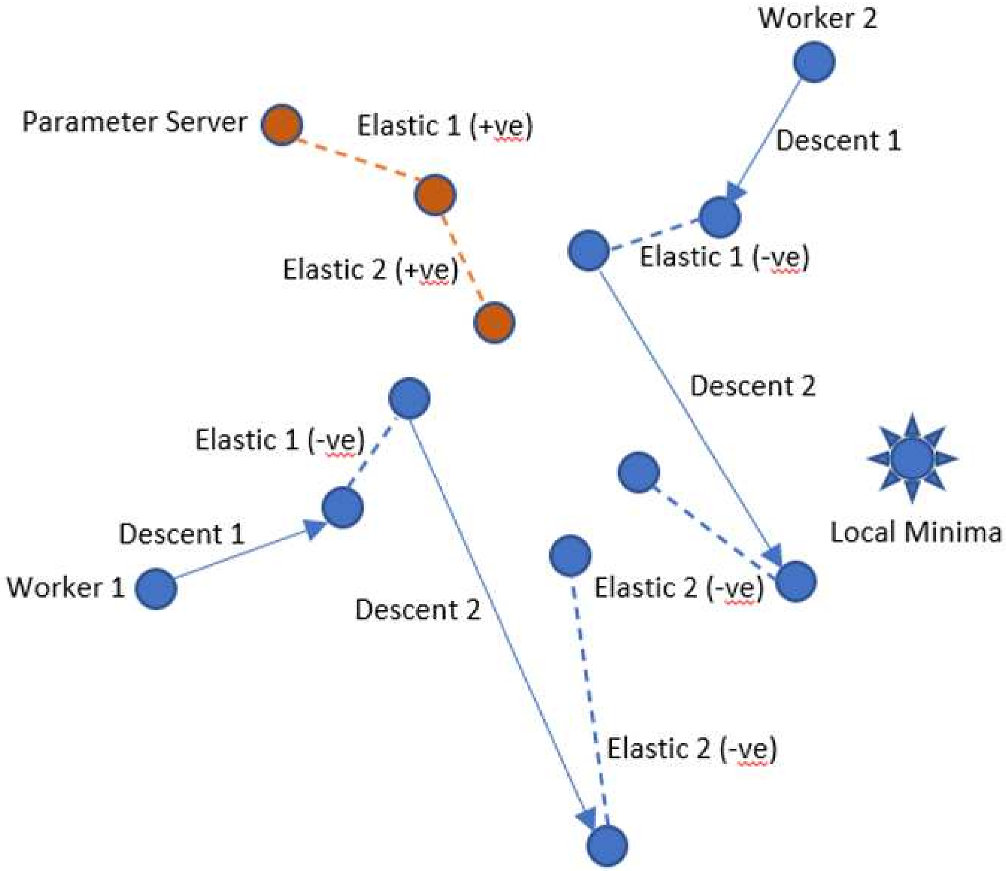

**Figure 3.1:** Steps Performed for EASGD

The asynchronous variant does not enforce that each worker is synchronized at the same step. Here, as a worker completes its individual gradient descent, it sends its weights to the parameter server. The parameter server then computes an elastic force, based entirely on its weights and the requesting worker's weights. It then applies the elastic force to itself and also sends this force back to the worker, which then applies this force to its own weights. The aforementioned process is atomic and non-parallel. Each worker obtains a lock on the parameter server weights when the aggregation process commences and only releases its lock once aggregation is complete. This means that while the weights are locked, all other workers that arrive must wait for the lock in a first-come-first-serve manner. Hence the asynchronous variant is not entirely free of wait times but drastically reduces it. Furthermore, if the communication period is long, the chances of concurrent requests reduce as these workers mostly take varying times to complete their descent. Figure 3.2 demonstrates the sequence of steps performed as part of A-EASGD.

## 3.3 Predicting the Elasticity Hyper-Parameter (RQ2)

The performance of EASGD relies heavily on the configuration of the 'Elasticity' hyper-parameter or $\rho$. This parameter is responsible for deciding the magnitude of the elastic force applied to the parameter server and the workers. Setting $\rho$ to be a higher value results in a greater elastic force being applied. This would restrict workers from exploring areas further away from the parameter server. However this would also lead to a higher force being applied to pull the parameter server closer to the workers. In non-convex problems where multiple similar local minima exist, it might be better to set $\rho$ to a greater value. Setting $\rho$ to a lower value will let workers perform more exploration in hyperspace. This might be better suited to applications where local minima are difficult to find. ORCA attempts to automate the process of finding optimal values for $\rho$. As no prior data would be available in most cases to help find the best value for $\rho$, ORCA turned to unsupervised artificial intelligence techniques to aid with predicting the best $\rho$ and 'Reinforcement learning' [58] was selected to assist with the same. The research presented in [68] attempts to predict the best neural network structure using model parameters such as prediction accuracy. [69] also attempts a similar approach with finding the best hyper-parameter set for convolutional neural networks. We attempt a similar approach to try to tune the elasticity hyper-parameter. Reinforcement

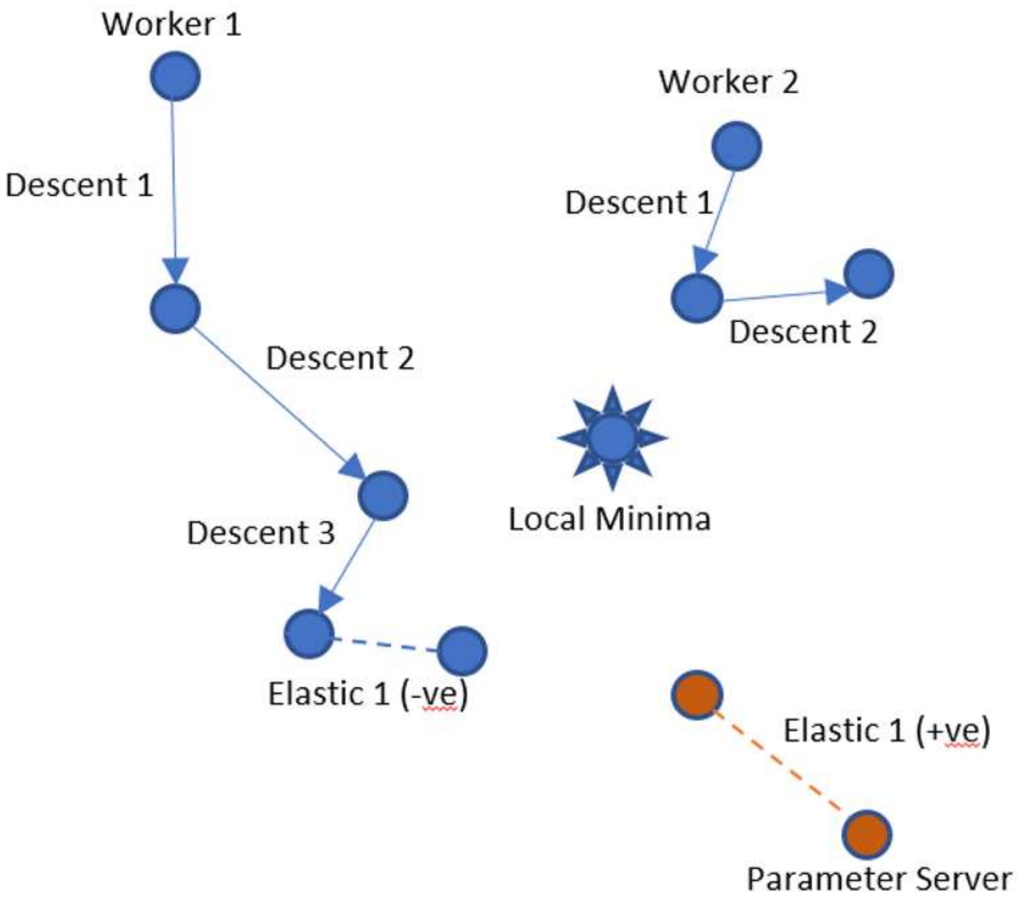

**Figure 3.2:** Steps Performed for A-EASGD

learning does not require labelled data to train and learn. Our problem setting is also incapable of providing labelled training data on initialization. We need the hyper-parameter prediction logic to be able to learn the best $\rho$ from the previous decisions it makes, and this makes reinforcement learning a great fit for our problem setting.

Reinforcement learning is an area of machine learning where the goal is to maximize the cumulative reward obtained by taking a certain action. The problem space is divided into several 'states' which can be traversed by taking specific *'actions.'* A *'reward'* is awarded for every action taken based on how favorable the resulting state is. The goal is to maximize our reward and reach a 'goal' state. Q-Learning hinges on a Q-table that stores the best previous state-action-reward tuples, and it then uses this table for future predictions. However, sometimes the state-action space is so vast that using a Q-table becomes impractical. Deep-Q Learning [60] proposed using a neural network to predict the action with the best Q-value instead of using a table. We use this approach to find the most optimal value for $\rho$. State-space was defined with the help of a few federated learning hyper-parameters as well as network score and accuracy. A set of predefined values for $\rho$ formed the set of possible actions. Reinforcement was provided based on the improvement in model accuracy based on the $\rho$ selected.

# Chapter 4

# System Architecture

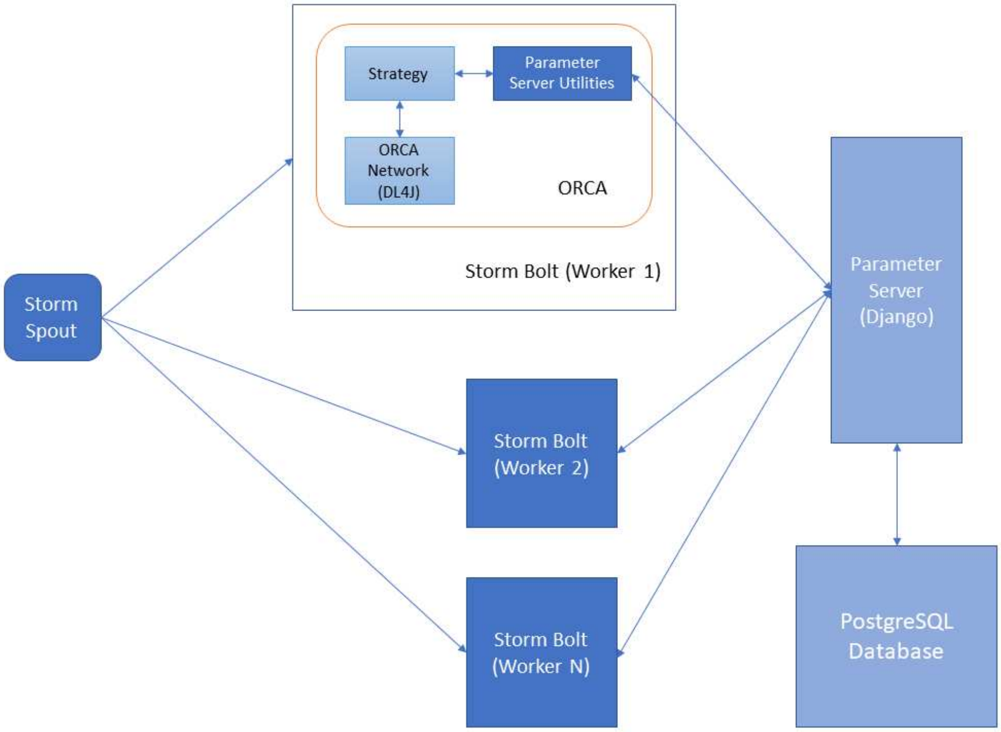

**Figure 4.1:** ORCA architecture.

An overview of the system architecture and working environment of ORCA is presented in Figure 4.1. ORCA as two main working components, the Java-based component that we refer to as ORCA and the Django based Parameter Server. As ORCA is executed atop a storm topology, it runs on multiple storm ’Bolts’ that run in parallel. A single storm ’Spout’ is responsible for sending data into the topology to these ORCA bolts using ’Shuffle Grouping,’ which ensures that each successive tuple emitted by the spout is sent to a new bolt. This ensures that all bolts receive separate data, and no bottlenecks occur. The various components of ORCA and the parameter server and their internal structures are explained in sections 4.1 and 4.2.

## 4.1 Java-based ORCA

Java-based ORCA is responsible for performing the distributed gradient descent for our federated learning approach. It is built to run upon a storm topology. The design structure of Storm allows us to take advantage of sequential code running successively for a given tuple. Storm consists of two kinds of components, *Spouts* and *Bolts*. Spouts serve as data sources, and they are responsible for generating data for the entire storm topology. Bolts are components that feed off the data that spouts emit and can be daisy-chained such that the output from one serves as the input to another. Various data grouping mechanisms then define how data is to be grouped as it travels from the output of one component to the input of another.

ORCA, in its most basic form, consists of a single spout and bolt. However, if users have additional pre-processing steps, additional components can be added. The single bolt used by ORCA is parallelized, which means that multiple instances of the same code are executed on multiple workers nodes. All of the federated learning code is bundled into this single bolt that then runs in parallel. As seen in Figure 4.1, ORCA consists of three main components, a *Strategy*, an ORCA *Neural Network* and a few parameter server utilities. The strategy is responsible for defining the working logic for the underlying Federated Learning strategy that is being used. The Neural Network defines the structure of the underlying artificial neural network that performs minimization of an objective function based on the input data. The parameter server utilities help with communication with the parameter server. ORCA provides support for two strategies and four networks out-of-the-box that were used for the evaluation of this thesis. The end-user can easily develop additional strategies and networks. A basic structural description of these components is presented in subsection 4.1.1. All that is required to run ORCA on Storm is to define a spout that emits data from the data source and to define a bolt that employs a certain strategy and network.

### 4.1.1 Lifecycle

The lifecycle of an ORCA application is explained in Figure 4.2. The bolt first needs to initialize a strategy and network of choice. For every incoming tuple, `handleLocalTuple()`

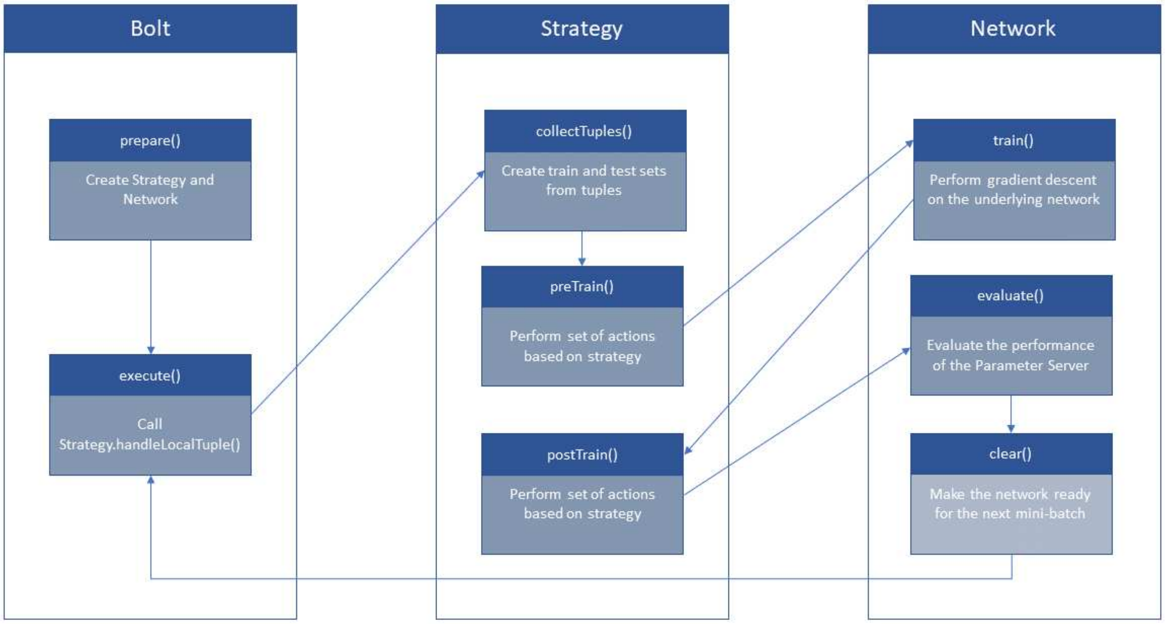
Bolt
Strategy
Network
prepare()
Create Strategy and Network
execute()
Call Strategy.handleLocalTuple()
collectTuples()
Create train and test sets from tuples
preTrain()
Perform set of actions based on strategy
postTrain()
Perform set of actions based on strategy
train()
Perform gradient descent on the underlying network
evaluate()
Evaluate the performance of the Parameter Server
clear()
Make the network ready for the next mini-batch

**Figure 4.2:** Lifecycle of an ORCA application

from the strategy is called. This method either collects a tuple as part of the test or train set for the underlying neural network or performs training actions. This behaviour can be defined in `collectTuples()` of the strategy. Once the train and test sets are filled based on the mini-batch size, training actions are performed. These are broken down into three stages: `preTrain()`, `Network.train()` and `postTrain()`. The actions performed as part of `preTrain()` and `postTrain()` are what form the bases of every Strategy. These methods have to be overridden and defined for every new strategy and examples of how these are defined for EASGD and A-EASGD are demonstrated in section 4.1.2 and 4.1.3. The actions performed as part of `Network.train()` are specific to the underlying DL4J neural network definition. In addition to this method, Network also needs `evaluate()` and `clear()` to be defined where `evaluate()` is responsible for fetching parameters from the parameter server and evaluating the network score and `clear()` clears the train and test set to make the network ready to work on the next mini-batch of data. For training actions, `preTrain()` is called first followed by `Network.train()` and `postTrain()`. These set of methods indicate the end of a single federated learning local training step. If merging with the parameter server occurs, `Network.evaluate()` is called.

Finally, `Network.clear()` is called. This signifies the end of processing for a single tuple and these operations occur for every single input tuple of data.

### 4.1.2 EASGD Implementation

The synchronous variant of EASGD requires a small modification for our specific use case. As we have the possibility of worker nodes being added dynamically, if a node has just been added, we would need to pull the latest weights from the parameter server and initialize the worker's weights based on the newly retrieved values. Workers also need to be synchronized at every merge. In our implementation, as soon as a worker has finished a certain amount of gradient descent and has sent its weights to the parameter server, it increments its training step. The parameter server increments its training step only when it has received weights from all workers and applied the appropriate elastic force to its weights. Each worker then waits for its training step to match that of the parameter server. It polls the parameter server in intervals of 5 seconds till these steps match. This ensures that all workers are synchronized at all times.

The update rules from [11] are followed for ORCA's implementation. let $\rho$ denote the elasticity hyper-parameter, $\eta$ denote the learning rate, $\tau$ denote the communication period, $\pi$ denote the local worker step, $\pi^{ps}$ be the parameter server step and $\phi$ denote the number of workers. Hence as defined in [11], $\alpha = \eta\rho$ and $\beta = \phi\alpha$. Let the total number of weights on a worker be denoted by *i* and hence, the weights at the $t^{th}$ iteration be denoted as $x_t^i$ and the cumulative gradient descent performed on these weights be denoted by $\eta g(x_t^i)$. Let the weights at the parameter server at time $t^{th}$ be denoted by $\tilde{x_t^i}$. Additionally, each worker maintains an iteration counter denoted by *c*. Let the total elastic force computed be denoted by *F*. The update rule at each worker is given by equation 4.1 and is the rule used to develop Algorithm 1. For this algorithm's stability criteria to be met, $0 < \beta < 1$ and $0 < \alpha < (4 - 2\eta)/(4 - \eta)$ have to be met.

$$x_{t+1}^i = x_t^i - \eta g(x_t^i) - \alpha(x_t^i - \tilde{x_t^i}) \tag{4.1}$$

**Algorithm 1** EASGD for ORCA

```
Initialize: Weights on all workers with the same seed, $\pi$=0, $\pi^w$=0
Ensure: $0 < \alpha < (4 - 2\eta)/(4 - \eta)$ and $0 < \beta < 1$
  while exist(train, test batches) do
    if $\pi \neq \pi^{ps}$ then
      Wait till $\pi = \pi^{ps}$
    end if
    if newly initialized then
      if isSet($\tilde{x}_t^i$) then
        $x_t^i \leftarrow \tilde{x}_t^i$
      else
        $\tilde{x}_t^i \leftarrow x_t^i$
      end if
    end if
    if $c = 0$ then
      $F = \alpha\ (x_t^i - \tilde{x}_t^i)$
    end if
    $x_t^i \leftarrow x_t^i - \eta g(x_t^i)$
    if $c = \tau$ then
      $x_t^i \leftarrow x_t^i - F$
      $c = 0$
      $\pi$++
    end if
  end while
```

Algorithm 1 defines the behavior of each worker node that takes part in the EASGD variant Java-based ORCA. It is important to note that when $F$ is computed, additional steps are performed at the parameter server, and these steps are explained in sub-section 4.2.1. While each worker is at its first local step, it shares its weights, as well as receives the weights from the parameter server. The value of $\beta$ is used to update the weights at the parameter server.

### 4.1.3 A-EASGD Implementation

The asynchronous variant of EASGD also has the additional step of having to pull the latest weights from the parameter server and initialize the worker's weights based on the newly retrieved values when a new worker is initialized. Synchronization requirements do not restrict this variant. Here every worker node merges with the parameter server in a one-on-one fashion. Each worker maintains its local gradient descent count and merges with the parameter server on a first-come-first-serve basis. This variant uses equation 4.1 with a small modification to perform updates. In EASGD, the elastic force is computed using local worker weights before its gradient steps, whereas A-EASGD uses the latest weights after gradient descent. The parameter server merging, however, differs vastly from EASGD and will be explained in sub-section 4.2.2. Algorithm 2 describes how ORCA implements A-EASGD. A-EASGD does not maintain a $\pi$ and uses only $c$, which is not reset after every merge. $c\%\tau$ is used to determine when merging with the parameter server occurs. For this algorithm's stability criteria to be met, $0 < \alpha < (4 - 2\eta)/(4 - \eta)$ has to hold.

## 4.2 Django-based Parameter Server

ORCA implements the parameter server as a Django app. Java-based ORCA communicates with this parameter server using REST-API calls. The Django app uses a PostgreSQL database as its underlying storage engine. Django was chosen as a framework because of two significant reasons. Firstly, performing machine learning tasks are easy using python-powered libraries, and developing a reinforcement learning module would pose lesser difficulty on a Python-compatible server. Secondly, concurrency can be handled on the database level using Django's ORM, a feature

**Algorithm 2** A-EASGD for ORCA

```
Initialize: Weights on all workers with the same seed
Ensure: $0 < \alpha < (4 - 2\eta)/(4 - \eta)$
  while exist(train, test batches) do
    if $c = 0$ then
      if isSet($\tilde{x}_t^i$) then
        $x_t^i \leftarrow \tilde{x}_t^i$
      else
        $\tilde{x}_t^i \leftarrow x_t^i$
      end if
    end if
    if $c \neq 0$ & $c\%\tau = 0$ then
      $x_t^i \leftarrow x_t^i - \alpha (x_t^i - \tilde{x}_t^i)$
    end if
    $x_t^i \leftarrow x_t^i - \eta g(x_t^i)$
    $c$ ++
  end while
```

that no other Python powered application server provides. The parameter server stores model weights in the form of a concatenated string inside the PostgreSQL database. Any modifying request has to first acquire a lock on this value before modifying it. The parameter server also provides REST-APIs to save model accuracy, get $\pi^{ps}$, get current weights, set weights, and perform reinforcement learning actions.

### 4.2.1 EASGD Update on Parameter Server

One of the most important APIs that the parameter server provides is the API to perform aggregation of weights on the parameter server for the EASGD variant. This action occurs when the worker node requests the parameter server weights while computing *F*. This single API has dual responsibilities. Firstly, it must consider the weights that the worker has sent for its eventual aggregation, and secondly, it returns its current weights so that the worker can compute its *F*. Each worker also sends the total number of nodes contributing to the aggregation in each request. Let us denote this value by *N*. The parameter server stores each worker's weights till *N* requests arrive, at which point it performs aggregation given by equation 4.2, which uses all constants and terminologies defined in subsection 4.1.2. Algorithm 3 further describes how aggregation for EASGD occurs.

$$\tilde{x_{t+1}} = (1-\beta)\tilde{x}_t + \beta(\frac{1}{\phi}\sum_{i=1}^{\phi} x_t^i) \tag{4.2}$$

**Algorithm 3** Parameter Server weight aggregation for EASGD

```
Initialize: weight-count = 0
  store($x_t^i$)
  weight-count ← weight-count + 1
  temp-weights = $\tilde{x}_t$
  if weight-count = $N$ then
    $\tilde{x} = (1-\beta)\tilde{x} + \beta(\frac{1}{\phi}\sum_{i=1}^{\phi} x^i)$
    weight-count = 0
    $\pi^{ps} \leftarrow \pi^{ps} + 1$
  end if
  return temp-weights
```

### 4.2.2 A-EASGD Update on Parameter Server

The A-EASGD variant has a similar parameter server aggregation method to that of the EASGD variant. The only major difference being that it only considers a single worker in the aggregation operation. It also computes an effective elastic force given by $F$ = $\alpha$ ($x_t^i$ - $\tilde{x_t^i}$). It then applies this force its weights as well as returns this force to the requesting worker too. The worker then goes ahead and applies this force to itself in the opposite direction. Equation 4.3 demonstrates the parameter server update rule for A-EASGD. Algorithm 4 describes how aggregation for A-EASGD occurs.

$$\tilde{x_{t+1}} = \tilde{x}_t + \alpha(x_t^i - \tilde{x_t^i}) \tag{4.3}$$

**Algorithm 4** Parameter Server weight aggregation for A-EASGD

```
$F$ = $\alpha$ ($x_t^i$ - $\tilde{x}$)
$\tilde{x} = \tilde{x} + F$
**return** $F$
```

### 4.2.3 Reinforcement Learning

The choice of the elasticity hyper-parameter($\rho$) and $\alpha$ is crucial to the working of EASGD. The original paper defines a range for $\alpha$ inside, in which the algorithm remains stable. The values of $\rho$ and $\eta$ define how we intend to perform exploration vs. exploitation. If a problem has many local minima that are close to optimal, hinging towards exploitation might be beneficial for the convergence of the center variable. However, if there occur exceptionally few local minima which prove to be sub-optimal, then performing more exploration might be beneficial. We hence propose a Reinforcement Learning [58] powered Deep-Q Learning [60] based approach that can assist with the selection of the most optimal elasticity hyper-parameter. We limit the scope of this work to only the A-EASGD algorithm.

The first step is to define our problem space into state-action pairs. For our problem setting, we define a state at any given time as a combination of test accuracy and training loss of both the local worker and the parameter server. We also define a set of probable values for $\alpha$ as possible actions. We define our reward based on the difference in test accuracy achieved by aggregating weights based on the chosen value of $\alpha$. We also employ an epsilon-greedy approach to pick actions randomly. This approach helps avoid biased picking of values during the initial training of the model. We also employed a 10,000 action-based experience replay mechanism so that the network does not forget initially learned characteristics. This forces the network to periodically re-train itself from a random batch of 5,000 from the 10,000 previously-stored state-action-reward pairs.

The reinforcement learning module executes as *'Act,'* and *'Update'* calls on the parameter server. Act returns a prediction for $\alpha$ based on an epsilon-greedy approach, and Update stores every state-action-reward tuple. 'Update' also periodically re-trains the network using values from the replay memory. This module can be enabled in the A-EASGD strategy by setting a boolean value while instantiating the strategy.

We provide the set of actions (possible values for alpha) as [0.0001, 0.001, 0.01, 0.03, 0.99]. A state in the Deep-Q network is represented by the following:

- Current aggregation step on the parameter server.
- Test accuracy at the local worker.
- Training loss at the local worker.
- Test accuracy at the parameter server.
- Training loss at the parameter server.

These also serve as features for the neural network. The structure of the Deep-Q network is {Dense(5,24), Dense(24,24), Dense(24,5)}. ReLU activation is used on the inner dense layers, and linear activation is used on the output. Mean Squared Error is used as the loss function, and the network is configured with a learning rate of 0.001. The Epsilon decay rate is set to 0.99, and the initial and lowest values of epsilon are 1.0 and 0.1, respectively. The discount rate for the reinforcement is set to 0.95. The reinforcement value is calculated by using the accuracy metrics before and after an action is taken and is calculated by (AccuracyAfter - AccuracyBefore) / AggregationStep. If the result is greater than 0.01, reinforcement of 10 is provided. If it is less than 0.01 and greater than 0, a default reinforcement of 1 is provided. If this value is negative, 0 reinforcement is provided. This policy was defined after looking at how A-EASGD for MNIST mostly performed and converged. This would have to be changed as per the convergence problem.

Figure 4.3 depicts the flow of actions that occur when the reinforcement learning module is activated. Flow starts at the ORCA bolt where the current node and parameter server accuracy and loss values are saved. Control then goes to 'Act,' which then predicts a value for $\rho$ based on these saved values. This value of $\rho$ is then sent to the ORCA bolt, where it is used to compute elastic force and aggregate. Current node and parameter server loss and accuracy values are then

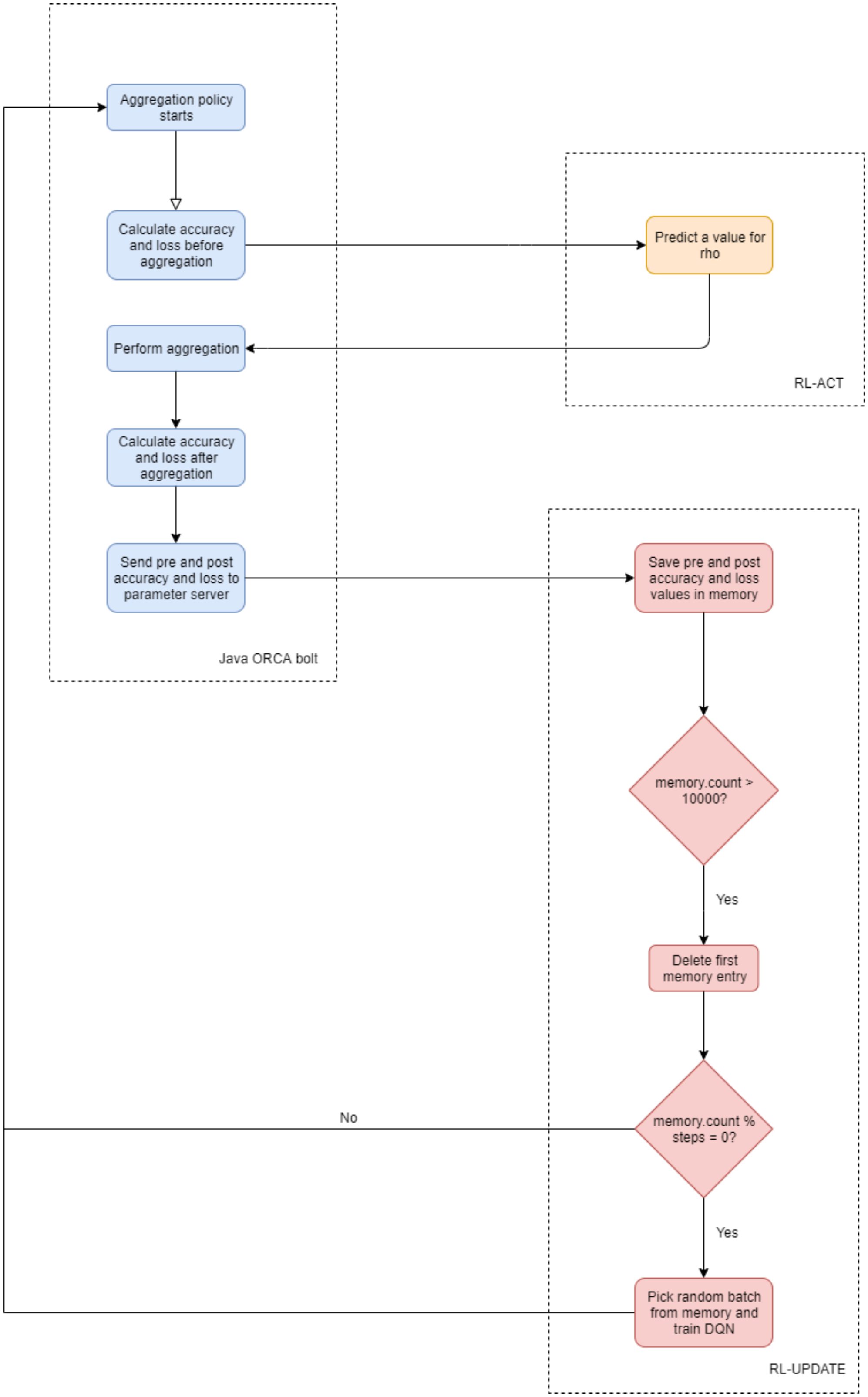
Aggregation policy starts
Calculate accuracy and loss before aggregation
Perform aggregation
Calculate accuracy and loss after aggregation
Send pre and post accuracy and loss to parameter server
Java ORCA bolt
Predict a value for rho
RL-ACT
Save pre and post accuracy and loss values in memory
memory.count > 10000?
Yes
Delete first memory entry
memory.count % steps = 0?
No
Yes
Pick random batch from memory and train DQN
RL-UPDATE

**Figure 4.3:** Flow of actions for elasticity hyper-parameter prediction.

saved, and these pre and post accuracy and loss values are sent to the ’Update’ call. This call saves the accuracy values in memory and then also cleans up memory if necessary. It then ascertains if training has to be done. If yes, it then randomly picks 5000 entries from memory and trains the Deep-Q network using these entries.

# Chapter 5

# Empirical Evaluation

Several experiments were performed to evaluate the performance of ORCA. These experiments were performed on a cluster of 20 nodes where each worker node was powered by an 8-core Intel(R) Xeon(R) E5-2560v2 CPU clocked at 2.10GHz with 32GB of RAM. Most of our experiments were performed on Apache Storm 1.1.1 except for the scalability tests that were run on a specially built scalable version Apache Storm 2.0. The parameter server was set up on a node with the same configuration. Storm topologies used configurations consisting of a single spout and multiple (5-15) bolts with each machine having 8256 MB of dedicated worker heap memory. The PostgreSQL server was hosted on a machine powered by a 2-core Intel(R) Xeon(R) E5-5650 CPU clocked at 2.66GHz with 24GB of RAM. Experiments were performed on a wide variety of datasets under different conditions, and this section presents our observations over the same. We collected observations over four different datasets under different conditions. It is also important to note that every run on each configuration initializes its neural network from the same random seed.

We would also like to put forth that the network configurations we used are not the best network configurations available for these datasets as the goal of ORCA is to achieve similar accuracy to that of non-federated-learning approaches whilst achieving maximum throughput. Hence ORCA'S accuracy would be achieved by processing a large number of training samples. For most results, we have compared ORCA'S performance to that of a similarly configured *'Single'* worker. This is a worker that skips all gradient aggregation steps and performs individual descent while training over similar hyper-parameters to that of an ORCA Neural Network. In terms of Storm, this is a single spout, single bolt configuration that also processes in the same 'mini-batch' configuration. Under all training circumstances, the single worker processes a lesser number of tuples in the same training time as ORCA, and the accuracy it shows is over a lesser number of tuples. The network configuration for each dataset is described in that dataset's section with the following naming convention typeinput,output,... Here, each layer is comma-separated and consists of a type, input, and output. Input and output denote the number of input and output connections in that layer and type denotes the kind of layer it is where C = ’Convolutional’, P = ’Pooling’, D = ’Dense’, S = ’Softmax’, LSTM = ’LSTM layer’ and RO = ’RNN output layer’. In convolutional layers, ’input’ denotes the horizontal and vertical dimensions of the input image. Pooling layers have the dimensions of the kernel size as input and output. Dense layers only have the number of outputs, and Softmax layers have the number of output classes.

The MNIST dataset results also include graphs that indicate the mean and standard deviation for successive results. As major focus of the results was placed on MNIST, these values are not included for the other datasets. As our main graphs plot only the results of the ’best run’, these mean and standard deviation graphs depict the spread of the results. Accuracy values were averaged with respect to the wallclock time for successive runs and each 10 second window has the mean and standard deviation of all successive runs that showed a result for that window. There are a few points where the mean and standard deviation seem to intersect, and this is because those windows had only a single observed value. For MNIST and the NOAA MDG dataset, readings were presented choosing the quickest convergence over 10 runs whereas for the CIFAR-10 and EMC datasets, readings were presented choosing the quickest convergence over 3 runs.

## 5.1 MNIST Dataset

The MNIST dataset [70] presents a collection of handwritten digits from 0-9. Each image is 28x28 pixels, and the whole dataset consists of 60,000 training samples and 10,000 test samples. As convolutional neural networks were shown to do well with this problem, we used a simple convolutional network to classify the MNIST dataset on ORCA. The structure of the network that was used is {C(5,20), P(2,2), C(5,50), P(2,2), D(500), S(10)}. Identity activation was used for the convolutional layers, and Negative log-likelihood was used as the loss function at the softmax layer. The learning rate was set to 0.001, and we trained in mini-batches of 300. This means that for each training step, we provided a total dataset of 300 samples with a batch size of 300. We performed ten epochs with one iteration each for each training step. Hence if the communication period of ORCA was set to twenty, twenty of these training steps were performed before weights were aggregated.

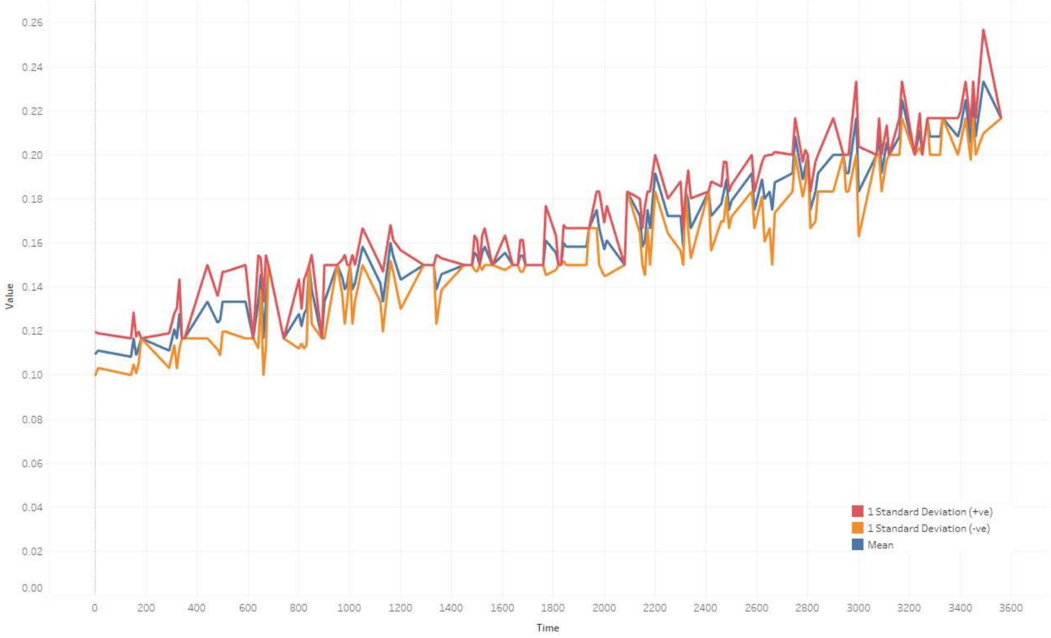

**Figure 5.1:** Mean and standard deviation for $\alpha$ = 0.01 for A-EASGD Strategy.

### 5.1.1 A-EASGD Evaluation

Figure 5.4 shows the test accuracy for MNIST running on various configurations of the elasticity hyper-parameter $\rho$ of ORCA'S A-EASGD implementation. This setting was run on a topology consisting of 15 parallel worker bolts. Let us recall that $\alpha = \eta\rho$, and this graph demonstrates how the performance of the algorithm is dependent on the value of $\alpha$. Also note that our results include an entry for $\alpha = 0.99$. This is an extremely high value and almost means that the worker and parameter server swap places in the hyperspace. Such a high value is not recommended; however, we found it to work best for this particular case. For all our other experiments, such a high value led to exploding gradients, and hence this result was added here as it stood out as an anomaly. For A-EASGD, a high $\alpha$ fits the problem best and converges faster than the 'single' configuration. It is important to note that the rate of input tuples was too fast for all tuples to be consumed, and

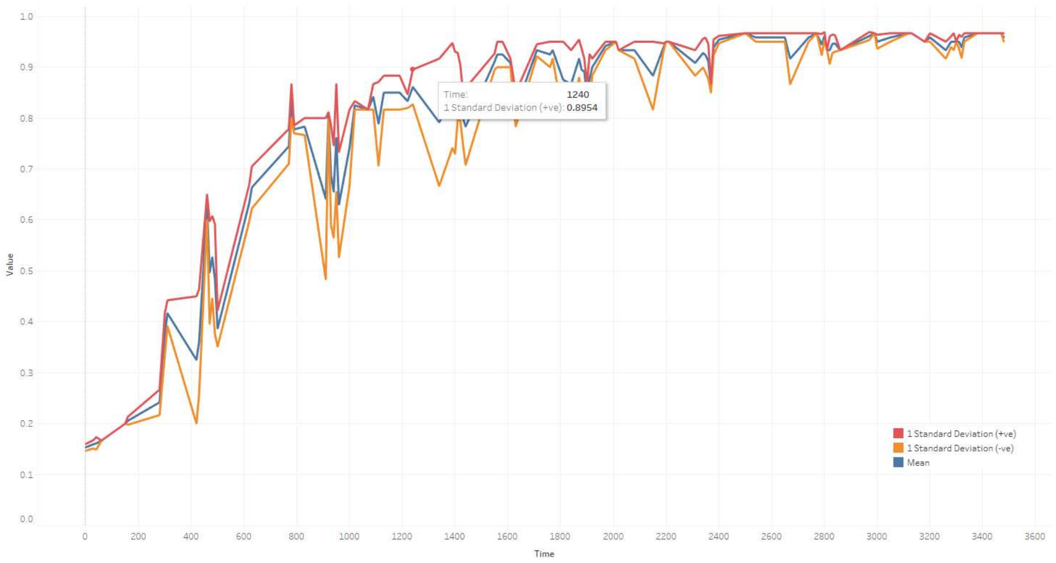

**Figure 5.2:** Mean and standard deviation for $\alpha$ = 0.3 for A-EASGD Strategy.

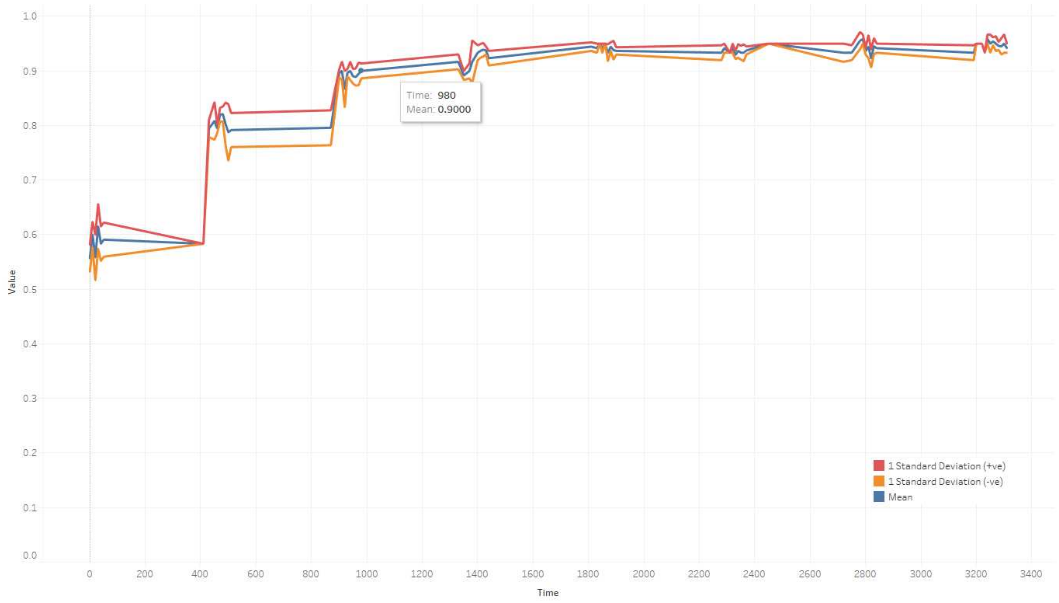

**Figure 5.3:** Mean and standard deviation for $\alpha$ = 0.99 for A-EASGD Strategy.

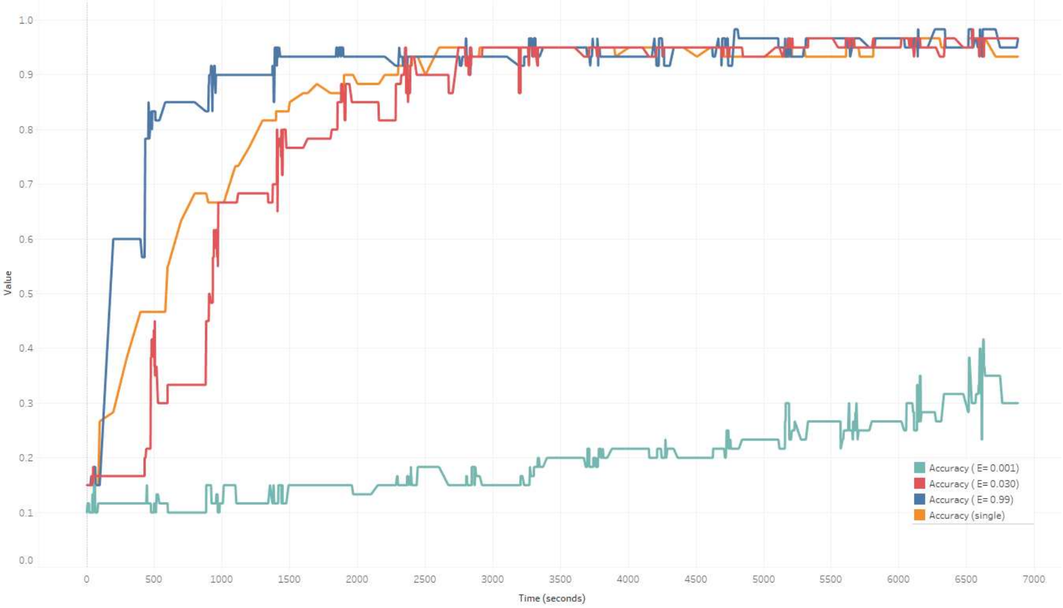

**Figure 5.4:** Test accuracy for A-EASGD strategy.

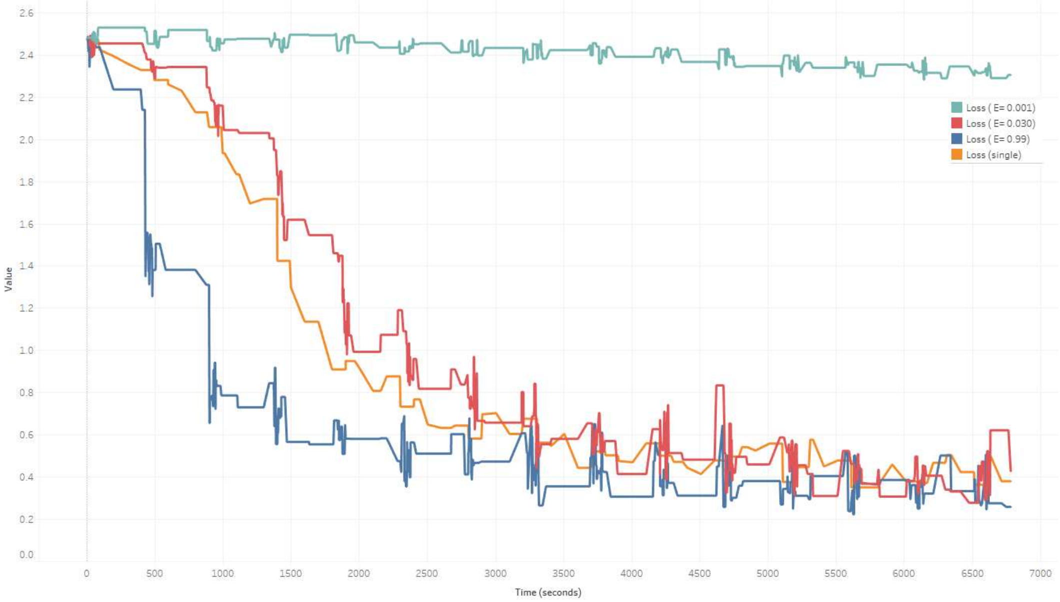

**Figure 5.5:** Training loss for A-EASGD strategy.

hence the 'single' setting is effectively consuming lesser data per second. Given enough time, an $\alpha$ of 0.03 catches up with the single setting in terms of accuracy. An $\alpha$ of 0.99 outperforms its 'single' counterpart by quite a significant margin. A low $\alpha$ of 0.001 converges extremely slowly. As alpha reduces, workers tend to combine lesser towards the center variable and perform more exploration themselves. In a problem like MNIST, there are many and comparable local minima, and hence convergence occurs quickly. A higher elasticity hyper-parameter suits this problem better. Figure 5.5 shows the training loss, which is consistent with the results obtained in Figure 5.4.

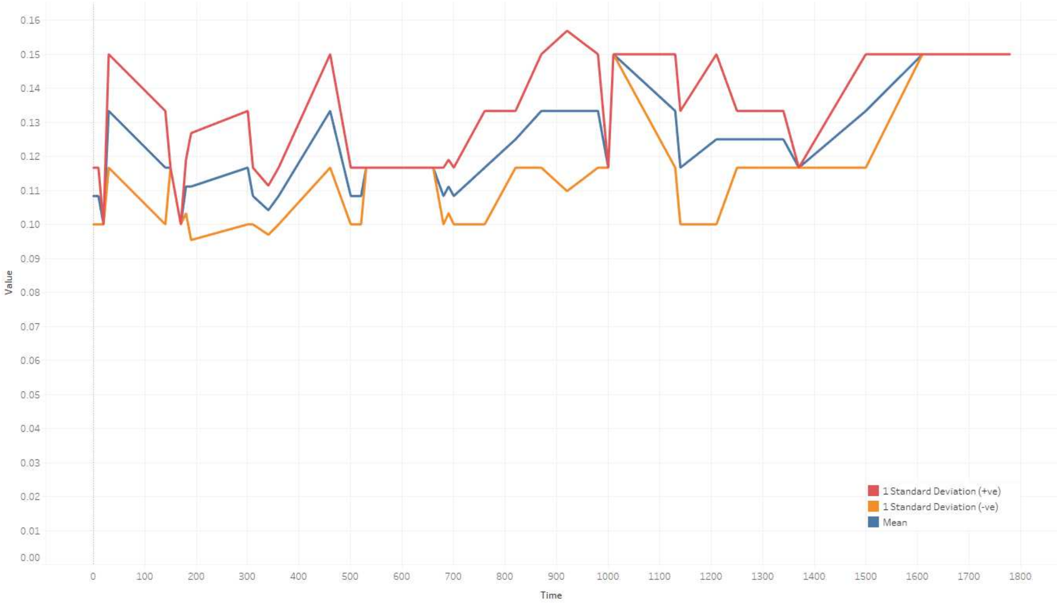

**Figure 5.6:** Mean and standard deviation for $\alpha$ = 0.01 for EASGD Strategy.

### 5.1.2 EASGD Evaluation

Figure 5.9 shows the test accuracy for MNIST running on various configurations of the elasticity hyper-parameter $\rho$ on ORCA'S EASGD implementation. $\beta$ was set to 0.9, and this setting was computed on five worker nodes. Hence $\alpha$ could be set to a maximum value of 0.18. Setting it higher than this leads to exploding gradients. As there is only one update to the parameter server weights (when all the workers finish), the number of entries on this graph is lesser than that of A-EASGD. A high value for $\alpha$ outperforms its corresponding 'single' variant. When $\alpha$ is set to

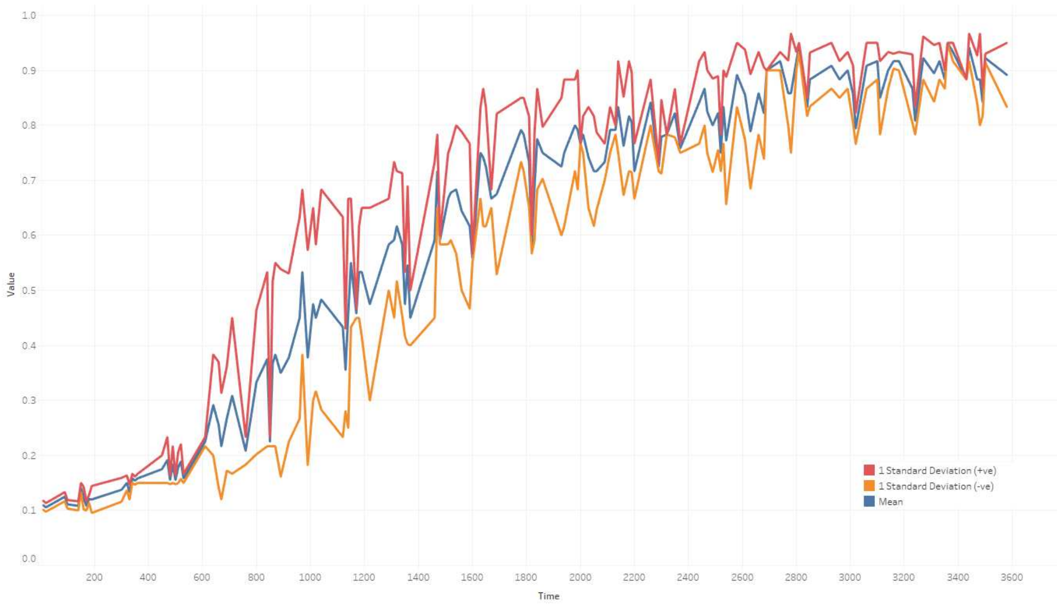

**Figure 5.7:** Mean and standard deviation for $\alpha$ = 0.1 for EASGD Strategy.

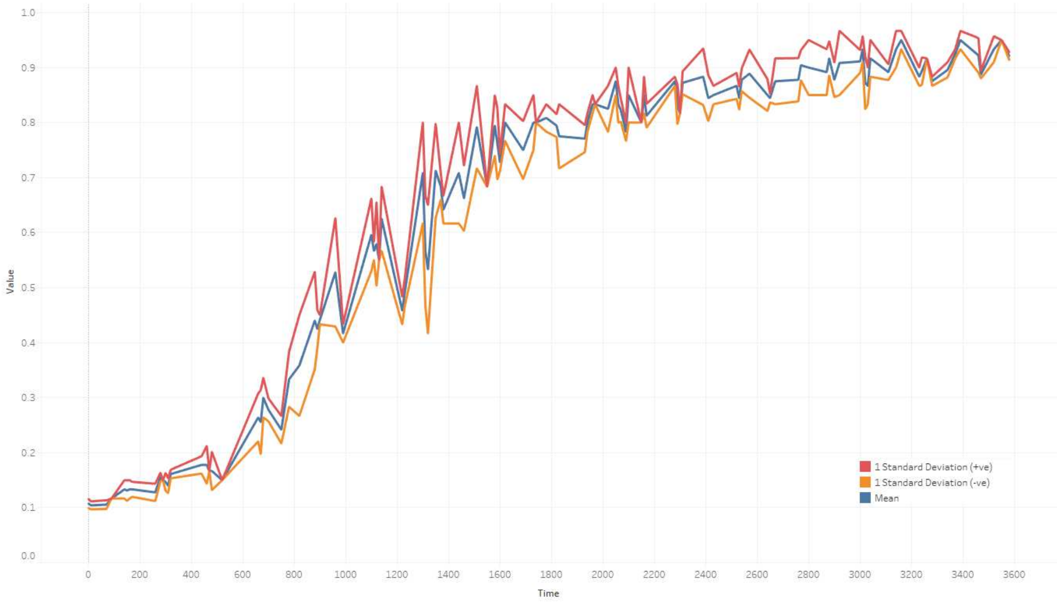

**Figure 5.8:** Mean and standard deviation for $\alpha$ = 0.18 for EASGD Strategy.

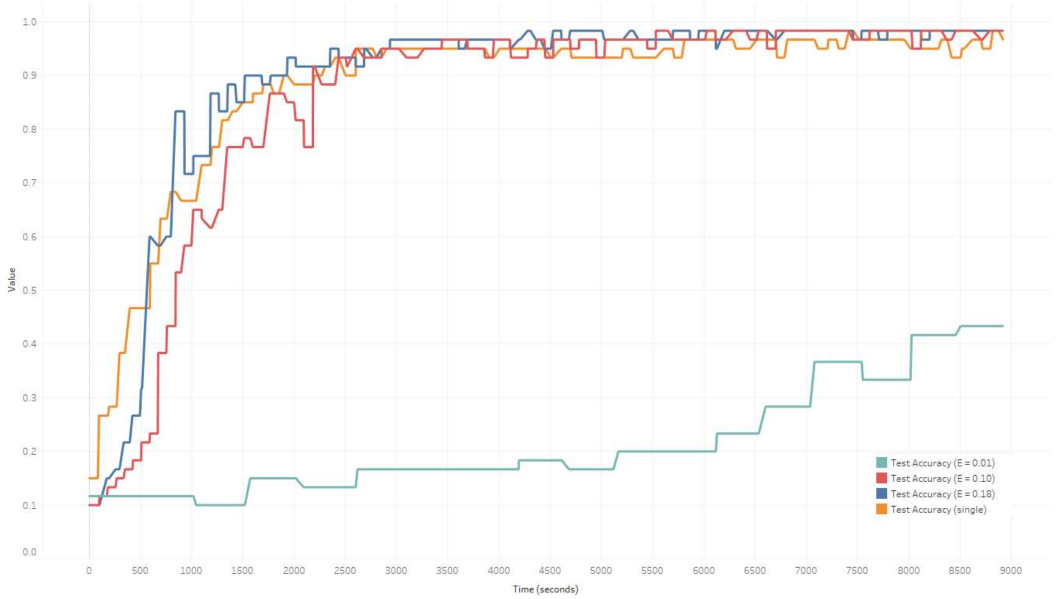

**Figure 5.9:** Test accuracy for EASGD strategy.

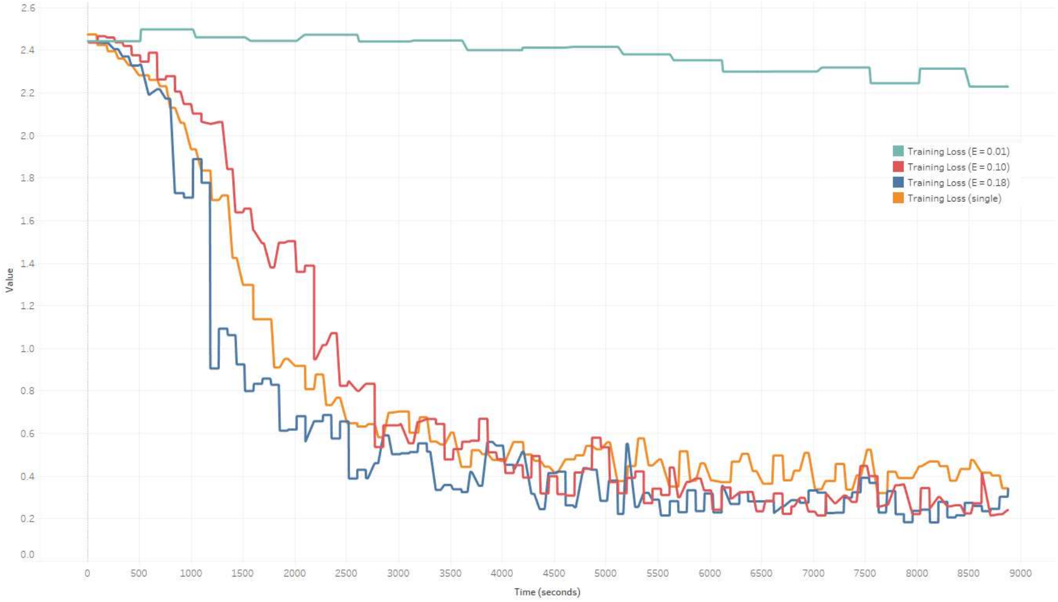

**Figure 5.10:** Training loss for EASGD strategy.

0.10, it performs slightly worse than the ’single’ variant but catches up quickly. When $\alpha$ is set to 0.01, it converges slowly but still catches up after enough time has passed. Figure 5.10 shows the training loss for this setting, which is consistent with the results obtained in Figure 5.9.

### 5.1.3 Effect of $\tau$ on A-EASGD

We also ran experiments to compare and contrast how the communication period ($\tau$) affects the A-EASGD algorithm. One of the most important points taken into account while finalizing EASGD as our algorithm to drive federated learning was the fact that satisfactory accuracy is obtained even with a lesser number of aggregations with the parameter server. It is worth noting that as we decided to go with a naive parameter server, we decided to provide consistency by locking rows on every update. This meant that every other worker update request has to wait for the current update to finish in a first-come-first-serve manner. In our experiments, we observed high wait times for workers, even in the ’A-EASGD’ setting. This led to an overall drastic drop in overall throughput. However, this was mitigated when we increased $\tau$ to 5 from 2 for 10 workers. This further decreased when $\tau$ was increased to 20. In general, we observed that when the number of workers was increased, $\tau$ also had to be increased to maintain throughput. This, however, did not affect the overall accuracy, and this can be observed in Figure 5.12 and Figure 5.13. We can hence conclude that ORCA can ensure that accuracy and throughput are maintained while the number of workers is increased, which was an important requirement for designing a scalable architecture.

### 5.1.4 Scalability Evaluation

An essential part of evaluating the performance of ORCA is ensuring that it can cope with the addition and removal of worker nodes. A scalable DSPE can trigger this addition and removal in response to fluctuations in the arrival rate of input tuples. For the evaluation of this section, we used a scalable version of Apache Storm 2.0, which was developed by the Big Data Research Group at the Department of Computer Science at Colorado State University. Each worker maintains a local queue for incoming tuples that are waiting to be processed, and this scalable version continuously monitors the queue size at every worker node. If the queue size passes a certain threshold, it

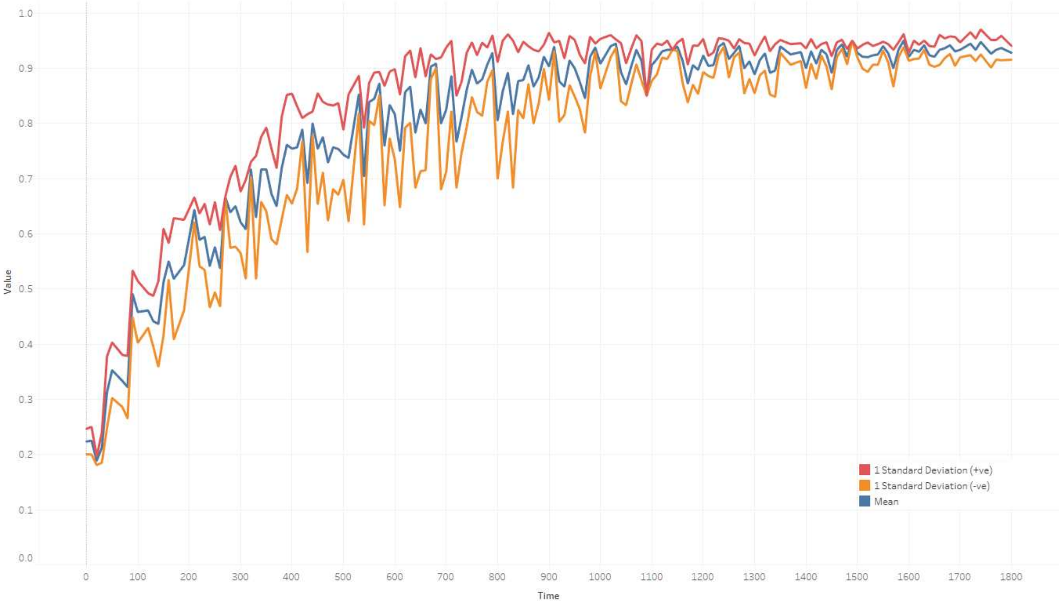

**Figure 5.11:** Mean and standard deviation for $\alpha$ = 0.99 and $\tau$ = 5 for A-EASGD Strategy.

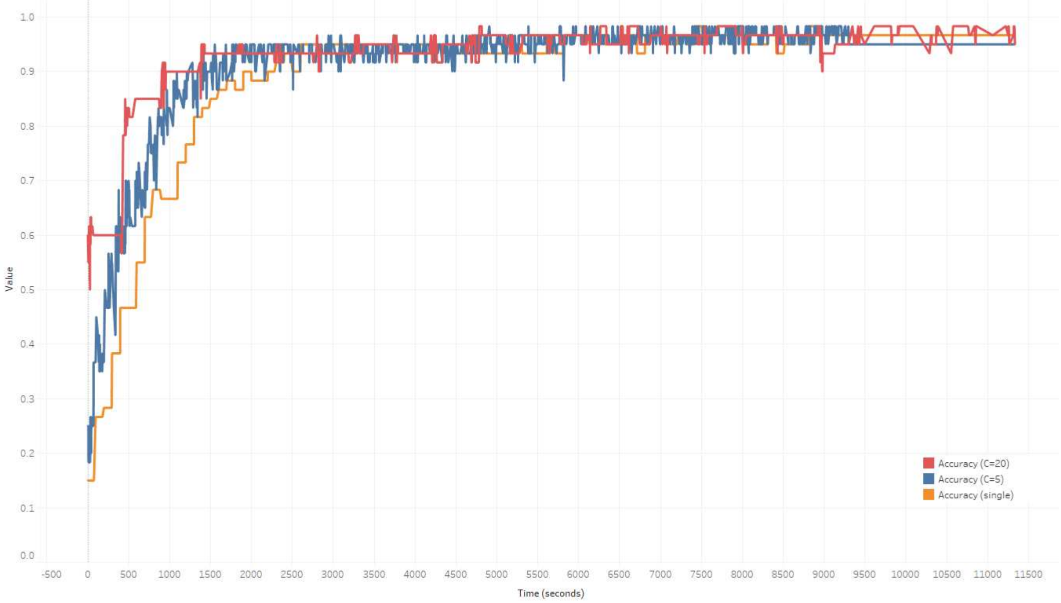

**Figure 5.12:** Test accuracy for different '$\tau$'

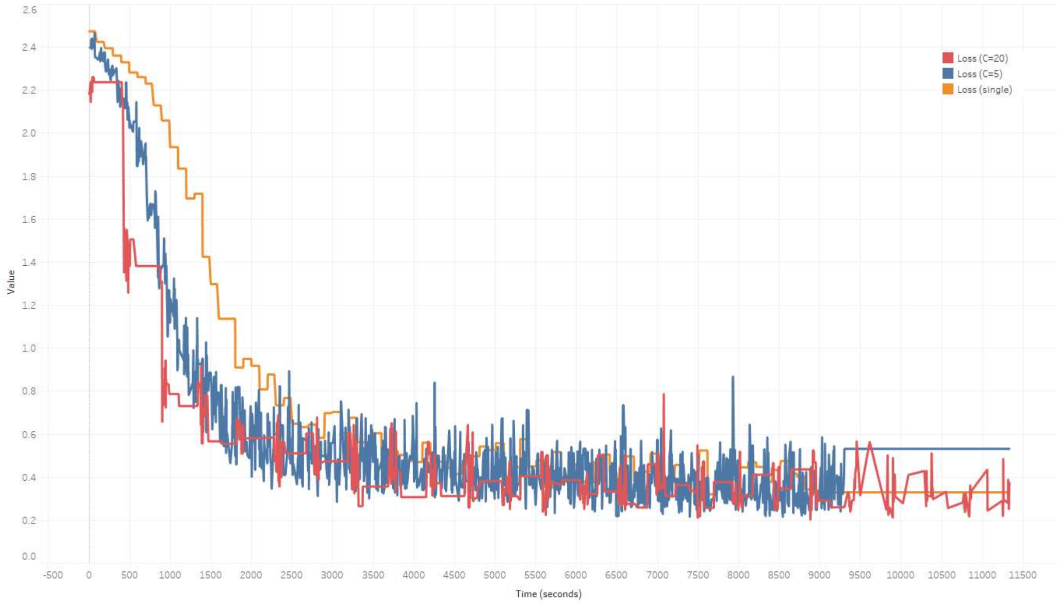

**Figure 5.13:** Training loss for different '$\tau$'

adds a worker node and calls Storm's `rebalance` method. `Rebalance` triggers a restart of the topology on all the nodes currently in the cluster. Here is where the choice of federated learning helps adjust to this particular use case. When the topology restarts, each node first loads the most recent weights and then continues to follow the algorithm defined by the strategy employed. For our experiment, we set the queue size threshold to 20% and also employed a variable arrival rate in the spout by generating sleep times based on a gaussian distribution. The framework also starts with a single node by default and then upscales and downscales as needed.

For our experiments, we employ A-EASGD strategy with $\alpha$ = 0.02 and $\tau$ = 20. Figure 5.14 demonstrates how the scalable system adapts to a change in arrival rates. The number of workers increases or decreases based on fluctuations in arrival rates. As evident in Figure 5.14, the system is also able to provide momentarily high throughputs in response to fluctuating arrival rates that are shown in Figure 5.15. Also, observe that the time axis on Figure 5.14 and Figure 5.15 are not the same, which is due to the underlying framework's inability to measure arrival rates while a `rebalance` is in progress. A non-scalable framework would not be able to increase throughput above 200 tuples per second, as that was the observed maximum value. ORCA manages to achieve peak values of up to 600 tuples per second. Figure 5.16 then demonstrates how accuracy values are

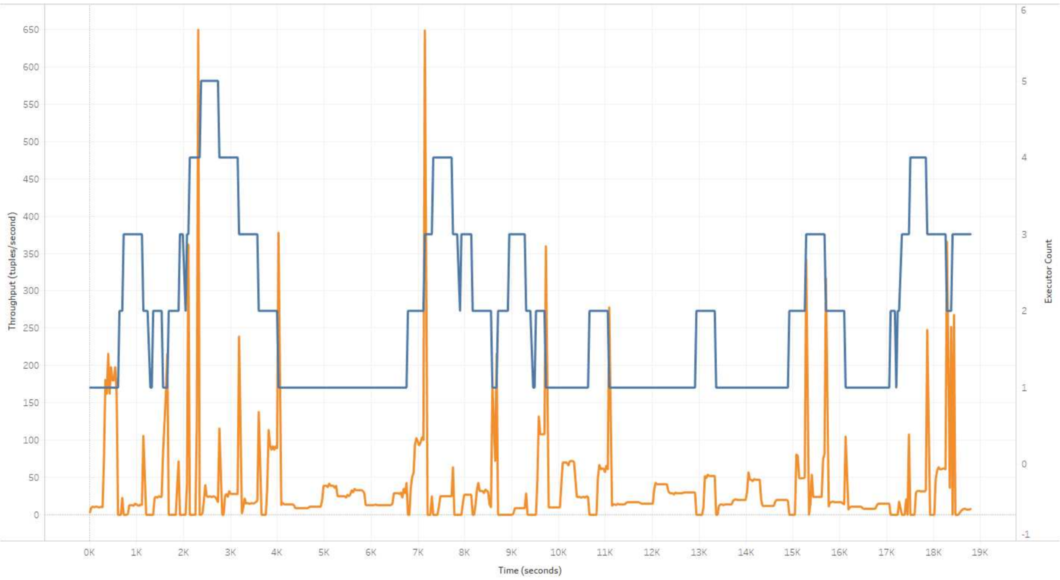

**Figure 5.14:** Throughput and Number of workers against Time

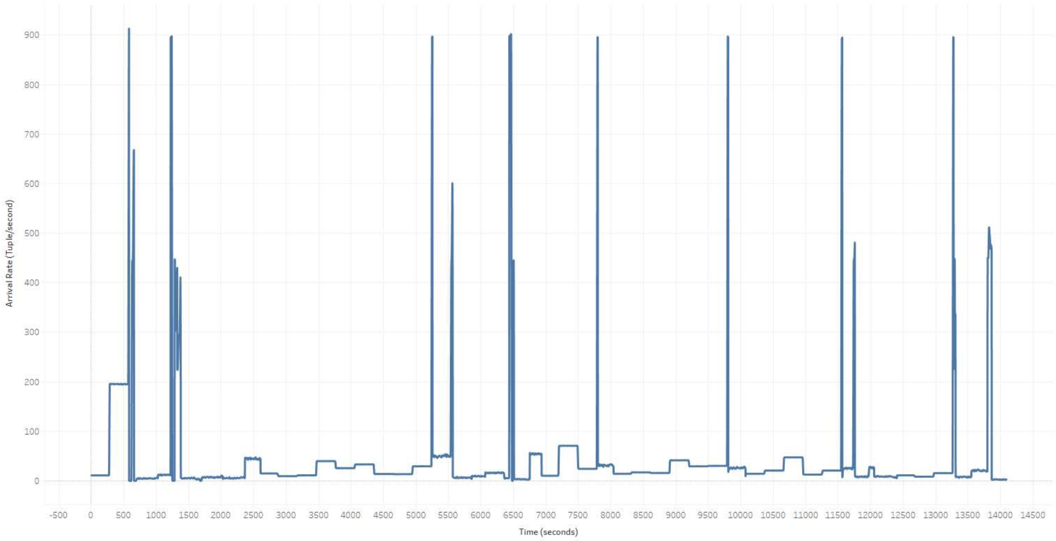

**Figure 5.15:** Arrival rate against Time

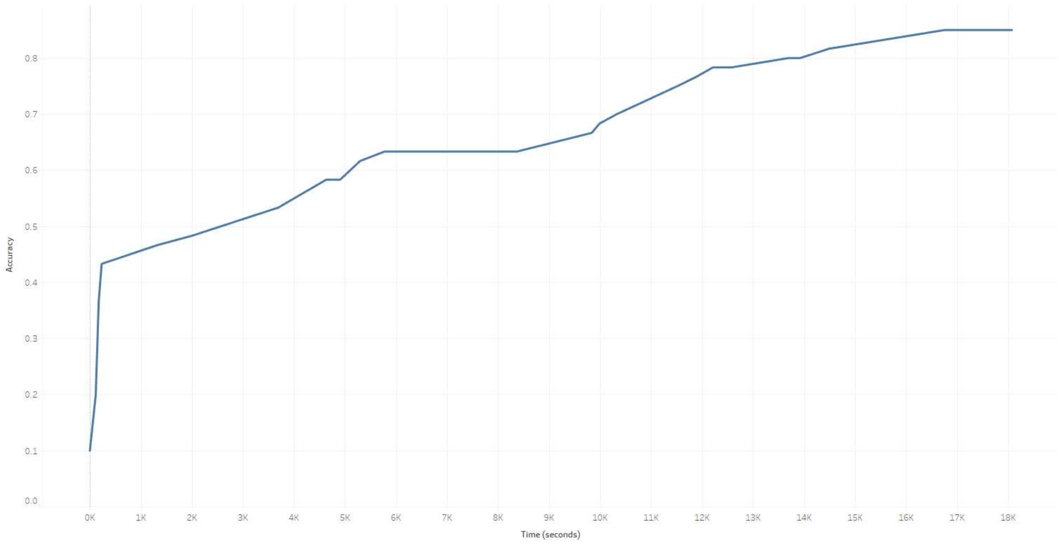

**Figure 5.16:** Test accuracy against Time

unaffected by the scale-in and scale-out operations of the underlying framework. It still manages to achieve a trajectory similar to that of the one demonstrated earlier in sub-section 5.1.1. It does take a significant amount of time to converge as compared to the previous results because the variable arrival rates are centered around a lower arrival rate. Orca manages to handle seamlessly, in, and out-scaling of the underlying architecture.

## 5.2 CIFAR-10 dataset

The CIFAR-10 dataset [71] consists of a set of RGB color images labeled into ten classes. It is widely used as a benchmark dataset for classification problems. It is a fairly difficult problem to solve, and deep convolutional networks do well with classifying these images. We included this dataset to gauge how well Orca performs on a considerably difficult classification problem. We used a network that performs rather poorly on this dataset in terms of accuracy (peak accuracy of any sort being 0.65 or 65%), as we wanted to test the performance of Orca on such a problem. the structure of the network we used was {C(5,32), P(2,2), C(3,64), C(3,64), P(2,2), D(512), S(10)}. Identity activation is used on the convolutional layers, and ReLU is used on the dense layer. Reconstruction Cross-entropy was used as the loss function at the softmax layer. We set the learning rate to 0.01 and trained in mini-batches of 300, and is identical to that of our MNIST experiments.

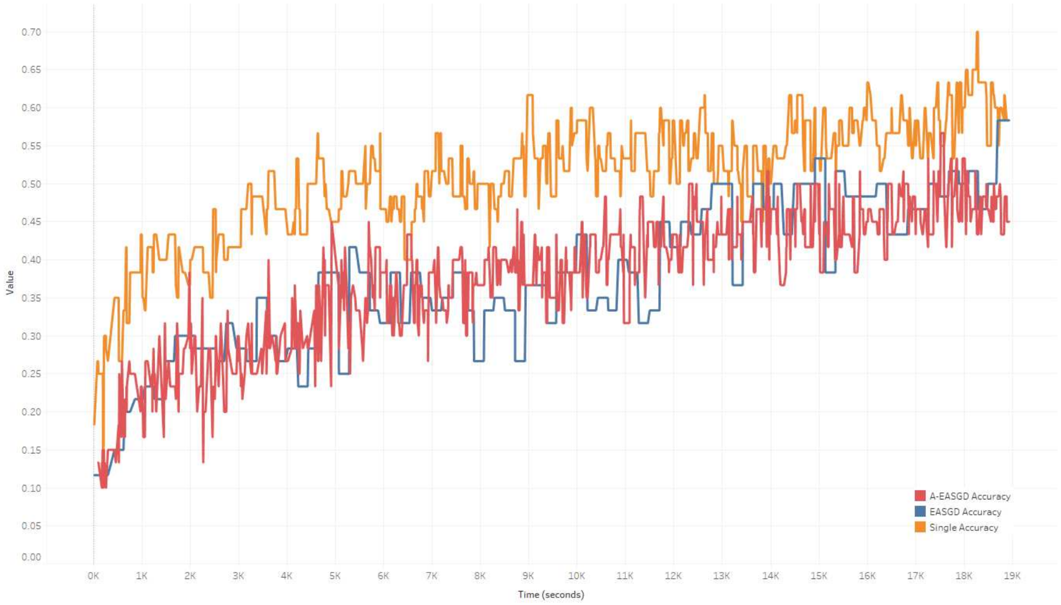

**Figure 5.17:** Test accuracy for CIFAR experiments

To simplify our results, we plotted only the best results we achieved for both EASGD and A-EASGD strategies and compared them to the ’single’ run. Training loss was also skipped as these results were consistent with the test accuracy findings. Figure 5.17 represents our findings from the CIFAR dataset. The single run outperforms both federated learning strategies, although both strategies do show strong signs of picking up. A-EASGD performs best with $\alpha$ = 0.2 and EASGD performs best with $\alpha$ = 0.18. For values higher than this, we observed exploding gradients. We also observed that with periodic fluctuations in training loss (sudden successive peaks and troughs), federated learning algorithms struggle to converge, as is the case with the CIFAR-10 dataset. We also observed that with a higher $\alpha$, these peaks and troughs are magnified in A-EASGD; however, it also does achieve a higher mean accuracy faster. Similar to our findings with MNIST, a smaller communication period led to longer wait/idle times, and a $\tau$ = 20 seemed to work well for both variants. One more interesting observation was that as this is a computationally difficult problem, workers were not in synchronization in their updates. These updates were closely clustered for MNIST but are fairly random for CIFAR-10. EASGD also had longer wait times as most nodes had to wait for straggling nodes.

## 5.3 Dell EMC dataset (API Access Prediction)

The results obtained from the CIFAR-10 dataset got us curious as to how ORCA performed on a challenging convergence problem. Ideally, we needed to use a dataset that was not widely benchmarked and one that provided a challenging convergence problem. As a result of a past internship, Dell EMC [72] provided us with a dataset of anonymized REST-API calls. API calls were logged along with begin and end timestamps as well as five features about the data. API response time was calculated using these timestamps and provided a sixth and final feature. We ended up with a total of 347008 rows in our dataset. After feature engineering steps, this problem was formulated as a classification problem. Feature number four was selected as the target variable and was divided into four classes. The top three most occurring values made the first three classes, and all other values were classified as the fourth class. A simple Artificial Neural Network was designed to try to predict this problem. the structure of the network was {D(5,20), S(20,1)}. TanH activation was used for the dense layer, and Negative log-likelihood was used as the loss function at the softmax layer. The learning rate was set to 0.001, and we trained on mini-batches of 1,000.

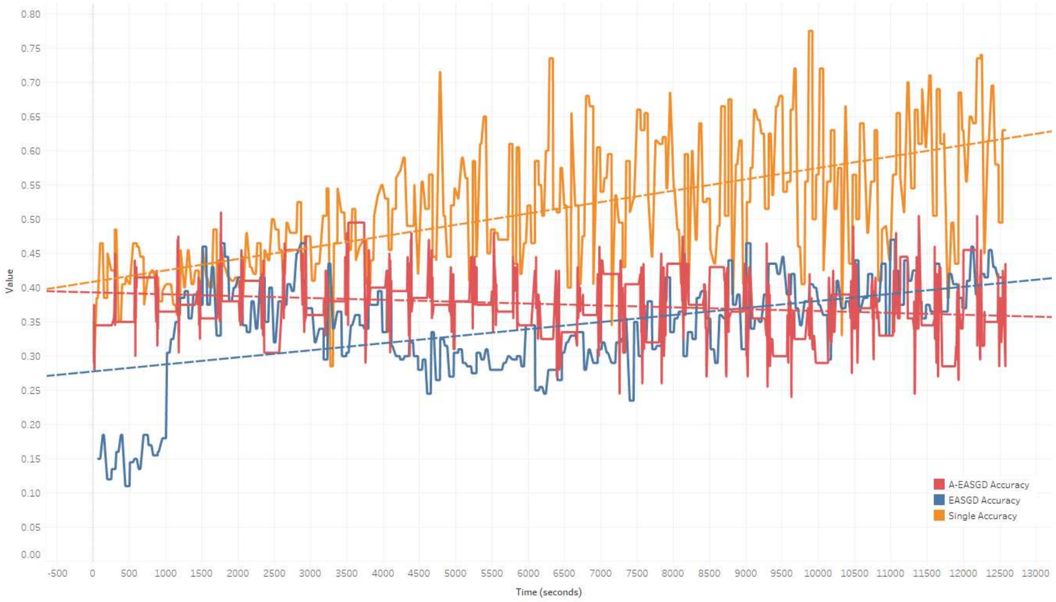

**Figure 5.18:** Test accuracy for Dell EMC experiments

Figure 5.18 demonstrates how ORCA performs on the Dell EMC dataset. An $\alpha$ of 0.1 worked best for A-EASGD and 0.05 worked best for EASGD; however, it is worth noting that almost all values above 0.5 in both settings provided almost equally bad results. As there are extreme fluctuations in training accuracy, trend lines are added to aid visualization. Once again, in a difficult setting, ORCA is outperformed by its corresponding single variant. A-EASGD seemed to repeatedly show a negative trend indicating that it was failing to learn a lot irrespective of the value of $\alpha$ that was selected. EASGD performed much better than its asynchronous sibling. As evident from the trend line, Its performance does show an upward trend in terms of test accuracy. However, it did tend to show a steady rise and then drop, and this behavior was observed repeatedly. We can hence conclude that ORCA does not perform well on problems that are extremely difficult and have a great number of fluctuations in their training loss. A-EASGD actually shows a negative trend and struggles with this problem.

## 5.4 NOAA dataset (LSTM Usage)

Long Short Term Memory networks [73] differ from regular feedforward networks as they also contain feedback connections. They are capable of not only analyzing single data points but also sequences of data. We had to ensure that ORCA was capable of handling sequence-based networks as well. Mountain Data Group [74] provided us with a dataset consisting of data collected from the NOAA dataset [75]. A script was developed by Mountain Data Group that assisted with collecting minimum and maximum temperature values for every county in the United States from the year 1960. We used a subset of this data where we considered only the maximum temperature for every day from 1960 till 2019, reported by the weather station at Larimer County, Colorado. We then attempted to feed this data sequentially to a simple LSTM to find out if it could predict the maximum temperature on the $31^{st}$ day given the maximum temperatures of the 30 previous days. We modeled this as a regression problem that our simple LSTM attempted to solve.

The structure of the network used by us was {C(1,7), LSTM(7,10), RO(10,1)}. We used RELU activation for the convolutional layer, SoftSign activation for the LSTM layer, and Identity acti-

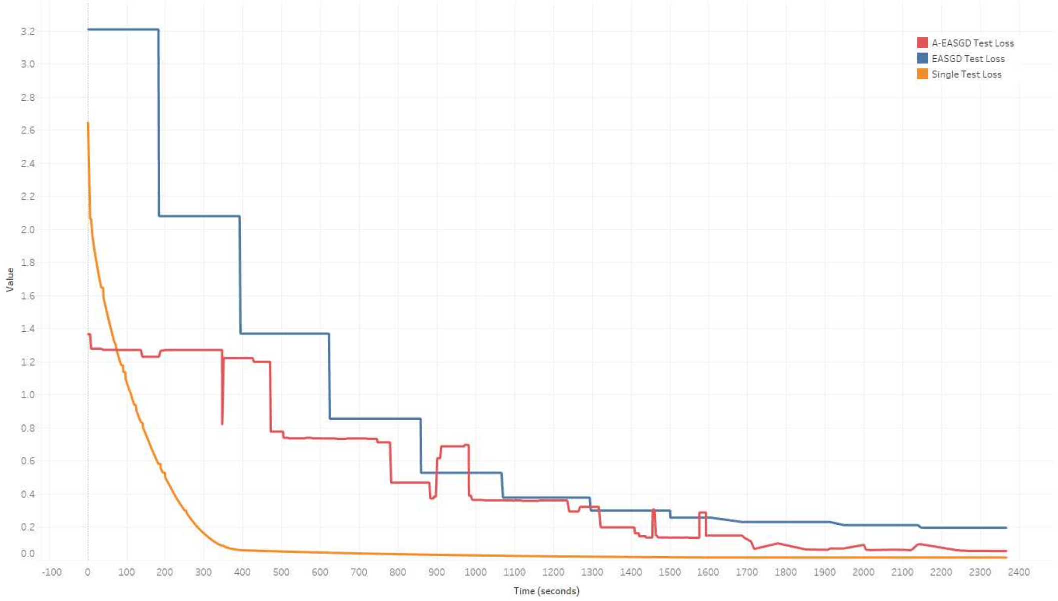

**Figure 5.19:** Test loss for NOAA experiments

vation for the RNN output layer. While creating our dataset, we supplied 30 time-stacked temperatures one-after-the-other as a single record, and served as our input to the net. The predicted variable was the temperature at the $31^{st}$ day. We ended up with 21,218 of these records. We batched 20 of these into a single batch that was then used for training. The 'single' variant was trained using successive batches, which were ordered according to their date and sent into the network. The challenge with federated learning is that these batches need not be ordered, and hence, while evaluating the performance of ORCA, we did not send batches sequentially. Instead, these batches were ordered randomly. We used a learning rate of 0.001 as higher rates lead to too quick convergence.

Figure 5.19 demonstrates how ORCA performs on the NOAA dataset. the sequential variant shows a rapid and smooth convergence. EASGD, which was configured with $\alpha$ = 0.1 (best performance), also had a relatively smooth convergence, and the steps seen in Figure 5.19 are caused because values from the previous aggregation are preserved till the next. A-EASGD, configured with $\alpha$ = 0.3 for best performance, shows a slightly erratic convergence that could be explained by different workers pulling the center variable in multiple different directions. A-EASGD does descend faster and almost catches up with the single variant in our measured time-frame. Both federated learning variants show positive results and almost catch up with the ’single’ variant.

## 5.5 Reinforcement Learning

For our evaluations for reinforcement learning, we consider only the A-EASGD strategy employed on the MNIST dataset. We also employed a smaller communication period of 5 so that we could get more data points for our reinforcement learning algorithm. Because of the lower communication period, we had to also reduce the number of workers to 7. Recall that an unusually high value of $\alpha = 0.99$ performed best on this dataset. Hence the goal of our reinforcement learning approach would be that given multiple values of $\alpha$, it would need to settle on 0.99 eventually.

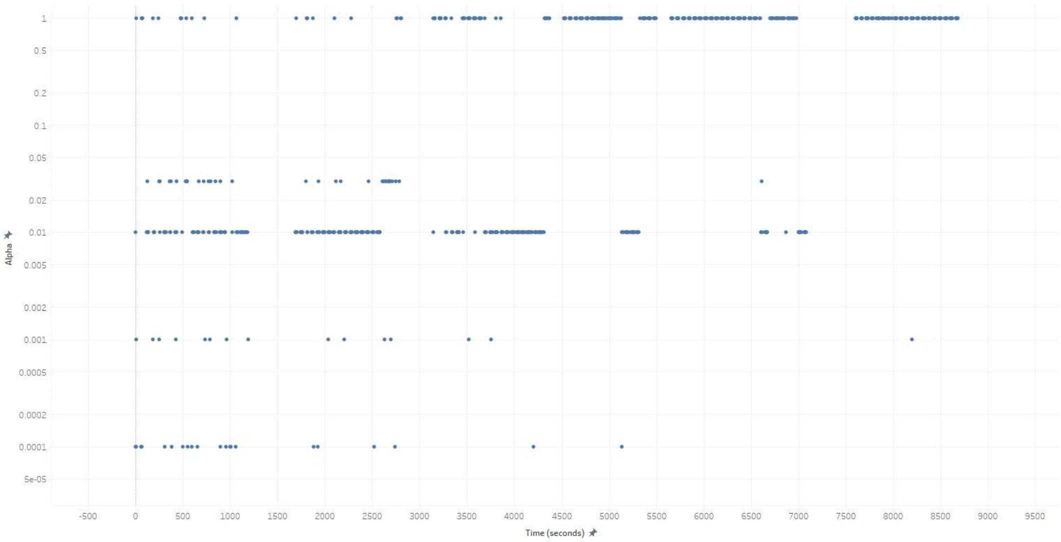

**Figure 5.20:** Prediction for $\alpha$ against Time

It is important to note that MNIST under A-EASGD tends to converge faster than epsilon decay occurs. Hence we preserved Reinforcement learning values through 10 successive runs of the federated learning algorithm. Each run was terminated as soon as the center variable reached an accuracy of 0.9. Figure 5.20 demonstrates how the Reinforcement learning algorithm predicts $\alpha$. It starts with mostly random picks governed by the epsilon greedy approach, but as soon as the value of epsilon reaches 0.1 at approximately the 5000-second mark, It starts to pick 0.99 repeatedly. It

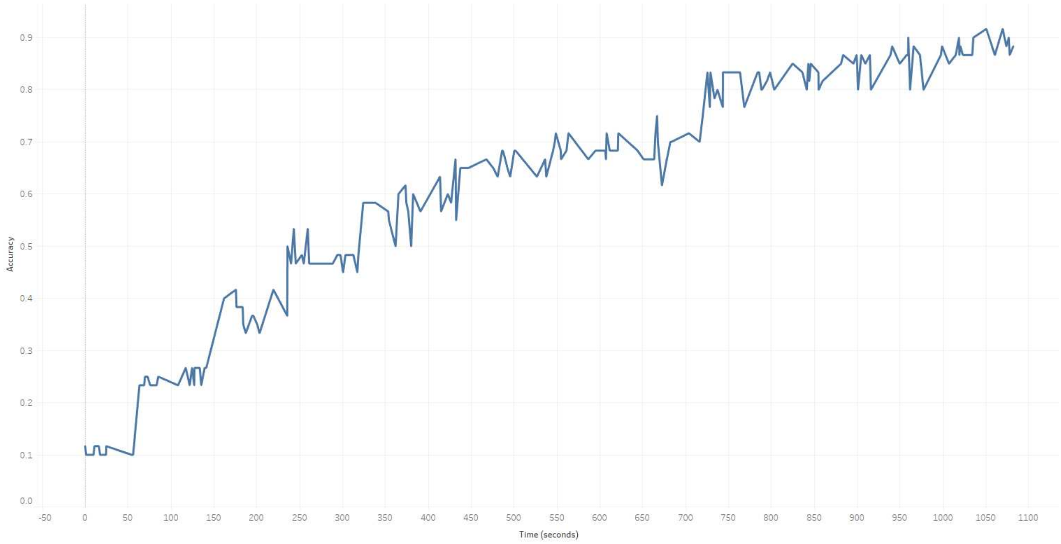

**Figure 5.21:** Accuracy of last run against Time

then converges onto 0.99, barring the occasional randomly picked values due to an epsilon value of 0.01. Figure 5.21 shows how the accuracy of the final run of our algorithm works. It reaches an accuracy of 0.9 at around the 1050 second mark, which is not far behind the pure A-EASGD run for $\alpha = 0.99$ that reached this at around the 900-second mark. Hence, it is evident from these results that Reinforcement Learning can be used to predict the best possible $\alpha$.

# Chapter 6

# Conclusions and Future Work

This study describes ORCA, a scalable deep learning framework for Apache Storm. It employs federated learning and reinforcement learning techniques to enable it to (1) deal seamlessly with addition and removal of worker nodes as the underlying architecture scales in and out, (2) ensure that acceptable accuracy is achieved whilst being elastic, (3) ensure that throughput does not fall as the arrival rate of tuples increases and the underlying architecture scales in and out and (4) predict the best possible value for the elasticity hyper-parameter used by the underlying architecture. Using DL4J and Elastic Averaging Stochastic Gradient Descent on top of a Storm topology enables scalable deep learning over apache storm [RQ1]. The reinforcement learning-based approach successfully predicts the best value for the elasticity hyper-parameter over for A-EASGD over the MNIST dataset [RQ2]. We have evaluated how ORCA performs under various circumstances and how it compares to similar non-distributed settings in terms of accuracy. We have also evaluated how well it performs in a scalable environment. Our results show that ORCA is capable of scaling up and down according to the demand whilst ensuring that the underlying algorithm converges while taking most of the input tuples into account while providing said accuracy.

## 6.1 Future Work

ORCA employs a naive Parameter Server that offers consistency by employing locking of most values while they are being updated. Smarter parameter server approaches that use advanced consistency and concurrency mechanisms like the approach mentioned in [57] could be employed to improve the performance of ORCA further. Techniques to reduce the number of combinations with the parameter server can also be employed to increase performance further. Model parallelism, similar to that of [42], can also be employed to reduce the load on individual worker nodes for large and complex models. Additional Federated Learning techniques that can be incorporated by ORCA's Lifecycle can also be implemented and evaluated. Reinforcement Learning can be extended, applied, and evaluated for more strategies and datasets. Momentum based EAMSGD [11] can be implemented over DL4J, which would drastically increase the performance of the algorithm in terms of convergence.

# Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[2] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[3] The Apache Software Foundation. Apache hadoop. https://hadoop.apache.org/, December 2019.

[4] The Apache Software Foundation. Apache spark. https://spark.apache.org/, December 2019.

[5] Khoshgoftaar T.M. Richter A.N. et al Landset, S. A survey of open source tools for machine learning with big data in the hadoop ecosystem. In *Journal of Big Data*. 2 2015.

[6] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, pages 1273–1282, 2017.

[7] The Apache Software Foundation. Apache storm. https://storm.apache.org/, December 2019.

[8] The Apache Software Foundation. Lifecycle of a storm topology. https://storm.apache.org/releases/current/Lifecycle-of-a-topology.html, December 2019.

[9] Josh Patterson Alex D. Black, Adam Gibson. Deeplearning4j. https://deeplearning4j.org/, December 2019.

[10] Django Software Foundation. Django. https://www.djangoproject.com/, December 2019.

[11] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 685–693. Curran Associates, Inc., 2015.

[12] Google Brain Team. Tensorflow. https://www.tensorflow.org/, December 2019.

[13] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. An overview of the granules runtime for cloud computing. *in IEEE International Conference on e-Science, Indianapolis, USA.*, pages 412–413, 12 2008.

[14] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Charles Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.

[15] Thilina Buddhika and Shrideep Pallickara. Neptune: Real time stream processing for internet of things and sensing environments. *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium., Chicago, USA.*, 2016.

[16] Geoffrey Fox, Sang Lim, Shrideep Pallickara, and Marlon Pierce. Message-based cellular peer-to-peer grids: Foundations for secure federation and autonomic services. *Future Generation Computer Systems*, 21:401–415, 03 2005.

[17] Geoffrey Fox, Shrideep Pallickara, and Xi Rao. Towards enabling peer-to-peer grids. *Concurrency - Practice and Experience*, 17:1109–1131, 06 2005.

[18] Ahmet Uyar, Shrideep Pallickara, and Geoffrey Charles Fox. Towards an architecture for audio/video conferencing in distributed brokering systems. In *Communications in Computing*, pages 17–23, 2003.

[19] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569, Dec 2017.

[20] Sangmi Pallickara, Shrideep Pallickara, and Marlon Pierce. Scientific data management in the cloud: A survey of technologies, approaches and challenges. *In Handbook of Cloud Computing Boston, MA: Springer*, 2010.

[21] Yogesh Simmhan, Sangmi Pallickara, Nithya Vijayakumar, and Beth Plale. Data management in dynamic environment-driven computational science. *International Federation for Information Processing Digital Library; Grid-Based Problem Solving Environments;*, 239:317–333, 11 2007.

[22] Sangmi Pallickara, Marlon Pierce, Qunfeng Dong, and Chinhua Kong. Enabling large scale scientific computations for expressed sequence tag sequencing over grid and cloud computing clusters. *In Eighth International Conference on Parallel Processing and Applied Mathematics (PPAM), Wroclaw, Poland*, 01 2009.

[23] S. Jensen, B. Plale, S. L. Pallickara, and Yiming Sun. A hybrid xml-relational grid metadata catalog. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 8 pp.–24, Aug 2006.

[24] Sangmi Pallickara, Shrideep Pallickara, and Milija Zupanski. Towards efficient data search and subsetting of large-scale atmospheric datasets. *Future Generation Comp. Syst.*, 28:112–118, 01 2012.

[25] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Exploiting geospatial and chronological characteristics in data streams to enable efficient storage and retrievals. *Future Generation Computer Systems*, 29:1049–1061, 06 2013.

[26] Daniel Rammer, Walid Budgaga, Thilina Buddhika, Shrideep Pallickara, and Sangmi Pallickara. Alleviating i/o inefficiencies to enable effective model training over voluminous, high-dimensional datasets. *Proceedings of the IEEE International Conference on Big Data (IEEE BigData), Seattle, USA.*, 2018.

[27] M. Malensek, S. Pallickara, and S. Pallickara. Minerva: Proactive disk scheduling for qos in multitier, multitenant cloud environments. *IEEE Internet Computing*, 20(3):19–27, May 2016.

[28] M. Malensek, S. Pallickara, and S. Pallickara. Evaluating geospatial geometry and proximity queries using distributed hash tables. *Computing in Science Engineering*, 16(4):53–61, July 2014.

[29] M. Malensek, S. Pallickara, and S. Pallickara. Analytic queries over geospatial time-series data using distributed hash tables. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1408–1422, June 2016.

[30] S. Mitra, P. Khandelwal, S. Pallickara, and S. L. Pallickara. Stash : Fast hierarchical aggregation queries for effective visual spatiotemporal explorations. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, Sep. 2019.

[31] Walid Budgaga, Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. A framework for scalable real-time anomaly detection over voluminous, geospatial data streams: Scalable anomaly detection over voluminous geospatial data streams. *Concurrency and Computation: Practice and Experience*, 29:1–16, 03 2017.

[32] Thilina Buddhika, Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Synopsis: A distributed sketch over voluminous spatiotemporal observational streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2552–2556, 08 2017.

[33] D. Rammer, T. Buddhika, M. Malensek, S. Pallickara, and S. Pallickara. Enabling fast exploratory analyses over voluminous spatiotemporal data using analytical engines. *IEEE Transactions on Big Data*, page (To Appear), 2019.

[34] Kevin Bruhwiler and Shrideep Pallickara. Aperture: Fast visualizations over spatiotemporal datasets. *IEEE/ACM Conference on Utility and Cloud Computing., Auckland, New Zealand.*, 2019.

[35] Samuel Armstrong, Kevin Bruhwiler, and Sangmi Lee Pallickara. Rapid, progressive subgraph explorations for interactive visual analytics over large-scale graph datasets. In *Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, BDCAT '19, page 1–10, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Walid Budgaga, Matthew Malensek, Sangmi Pallickara, Neil Harvey, F. Breidt, and Shrideep Pallickara. Predictive analytics using statistical, learning, and ensemble methods to support real-time exploration of discrete event simulations. *Future Generation Computer Systems*, 56:360–374, 03 2016.

[37] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *in COMPSTAT*, 2010.

[38] Lei Mao. Data parallelism vs model parallelism. https://leimao.github.io/blog/Data-Parallelism-vs-Model-Paralelism/, December 2019.

[39] Robin C. Geyer, Tassilo J. Klein, and Moin Nabi. Differentially private federated learning: A client level perspective, 2019.

[40] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[41] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology*, 10:1–19, 01 2019.

[42] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q.

Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[43] F. Sattler, S. Wiedemann, K. Müller, and W. Samek. Robust and communication-efficient federated learning from non-i.i.d. data. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–14, 2019.

[44] L. WANG, W. WANG, and B. LI. Cmfl: Mitigating communication overhead for federated learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 954–964, July 2019.

[45] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R. Ganger, Phillip B. Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 629–647, Boston, MA, March 2017. USENIX Association.

[46] Mehryar Mohri, Gary Sivek, and Ananda Theertha Suresh. Agnostic federated learning. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4615–4625, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[47] Michael Blot, David Picard, and Matthieu Cord. Gosgd: Distributed optimization for deep learning with gossip exchange, 2018. cite arxiv:1804.01852.

[48] Z. Han and M. Xu. Machine learning techniques in storm. In *2015 Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 139–142, Dec 2015.

[49] The Apache Software Foundation. Trident api. https://storm.apache.org/releases/1.2.3/Trident-API-Overview.html, January 2020.

[50] The Apache Software Foundation. Machine learning for flink. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/index.html, December 2019.

[51] Favio Vázquez. Deep learning with apache spark — part 1 - towards data science. https://towardsdatascience.com/deep-learning-with-apache-spark-part-1-6d397c16abd, December 2019.

[52] François Chollet. Keras. https://keras.io/, December 2019.

[53] Soumith Chintala Gregory Chanan Adam Paszke, Sam Gross. Pytorch. https://pytorch.org/, December 2019.

[54] The Apache Software Foundation. Pyspark. https://spark.apache.org/docs/2.2.0/api/python/pyspark.html, December 2019.

[55] Theodora Brisimi, Ruidi Chen, Theofanie Mela, Alex Olshevsky, Ioannis Paschalidis, and Wei Shi. Federated learning of predictive models from federated electronic health records. *International Journal of Medical Informatics*, 112, 01 2018.

[56] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 1223–1231. Curran Associates, Inc., 2013.

[57] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 583–598, Berkeley, CA, USA, 2014. USENIX Association.

[58] Wikipedia Contributors. Reinforcement learning algorithms. https://en.wikipedia.org/wiki/Reinforcement_learning, December 2019.

[59] Wikipedia Contributors. Q-learning. https://en.wikipedia.org/wiki/Q-learning, December 2019.

[60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[61] Xingping Dong, Jianbing Shen, Wenguan Wang, Yu Liu, Ling Shao, and Fatih Porikli. Hyperparameter optimization for tracking with continuous deep q-learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[62] Lokesh Gupta. Rest apis. https://restfulapi.net/, January 2020.

[63] The PostgreSQL Global Development Group. Postgresql. https://www.postgresql.org/, January 2020.

[64] The University of Waikato. Weka 3. https://www.cs.waikato.ac.nz/ml/weka/index.html, January 2020.

[65] University of Waikato. Moa – machine learning for data streams. https://moa.cms.waikato.ac.nz/, January 2020.

[66] Wikipedia Contributors. Ensemble learning. https://en.wikipedia.org/wiki/Ensemble_learning, January 2020.

[67] Robi Polikar. *Ensemble Learning*, pages 1–34. Springer US, Boston, MA, 2012.

[68] Chi-Hung Hsu, Shu-Huan Chang, Jhao-Hong Liang, Hsin-Ping Chou, Chun-Hao Liu, Shih-Chieh Chang, Jia-Yu Pan, Yu-Ting Chen, Wei Wei, and Da-Cheng Juan. MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning. *arXiv e-prints*, page arXiv:1806.10332, Jun 2018.

[69] P. Neary. Automatic hyperparameter tuning in deep convolutional neural networks using asynchronous reinforcement learning. In *2018 IEEE International Conference on Cognitive Computing (ICCC)*, pages 73–77, July 2018.

[70] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[71] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research).

[72] Dell emc. https://www.dellemc.com/en-us/index.htm, January 2020.

[73] Long short-term memory. https://en.wikipedia.org/wiki/Long_short-term_memory.

[74] Mountain data group. http://mountaindatagroup.com/, January 2020.

[75] National Climatic Data Center. Noaa dataset. https://www.ncdc.noaa.gov/cdo-web/datasets, January 2020.