Cuttlefish: A Library For Building Elastic Distributed Neural Networks

Teresa Nicole Brooks*, Rania Almajalid†, Yu Hou‡ and Abu Kamruzzaman§
Computer Science, Pace University
Pleasantville, NY
Email: *tb93141n@pace.edu, †ra56319p@pace.edu, ‡yh50276p@pace.edu, §ak91252p@pace.edu

Abstract—This paper will discuss the implementation of Cuttlefish, a library for building configurable, fault tolerant, elastic, distributed neural networks. Cuttlefish is built on top of the Distributed TensorFlow library. It uses Docker containers to represent units or neurons in the a neural network. More specifically each container (neuron) will run as a service that can communicate with other neurons in the network. Our neural network will be a n layered network, implemented and configured to run on Amazon Web Service's Elastic Container Service (ECS).

1 Introduction

Machine learning and more specifically the use of Neural Networks have many applications in both the research and commercial software. Though most machine learning techniques and algorithms employed today were developed over 20 years ago the rise of cheap, powerful processors (GPUs and CPUs) and higher capacity storage has allowed these techniques and algorithms to be used at scale.

In this paper we will discuss the implementation of Cuttlefish, a library for building configurable, fault tolerant, elastic distributed neural networks. We will also discuss the technologies used to implement this library which includes the use of clustered Docker containers, where each container represents a neuron in the network, distributed TensorFlow and Amazon Web Service's Elastic Container Service (ECS).

2 Background

In this section we will briefly discuss the technologies used to build Cuttlefish. In later sections we will discuss in more detail how these technologies are used.

2.1 Deep Neural Networks

A neural network [1] is a set of algorithms, that is primarily designed to perform pattern recognition. It does data interpretation through a kind of machine perception or clustering input. Neural networks help us classify as well as cluster data. It is called neural network because it is loosely inspired by neuroscience. The motivation for the development of neural network technology stemmed from the desire to develop an artificial system that could perform "intelligent" tasks similar to those performed by the human brain. Neural networks resemble the human brain in the following two ways: Firstly, a neural network acquires knowledge through learning. Secondly, a neural network's knowledge is stored within inter-neuron connection strengths known as synaptic weights. The true power and advantage of neural networks lies in their ability to represent both linear and non-linear relationships and in their ability to learn these relationships directly from the data being modeled.

Deep feed forward networks [16], also often called feed forward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. A feed forward network is said to be deep when it has more than one hidden layer. A lot of the benefit in deep neural networks comes from the ability of lower layers to learn representations that the higher layers can then use to perform their classification [2][3].

2.2 Docker

Docker is the world's leading software containerization platform [4]. Docker is open source, designed to make it easier to create, deploy, and run distributed applications inside lightweight Linux containers. It provides a way to run applications securely isolated in a container by packaging it and all its dependencies as well as libraries. Docker containers can wrap the software into a self-contained unit with a complete file-system that contains everything needed to run: operating system, code, runtime, system tools, system libraries, and configured system resources, etc. By using Docker containers, we can deploy, and back up a workload regardless of environment quickly and easily more than using virtual machines.

Docker allows to change any application dynamically by adding new capabilities and scaling services which makes containers more portable and flexible to use. Docker also has a mechanism for configuring and spinning up containers that should be clustered together or that are dependencies of one another.

One of Docker's advantages is simplified maintenance which means Docker minimizes the efforts and risks associated with application dependencies. The other advantage is increasing developer productivity because it decreases the time that is spent setting up new environments.

2.3 TensorFlow

TensorFlow is a framework for building Deep Learning Neural Networks [5]. It was developed by engineers and researchers working on the Google Brain Team within Google's Machine Intelligence research organization. It is an open source software library for machine learning computation using data flow graphs. Data flow graphs describe mathematical computation with nodes and edges. Each node in the graph represents mathematical operations, while edges represent the relationships between nodes. A computation expressed using TensorFlow can be executed in one or more GPUs or CPUs in a desktop, mobile device, or server with a single API.

The initial open-source release of TensorFlow supports multiple devices (CPUs and GPUs) in a single computer which is called the single-machine or single node implementation, while the distributed version supports multiple devices (CPUs and GPUs) across a cluster of machines [6].

2.4 Amazon EC2 Container Service (ECS)

Amazon EC2 Container Service (ECS) is high performance container management service which supports Docker containers through Management Console or Command Line Interface. It is highly scalable and allows applications to run easily on a managed cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances. Amazon ECS enables applications to be scaled without any additional complexity from a single container to thousands of containers across hundreds of instances [3]. Amazon ECS allows launch and stop container-based applications with simple API calls.

3 Motivation

The motivation for Cuttlefish was born from the observation that neural networks at their core are computational graphs and more specifically directed computational graphs, where each neuron in the graph is a single computation unit. This observation coupled with the understanding that in order to train models and perform inference on new data presented to these models at scale, in a production environment, requires implementing neural networks in a way that enables us to infinitely scale the execution of the these computation graphs in way that allows us to leverage all available system resources.

Tasks for training models and performing inference in neural networks are inherently parallelizable. Hence, it should be possible to implement a library that allows users to create a fault tolerant, elastic, distributed neural network, using configurable hyperparameters to drive the dynamic creation of a directed graph of neurons (a neural network). To implement this library we need a way to represent each node in the network physically; as a device or machine where computation execution will take place. We also need a centralized database to store intermediate values for the weight parameters each neuron calculates during model training. Moreover, the neurons in the network would need an efficient mechanism to communicate and pass data from one neuron to another. Lastly, we would need a mechanism to automate configuring and "spinning up" a neural network. Below we briefly discuss the technologies and approaches needed to implement such a library.

3.1 Physical Representation of Neurons - Docker Containers

The use of Docker to represent a physical neuron was an natural choice because Docker containers are self contained units that enable you to provide everything an application would need to run and nothing more; this includes an operating system, file system (volatile file system), and any needed software, frameworks or tools. Using Docker's Compose tool [7] containers can easily be configured as dependencies of one another, allowing you to create a cluster of containers that can be spun up together and that can communicate with one another.

3.2 Centralized Storage Of Weighted Model Parameters - Distributed NoSql Database

For our centralized parameter store, because the data is not highly relational and we would need a scalable solution distributed key-value store such a Amazon Web Services' DynamoDB would be a good fit.

3.3 Interneuron Communication

For inter neural network communication we would need an Inbound and output queue for each neuron in the network in order to pass data such as computed weights, and training model data from node to node. To implement such communication fault tolerant message broker technologies such as rabbitmq [8] or Apache Kafka [9] are good choices.

3.4 Automation of "Spinning" Up Elastic Neural Networks

Infrastructure orchestration software such as Kubernetes, Mesos and Amazon Web Service's Elastic Container Service are tools used in both test and production environments to automate orchestrations for creating, configuring and managing docker containers, and hence makes them a natural fit for this task.

4 Initial Proposed Approach

During the initial research phase of this project, we discovered Google's Distributed TensorFlow project in spirit was very similar to the initial idea that motivated our research. Distributed TensorFlow allows you to create a cluster of servers, where each worker process task is associated to a server in the cluster.

Distributed TensorFlow though it is primarily used for modeling machine learning computational models is generic enough to be considered a general purpose distributed computing library. It offers most of the properties we thought were paramount in implementing a library to enable building elastic, distributed neural networks [3]: fault tolerance, a means of sharing common calculated parameters among devices, cross device communication, as well as distributed execution of a computation graph among nodes in a TensorFlow cluster [10]. Hence, we decided to use Distributed TensorFlow as the base for the Culttlefish library.

Note that Cuttlefish defines a docker container as a single server in a TensorFlow cluster. Hence a Cuttlefish distributed neural network is simply defined as TensorFlow cluster.

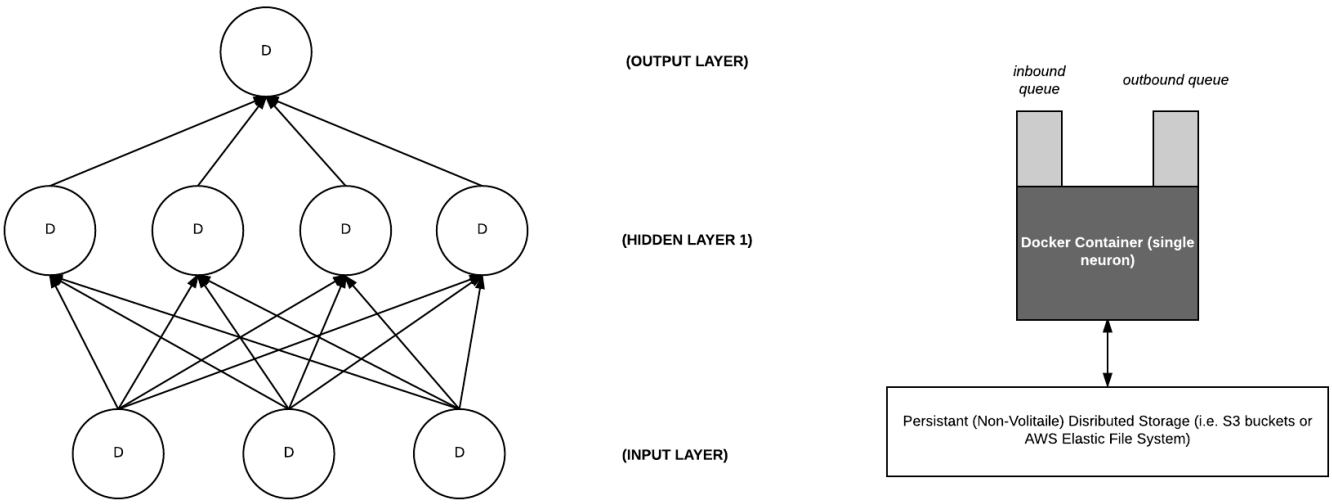

Fig. 1. Diagram example neural network nodes as docker containers. Note, the persistent distributed storage represent non-volatile storage for localized data per docker container. Persistent storage is needed to provide fault tolerance for shared and unshared data.

4.1 Computation Graph Distribution

TensorFlow uses a greedy heuristic algorithm called the ”placement algorithm” [5] to determine how a computation graph will be distributed for execution among all available devices. Default supported device types are CPUs and GPUs, there is also a registration mechanism so that users can implement and register their own device types [5]. TensorFlow provides an interface for users to influence how the computation graph is distributed, by allowing them to give ”hints and partial constraints” to the algorithm [5].

Cuttlefish, we propose using the above described functionality along with the configuration of each docker container’s system resources (memory, number of CPUs etc) to attempt to force TensorFlow’s placement algorithm to map one node in the computation graph to one Docker container. This is will test the viability of our motivation of using a single Docker container to represent a single computational unit in a distributed neural network (neuron).

4.2 Configuring Neural Network (Hyperparameters)

With Cuttlefish users define the configuration and shape of their neural network’s hyperparameters in a yaml file. By defining these parameters as ”code”, versioning of these parameters is simple, this also allows a user to automate building these configuration files as a tasks in a larger workflow where hyperparameters are being tested for a given set of training data.

4.3 Automation & Orchestration: Creating Docker Containers As Per Cuttlefish Configuration File

Cuttlefish’s ”build” functionality will use the Amazon Web Services’ Elastic Container Service’s (ECS) API [11] and it’s user defined elastic neural network configuration files to configure and build a distributed neural network using a cluster of docker containers (TensorFlow cluster).

With this approach, Cuttlefish takes the paradigm of infrastructure as code and applies it to the configuring and building of a distributed neural network as a cluster of resources readily available for computation tasks.

Note, though we are using AWS’ ECS [12] for orchestration for this proof of concept, tools like Kubernetes and Mesos are better choice as they offer finer grain control over configuring resource allocation per container. Fine grain control of such resources would allow for configuring and tuning system resources per neuron type, thus making the required system resources fit the computation task being performed by a particular neuron. This level of control would be useful when implementing neural networks such as convolutional neural networks, where different types of neurons perform different computational tasks and hence are likely to have different resource needs.

4.4 Data Set

We are using the MNIST data set for training distributed neural networks created by Cuttlefish. ”The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 example.” [13][14] We chose the MNIST data set because it is well documented, there is extensive benchmark data for it and TensorFlow as the full data set in a preprocessed ready to use format [14].

5 System & Application Architecture

In this section we will discuss the details of Cuttlefishs system and application architecture, design decisions as well as the challenges and limitations of employed technologies.

5.1 System Architecture

In this section we will discuss the details of Cuttlefishs system architecture.

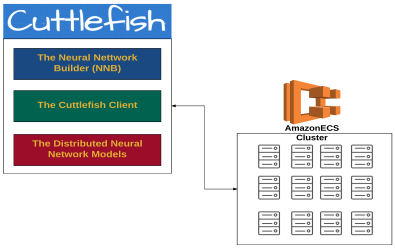

Fig. 2. Figure shows the major components of Cuttlefish.

5.1.1 Amazon EC2 Container Service (ECS)

The primary motivation for Cuttlefish is create a library that models distributed neural networks as computational graphs where each neuron is a single computational unit in the network and by using Docker containers to represent a physical neuron in a neural network it allows us to create highly configurable neural networks that can be spun up as infrastructure. In order to support this functionality, Cuttlefish must implement automated creating its cluster of containers.

Amazon Web Services EC2 Container Service (ECS) is the heart of Cuttlefish's system architecture. ECS enables users to easily automate the orchestration, configuring and deploying of a cluster of docker containers. ECS also contains a repository service to store the images used to build Docker containers. Cuttlefishs computational graphs are distributed and run on an elastic container cluster, where each distributed TensorFlow container runs on a single Elastic Compute Instance. Though the initial approach was to run all Docker distributed TensorFlow containers on one EC2 instance, in order to limit the financial resources required during the initial development phase, as well as limit unknown technical limitations or challenges, we discovered this approach was not possible due to limitations of the ECS service.

ECS is opinionated and makes assumptions about the likely use cases for the service. One assumption and major limitation of the current version of the service is by default containers listening to the same port, and hence running the same application(s) do not need to reside on the same EC2 instance. There are two solutions to address this limitation. One solution is to use dynamic port mapping by leveraging the AWS application load balancer service to map a single port number to multiple containers on the same EC2 instance. The other is to distribute the clusters docker containers, such that one container runs one EC2 instance. We chose the latter solution as it offered the most flexibility as it enabled Cuttlefish to maintain control how it distributes computational graphs and data.

5.1.2 Amazon Auto Scaling Groups

The ability to create new Docker containers based on neural network configuration files is a key feature of Cuttlefish. In order to support this feature our system infrastructure must support infrastructure elasticity. To implement this we employed the use of Amazon Web Servers Auto Scaling Groups feature. Auto Scaling Groups allow you to treat a collection of EC2 instances as "a logical grouping for the purposes of instance scaling and management." [22]

The ECS service allows users to create as many Docker container instances as needed as long as there is are underlying EC2 instances to support them. By creating an Auto Scaling Group, we configured the underlying infrastructure to support a configurable, maximum number of EC2 instances to support our container cluster. This is a key design decision as we would be unable to dynamical scale our cluster without it.

5.1.3 Logging & Monitoring

By default AWS allows the use of the CloudWatch service to support logging and monitoring ECS containers and their corresponding EC2 instances. To use this feature we simply configure a log group in the underlying task definition that is used as a template to create container instances in the Cuttlefish ECS cluster.

5.1.4 Amazon Elastic Compute (EC2) Virtual Machine Specifications

For proof of concept testing we are using T2 EC2 instances to host the distributed TensorFlow Docker containers. T2 instances are "instances that provide a baseline level of CPU performance with the ability to burst above the baseline." [aws-specs] The Amazon Machine Image (AMI) is an elastic container service optimized customer Amazon Linux distribution. Below is a summary of EC2 instance specifications for machines used to house the Cuttlefish ECS cluster.

- Box Type: t2.medium
- AMI: Amazon Linux AMI 2016.09.a x86_64 ECS HVM GP2
- Storage: EBS storage only
- Memory: 4GB
- CPU Units: 2

5.2 Application Architecture

Cuttlefish as a proof of concept application, was written to test our ideas regarding a novel way of modeling and executing neural networks as a distributed computational graph, where each neuron in the network is modeled as a single computational unit; as well as test automating spinning up distributed neural network as infrastructure using configuration files that describe the neural networks hyperparameters, number of epochs etc. Cuttlefish is currently comprised of three major components, the Neural

Network Builder (NNB), the Cuttlefish Client and the Cuttlefish neural network models.

The Neural Network Builder (NNB), is responsible for reading a given neural network configuration file and creating new instances of TensorFlow Docker containers using the configured number of hidden layers, number of nodes per layer and number a TensorFlow (shared) parameter servers to calculate how many Docker containers to create in it's ECS cluster. The NNB client drives this process of "spinning" distributed neural networks. The NBB client runs as a script and takes a neural network configuration file that describes the network to "spin" up as a command line argument.

```
neural_network:
  num_of_hidden_layers: 3
  num_nodes_per_layer: 5
  num_classes: 10

training:
  epochs: 300
  data_batch_size: 100
  num_parameter_servers: 2

ecs:
  task_definition: 'cuttlefish-task-dev2:7'
  cluster: 'cuttlefish-dev'
  auto_scaling_group_resource_id: 'EC2-Cuttlefish'
```

Fig. 3. Figure shows a simple deep feed forward neural network configuration. Note, the configuration includes hyperparameter values, TensorFlow and ECS related configuration values.

The Cuttlefish Client, is responsible for driving the process of creating, distributing and training distributed neural network models. The Cuttlefish Client like the NNB client, runs as a script and takes a neural network configuration file that describes the network to "spin" up as a command line argument. It then uses this configuration file to determine which distributed neural network model it will run, which auto scaling group should be used to get the ECS instance IP addresses to be used to configure the TensorFlow server's workers and shared parameter servers. Lastly, the Cuttlefish Client runs the distributed neural network model. The Cuttlefish Client also drives execution of the distributed neural network models.

Like many modern machine learning and deep learning frameworks and applications, Cuttlefish is implemented in python 3.5 and python 2.7 [15].

6 Challenges and Limitations Using Distributed TensorFlow

We encountered several technical challenges and limitations, specifically focused around using distributed TensorFlow to test our approach of distributing neural network nodes as a single computational unit to be executed. This section briefly discusses the challenges encountered and limitations of using TensorFlow to implement Cuttlefish.

6.1 Distributed TensorFlow Manual Process For Cluster and Server Setup

At the time of writing this paper running distributed TensorFlow is a very manual process. For each worker and shared parameter server (container) we needed to execute the client on each Docker container which limited our ability to effectively test distributing our computational graph, where each neuron runs on it's on Docker container. With this limitation in order to fully explore our ideas using TensorFlow as it would have required a significant development effort to properly orchestrate the process of not only "spinning" up a cluster of Docker containers but also automating the process of pulling the latest Cuttlefish code and executing / training the intended model.

We also encountered some challenges when trying to force TensorFlow's placement algorithm to distributed the graphs using our approach. This challenge stems from the framework's opinions on how graphs should be distributed. TensorFlow supports between-graph replication and in-graph replication and neither approach fully supports the approach we wanted to explore. Despite these challenges, we were able to successfully train a model against the MNIST dataset, using a basic, feed forward, deep neural network using both distributed and non-distributed Tensoflow against the MNIST dataset.

6.2 Limited Documentation

Though the TensorFlow white-paper, website and github repository are useful resources, they were limited in the depth and scope needed to implement any solutions beyond the basic examples implemented in the documentation. Given that TensorFlow, though gaining in recent popularity is still a fairly new technology which limits documentation (blog posts, tutorials etc) found in the general development community.

7 Results

Single-machine Results			
# of Nodes	# of Layers	# of epochs	Accuracy
5	3	1000	88.61%
500	4	10	94.99%
50	3	2000	95.82%
784, 2500, 2000,1500,1000,500,10	5	30	96.95%

Fig. 4. Results for Single-machine Testing.

Distributed version Results				
# of Nodes	# of machines	# of Layers	# of epochs	Accuracy
5	2	3	1000	50.89%
500	2	4	10	71.75
50	2	3	2000	85.75%

Fig. 5. Results for Distributed version Testing.

The model has been tested on a single computer which is called the single-machine or single node implementation

and on the distributed version which supports multiple devices (CPUs) across a cluster of machines. Fig. 4 illustrates the results for Single-machine testing while fig. 5 shows the results for distributed version.

8 Conclusion

Despite limitations and technical challenges we conclude based on the insights gains through the initial work on Cuttlefish that the problems it attempts to solve are worthy of continued exploration. Although we were unable to confirm the efficacy of our proposed approach to computational graph distribution there is evidence that this solution is not only possible to implement but a worthy approach to explore.

8.1 Future Work

In this section we briefly discuss future work and research on Cuttlefish.

8.1.1 Implement Cuttlefish to Distributed Neural Networks Without TensorFlow

In order to fully explore our ideas regarding computational graph distribution we propose continued work on Cuttlefisth without leveraging TensorFlow. This will allow us to flexibility to explore all facets of the problem.

8.1.2 Implement A More Robust Orchestration and Automation Solution

We propose implementing a more robust orchestration and automation solution to replace our use of Amazon's Elastic Container Service (ECS). Although ECS is fairly easy to use, it has technical limitations that make it far less flexible and mature than competing products such as Kubernetes or Docker Swarm.

References

[1] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *NIPS*, 2014, pp. 3104–3112.

[2] A. Angelova, A. Krizhevsky, and V. Vanhoucke, "Pedestrian detection with a Large-Field-Of-View deep network," in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2015-June, no. June, 2015, pp. 704–711.

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, and Q. V. Le, "Large scale distributed deep networks," *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

[4] "Docker Product Site - What is Docker?" [Online]. Available: https://www.docker.com/what-docker

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, J. Shlens, B. Steiner, I. Sutskever, P. Tucker, V. Vanhoucke, V. Vasudevan, O. Vinyals, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *None*, vol. 1, no. 212, p. 19, 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[6] L. Rampasek and A. Goldenberg, "TensorFlow: Biology's Gateway to Deep Learning?" *Cell Systems*, vol. 2, no. 1, pp. 12–14, 2016.

[7] "Docker Documentation - Compose." [Online]. Available: https://docs.docker.com/compose/

[8] "Rabbitmq Product Page." [Online]. Available: https://www.rabbitmq.com/

[9] "Apache Kafka." [Online]. Available: https://kafka.apache.org/

[10] "Distributed Tensor Flow." [Online]. Available: https://www.tensorflow.org/deploy/distributed

[11] "AWS Elastic Container Service API." [Online]. Available: https://aws.amazon.com/documentation/ecs/

[12] "AWS Elastic Container Service Product Page." [Online]. Available: https://aws.amazon.com/ecs/

[13] "The MNIST Datebase Of Handwritten Digits." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[14] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[15] C. McLeod, "A Framework for Distributed Deep Learning Layer Design in Python," *arXiv cs.LG*, vol. 10, p. 07303, 2015. [Online]. Available: http://arxiv.org/abs/1510.07303