Team name: Arvind Fan Club
Team members:
- Arthur Liang (liangw6@cs)
- Bryan Van Draanen (bryanvd@cs)
- Andrew Wei (nowei@cs)
Repo: https://github.com/liangw6/byteps

**Simultaneous training and inference in ML**

## Overview

We planned and considered multiple ways to implement Live Inference in the context of BytePS, a distributed training framework and implemented one of them.

## Introduction

Machine Learning as a field is moving more and more towards Deep Learning and the application of Deep Neural Networks (DNNs). Training DNNs is very resource-intensive and can take numerous GPUs and multiple weeks to train. Many early models used a centralized system that controls several GPUs. Now, people are moving towards distributed training (BytePS, Nexus, etc.) and learning algorithms (Federated Learning) to further parallelize portions of the network (or for more privacy in the latter case).

Live Inference provides the ability to utilize the DNN while the model is being trained. Some motivations for why people might be interested in a system with Live Inference include: being able to use the model in applications while it has not finished training or in the context of continuous training, where we keep receiving new data to train with. This will probably not see use in contexts that require immediate results, like self-driving cars/trucks, but this is probably for the best since people would not want their self-driving car/truck to wait on receiving messages for the next action the vehicle should take. However, Live Inference should help in situations where some delay can be tolerated.

We wanted to come up with ways in which we can implement Live Inference in one of these distributed training systems. We chose to build upon BytePS, a distributed training framework from Bytedance, because it is at the cutting-edge of this area.

## Initial Thoughts

We were worried that using noisy weights from partially updated models would be a bad idea when implementing Live Inference because they may actually perform worse in terms of accuracy due to the randomness of popular gradient descent methods. This ideal case would be actually implementing Live Inference using only the model with the best accuracy on some held-out validation set. But that would often indicate replicating existing model, which, given the large memory footprint of many state-of-the-art neural network models, may indicate doubling the existing resources. On the other hand, the cost of accuracy may be worth it in some cost-constrained or resource-constrained situations if that cost allows adding real-time inference to training without any need for additional resources.

We approached this problem with two things in mind, how we would store the model used for inference and how we would actually serve inference requests once we receive them. We came into this thinking that the model might have been partitioned on different workers, but existing example python scripts from BytePS do not separate the model among worker machines. But rather, in both example training scripts for ImageNet and MNIST, the model parameters are replicated among workers but trained on potentially different set of data.

For storing the model, we thought about where the requests would be handled and the cost associated with each scheme. We figured that we could either store the model on the server or store the model on separate servers. For storing the model on servers, we thought about having a separate server for each worker that stored a copy of model weights for that layer or maybe just having a single server that had all the weights for the model. The main concern here was whether the inference would use resources and slow down the training. This is not a major concern because we do not plan on doing large-scale requests for inference calls on the model, but it could become an issue when operating at larger scale.

For inference, we came up with two scenarios based on how we planned on storing the model: Either we contact the server that held the model separately or contact the server that was also handling training. Contacting the server that held the model separately and performing inference on those would reduce the number of messages going in and out of the node that also manages training, but would incur additional rounds of communication for updates. On the other hand, handling the inference on the workers that are also training might take away from the resources used for training.

We kept these things in mind and currently opted to go with the one we felt we could most comfortably implement on BytePS. We wanted to show that Live Inference was possible while the model was still training, so we opted to keep the best-performing model on the server that it is training on and perform the inference using that model.

**Trade-offs**

**For Storing the Model**

- Store a copy of the best performing weights based on validation set data
  - Pros:
    - More robust to overfitting
    - Only uses as many workers as there were originally
  - Cons:
    - Needs to store a copy of the model parameters
    - Inference may utilize hardware, which may slow down training due to needing to transfer information if we need to handle a significant number of client requests
- Separate server for each worker that holds the best weights on validation set
  - Pros:
    - More robust to overfitting

        - Doesn't hinder training
    - Cons:
        - Needs to store a copy of the model parameters
        - Needs 2x more servers to handle inference, potentially more hardware as well
        - Requires at least 2n messages to update weights
- Directly using the weights as they're being trained.
    - Pros:
        - Does not need to copy model parameters
        - Only uses as many workers as there were originally
    - Cons:
        - May hinder training
        - May lead to inconsistent results if not done safely
        - Results will likely be much more noisy
- Storing the best model weights on a single server
    - Pros:
        - More robust to overfitting
        - Inference only has 2 message delays
    - Cons:
        - Only works for small models
        - Needs at least 2n messages to update weights
        - Could become a single point of failure, but we can just replicate it

**For Inference**

- Support live client inference through a custom client request relayed to the primary server
    - Pros:
        - Does not need extra servers for inference
        - Easier to build on top of existing framework
    - Cons:
        - May hinder training in worker nodes
        - There may be a lot of traffic on the primary server if there are many client requests
- Performing inference by messaging non-worker servers that stores weight information
    - Pros:
        - Doesn't hinder training
        - Easier to build separately, but needs to extend existing framework
    - Cons:
        - The existence of more servers to hold weights
        - Needs to relay updates to weights using messages

**Implementation Designs**

In this section, we will discuss our implementation design for real-time inference on existing BytePS framework. After many discussions, we have decided to implement the design where real-time inference and training are executed on the same set of machines, i.e. weights are not

kept separated. Although this design would have potential resource competition problem between train and real-time inference, it allows developers to train and do real-time inference without any additional resources, making it incredibly useful for many resource-constrained or cost-constrained situations. In addition, separating training and real-time inference on different sets of machines offers a fairly straight-forward approach, with exactly the same training and real-time inference setups as usual and the addition of sockets to send weights from training nodes to real-time inference nodes. On the other hand, doing training and real-time inference on the same set of machines raises many interesting questions, e.g. how much does real-time inference impede training, how much does training impede real-time inference?

With hosting both training and real-time inference on the same set of machines in mind, we have designed to implement real-time inference on BytePS as follows.

Each worker machine maintains at least two threads with access to the deep learning model as a shared variable. One thread runs training with gradient descent, i.e. writing to the model, while the other runs as a server to serve inference, i.e. read only to the model. In other words, for each worker machine, there may be potentially many readers, but only one writer.

Each client will contact all the worker machines with each real-time inference request. Each worker, upon receiving the request, will spawn a new thread, which will do a forward pass to get predictions for the request and send those back to the client. Since each worker sends one response to the client, the client will receive multiple, potentially different, predictions from all workers. The client will then choose the “consensus” prediction from all workers, e.g. the mode of worker predictions, as a final prediction for this specific real-time inference task.

Because this design requires each client to communicate with all worker machines, it indicates a throughput bottleneck. But since the neural network is “sharded” across all worker machines (the weights are technically replicated. But since each worker has different version of the weights, the client would not be able to see the full picture without going through all of them, so it is more similar to “sharding” weights across network than “replicating”), a client has to go through all worker machines to get the correct prediction for the inference request, so this cost is necessary and optimal. This could potentially be reduced by replicating weights on more machines, which can be a potentially interesting project for another time.

Considering latency, since the client sends one request and gets one response from each worker, the latency would only cost one Round Trip Time (RTT), thus also optimal.

Based on our overall design, we have also eliminated the need for workers to communicate with each other during real-time inference, which not only greatly reduces latency but also allows us to demonstrate our real-time inference implementations with only one worker and potentially multiple clients.

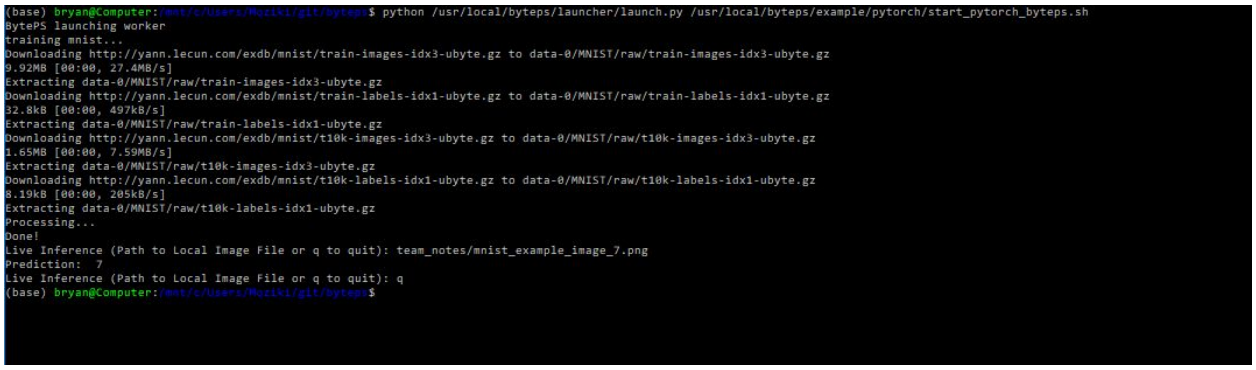
**Implementation Details**

We tried two different setups for our chosen implementation. The first functioned as a proof-of-concept for live inference using a simple single client console application, and the other communicated and handled multiple client requests across the network through a socket setup.

The console application makes the simplifying assumption that client requests are received from the command-line that spawned the distributed training job. However, note that this setup could easily replace the thread that listens to standard input with one listening to network sockets for inference requests. The interface operates by spawning a separate thread which initiates and executes the BytePS training protocol across multiple workers while the original thread waits for user input data associated with the inference request. The BytePS server training thread retains a copy of the best-performing model experienced during training which is directly accessible to the console application.

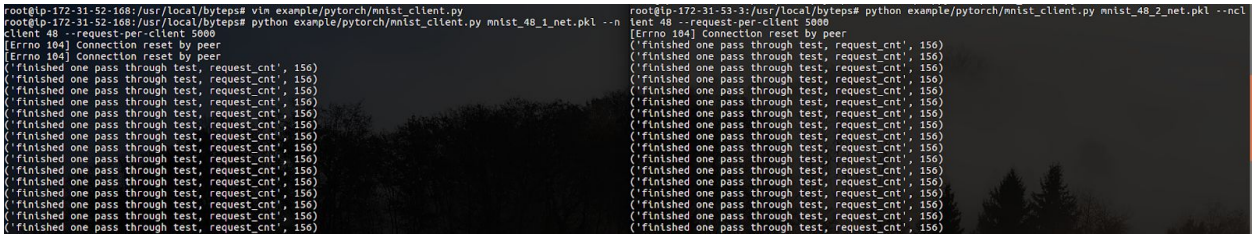
This simple live inference was created using the MNIST dataset as a demonstration of how it would operate. The user provides a file path to an image as the input for the request for model inference. Since the application is hosted on the server (rather than one of the workers), the server uses its stored best-performing model to complete the inference and provide a prediction to the client. Because training is distributed across workers, this only represents a bottleneck to the server in theory when weights are being aggregated to update the model. However, note that this is largely a limitation inherent to console applications and this style of live inference can be generalized to allow any worker to field client requests when retaining the model locally.

Below is an example simple live inference session on the BytePS console application using the MNIST dataset when querying a prediction for the digit “7”:

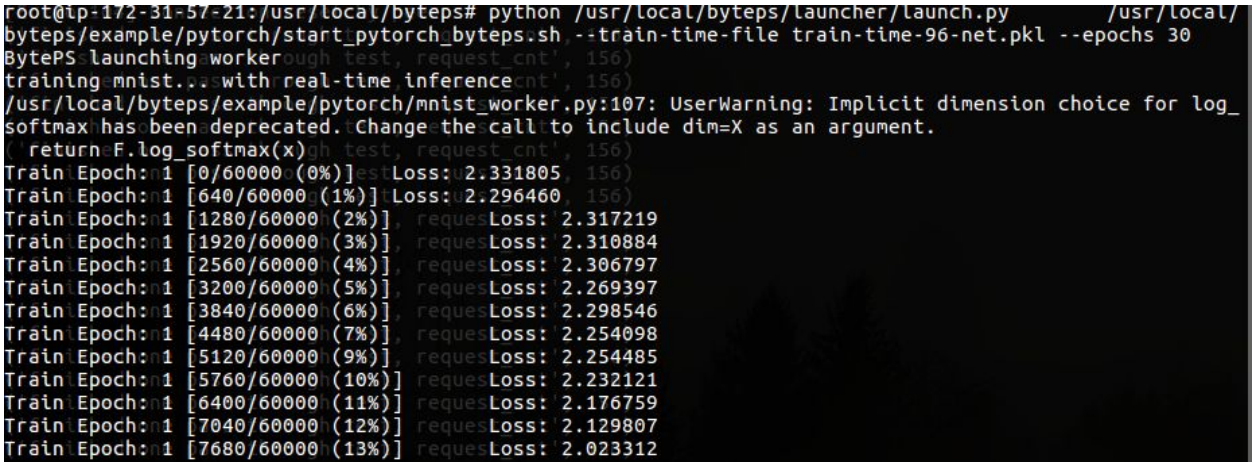



For the one using sockets, we have implemented two scripts, one for the worker and the other for the client. The worker script is our proof-of-concept product that runs on MNIST and supports both training and real-time inference. The client script, on the other hand, is meant to simulate client connections and real-time inference requests to workers. Each script can be used on multiple machines, for instance, we can have two client machines, each running a copy of client script and one worker machine, which runs a copy of worker script. For each client script, the user needs to specify the IP addresses of all workers. This allows each client to

contact all the worker machines and get predictions from all of them. The client script also uses multi-processing to simulate multiple clients, each with their own process, although the actual number of clients that are running in parallel at a time depends on the number of CPU cores. The below screenshot shows two client machines, each spawning 48 clients. So there will be a total of 96 user connections to each worker machine. Since each client machine runs in a 8-core CPU, that means, at most 16 clients are actively trying to communicate with the worker machine at any given time.




For the worker script, it does not need any special setup. Once started, It will listen to a certain port and spawns a new thread to handle each client. Worker script can also be run with special options to save the training time and validation accuracy that will be used later in the data analysis section. Below is a screenshot of launching the worker script. It is almost exactly the same output as normal training, but it will produce an error output if it encountered any while processing client requests.




**Problems Faced**

One of the biggest challenges in this project was to get the compute resources needed for this project. Because GPU-enabled machines, which are required for training machine learning models can be quite expensive on public cloud and also quite difficult to set up, our team spent a lot of time and effort to setup AWS spot machines for this project. Although we were able to a proof-of-concept product working, we do regret not developing the project more and analyzing data in a larger scale.

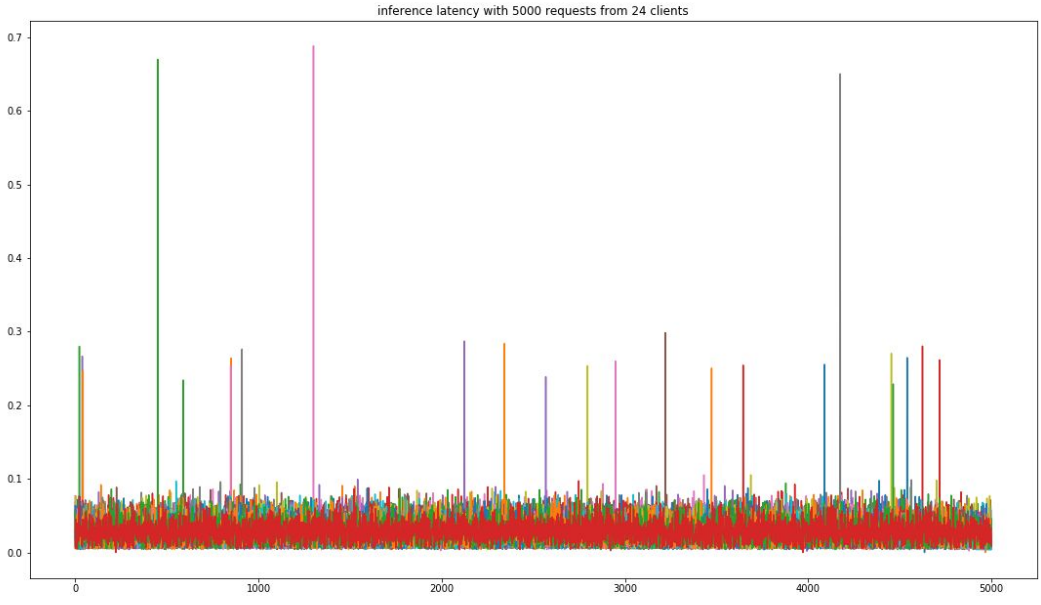
We also encountered many oversights in the BytePS codebase where the latest version was not thoroughly vetted on the GPU hardware we ran it on. We published some issues to the original repo and the developers then helped us modify the code and resolve the issues.

**Results**

We implemented real-time inference for MNIST with one worker and one client and first tested performance on a personal computer with a 4 core CPU and GTX 1070 GPU. Note that both the client and the worker are running in the same machine. Although running both client and worker on the same machine results in resource competition, it helps to eliminate factors such as network latency and network instability.

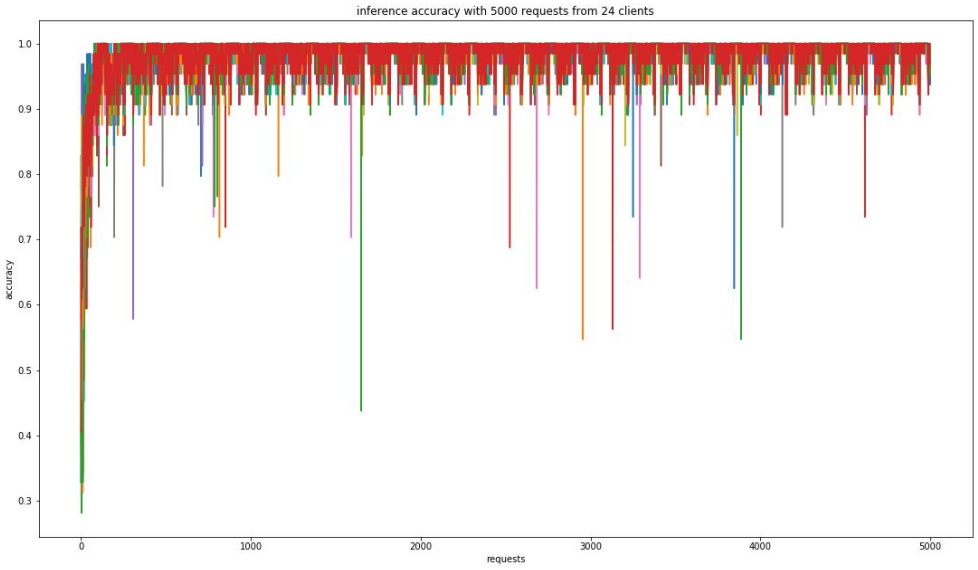
During the experiment, the client would spawn 24 threads, each trying to connect to the worker server and send MNIST images loaded from test part of MNIST dataset. Each client would send 5000 batches to the worker (one at time), each with the size of 64 images, and thus each client would go through multiple passes of the test set. On the worker side, the worker started training about the same as real-time inference comes in.

Below is a figure of the 5000 requests' latency for all 24 clients (each with a different color). Each latency is measured as the time duration between sending all images to the worker and receiving the prediction from the worker. As we can see, most requests were responded to within 0.1 seconds, while some did take a lot longer, up to 0.7 seconds.



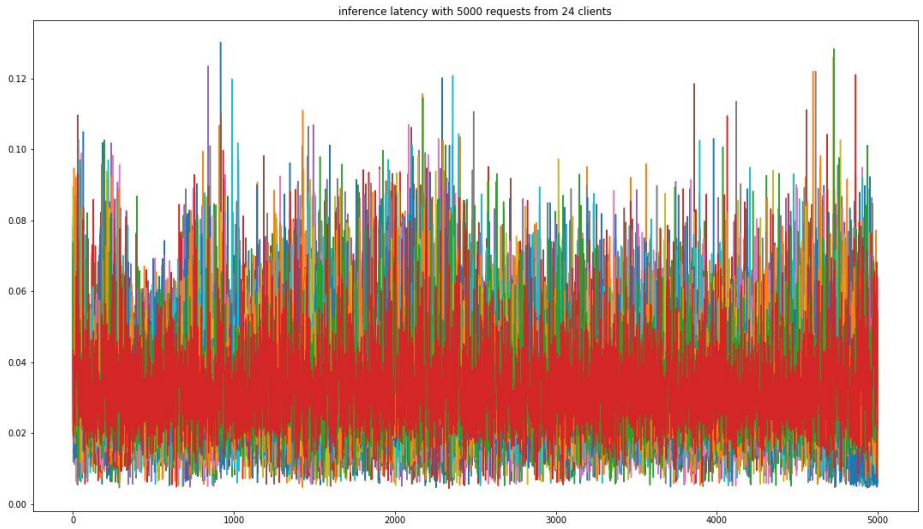

Below is a figure of the same experiment, but measuring the accuracy of each request, rather than latency. Because MNIST is a simple dataset, even though real-time inference requests have been sent since the beginning of the training process, the accuracy reached about 1.0

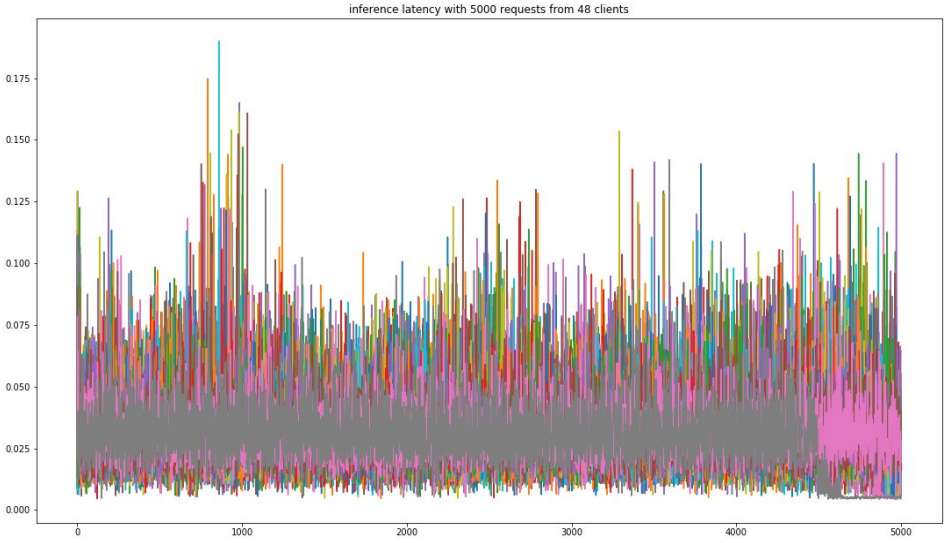
very fast.  However, it is also worth noting that some requests reach very low accuracy.  Those can be due to randomness within the network, or the use of unstable weights during inference, but their occurrences are only about 0.3% (to be exact, a total of 13 requests reached accuracy below 0.8 after epoch 1000).



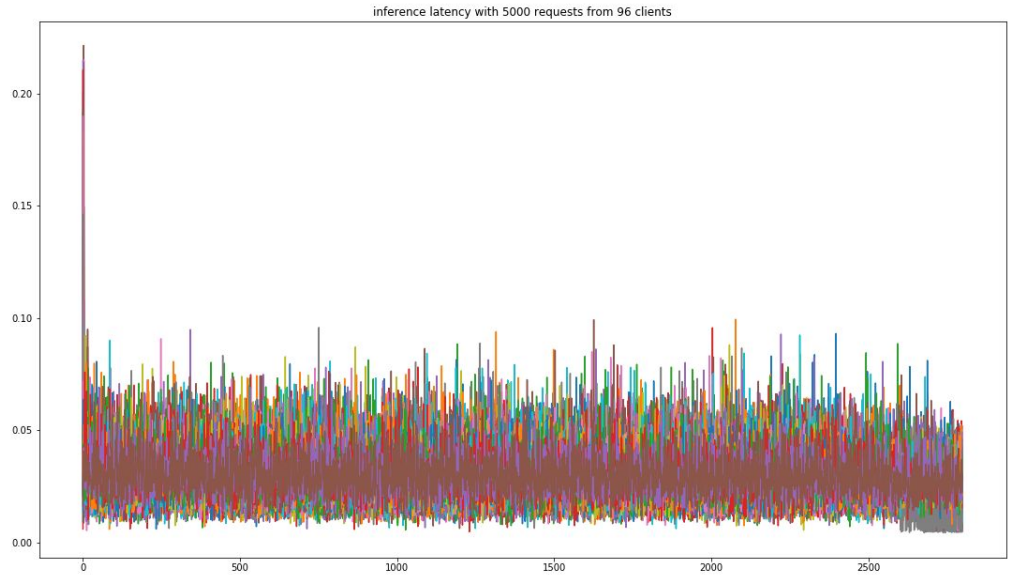
We then tested the same implementation on AWS machines.  In particular, we set up two AWS nodes, a c5.2xlarge (8 core CPU-only) node for the client and a p2.xlarge (4 core, CPU + GPU) machine for the worker.

Below are 2 figures that show the latency experienced by each client request.  Compared to the figure running locally, there are very few latency increases.  In addition, possibly thanks to the data center network in AWS, the variation in latency is very small.  Note that since all 24 or 48 clients run inside the same client machine, doubling the number of clients did not increase latency substantially.



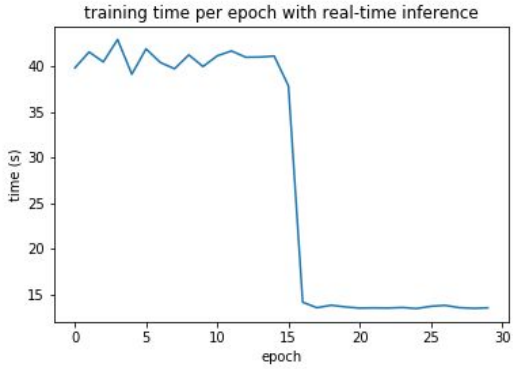
However, since an 8 core CPU client machine can only simulate 8 clients submitting requests at the same time, we decided to further examine the throughput bottleneck by including another 8 core client machine, which allowed up to 16 clients submitted requests at the same time.

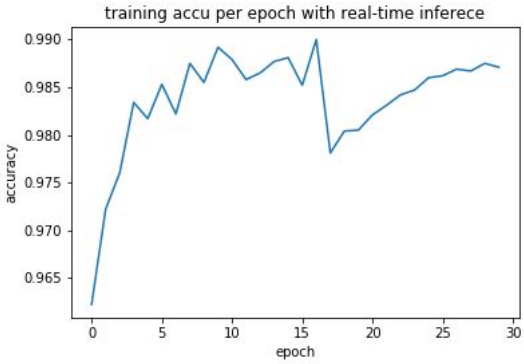



Although it may be hard to see from the figure, it does take each client almost twice as long to finish the same number of requests, confirming our hypothesis of there being a throughput bottleneck.

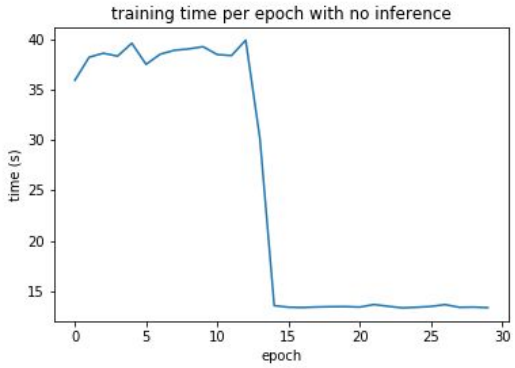
In all cases, the inference accuracies are very similar to the one running on personal laptop; so we will, instead, add those figures in the Supplementary Figures.

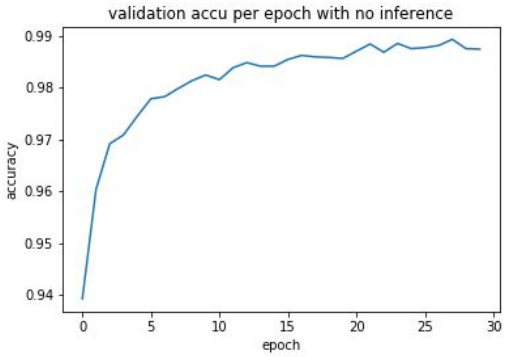
Another question we want to answer is how much does real-time inference impede training, in terms of both accuracy and training time. We used the initial experiment setup of 24 clients on 8 core client machine and a 4 core CPU and GPU worker machine. Below are a figure of training time per epoch and a figure of validation accuracy that was computed after each epoch. The data were collected while the worker machine was doing real-time inference. Note that the sharp decrease of training time in the middle was because real-time inference clients have finished




As a comparison, we have also run a similar setup but without real-time inference. While the client was still sending requests and worker responding to them, the worker did not actually input that number into the model, but instead return an arbitrary prediction. Below are two similar figures for this experiment:




There are a few interesting discoveries. Compared to the placebo inference, including actual inference with that serves 24 clients, increased training time for about 10% per epoch. But if the inference server was completely dormant, without any new threads to serve clients, the training time can decrease more than 50%. So the most cost of supporting real-time inference comes from the implementation of the socket server. In addition, it also means that our implementation of real-time inference can elastically adjust training time given the amount of client request.

Comparing the validation accuracy figures for both experiments, our implementation of real-time inference seems to interfere with the training of the model. In particular, we think the current

implementation has a race condition that allowed the gradients of validation data to flow back to the model. But this can be easily solved with a lock and in some cases, such as online learning, this may be useful feature.

**Future Work**

- Analyzing and understanding the performance of our design in terms of both accuracy and latency on more complex neural networks (BytePS currently does currently provide an example script for a more complex model, resnet50, on a much larger dataset. We wanted to try ImageNet, but since ImageNet is currently having trouble with copyright issues, none of the team members can have access to ImageNet data, and thus are unable to train or test it.)
- Solving throughput bottleneck by developing a new distributed Machine Learning protocol that would replicate model parameters to different machines
- Implementing Live Inference in another distributed training framework
- Coming up with more ways to perform live inference
- Testing this out in an application setting and seeing what happens
- Maybe implement our own distributed training framework with Live Inference in mind at the start
- Look more into Federated Learning and how they handle updates and training compared to distributed training frameworks

**Conclusion**

Our goal was to come up with ways to implement Live Inference in a distributed training setting and test some of those out. We believe we succeeded in this goal. In particular, we were able to implement real-time inference on existing BytePS framework. Although we only implemented real-time inference on a fairly simple model for MNIST dataset and on one worker, the techniques we designed and implemented can be easily transferred to other deeper and more complex neural networks and allow for the inclusion of more workers. In addition, our design allows users to receive prediction within one Round Trip Time. This low latency property satisfies the real-time requirement that we started with for inference. However, we do recognize the drawback of our design: since each client needs to contact each worker machine, there will be an unfortunate bottleneck on worker machines, limiting the throughput of real-time inference. We think this bottleneck can be potentially solved by replicating weights to different worker machines. But we were not able to dive into that topic for this project

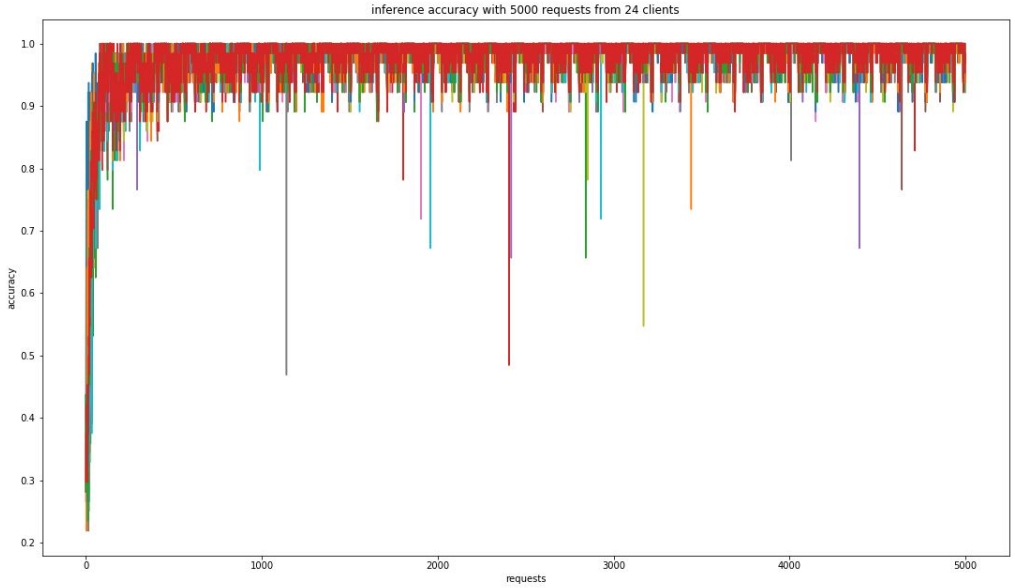
Currently, access to multiple GPUs is generally something that is only financially feasible for large companies that had a large number of GPUs to begin with, since renting them is generally expensive. Distributed training frameworks will probably become more mainstream as GPUs and resources become more accessible. Then the question of how to integrate training with live inference may become a bigger concern when models are going to be trained continuously.

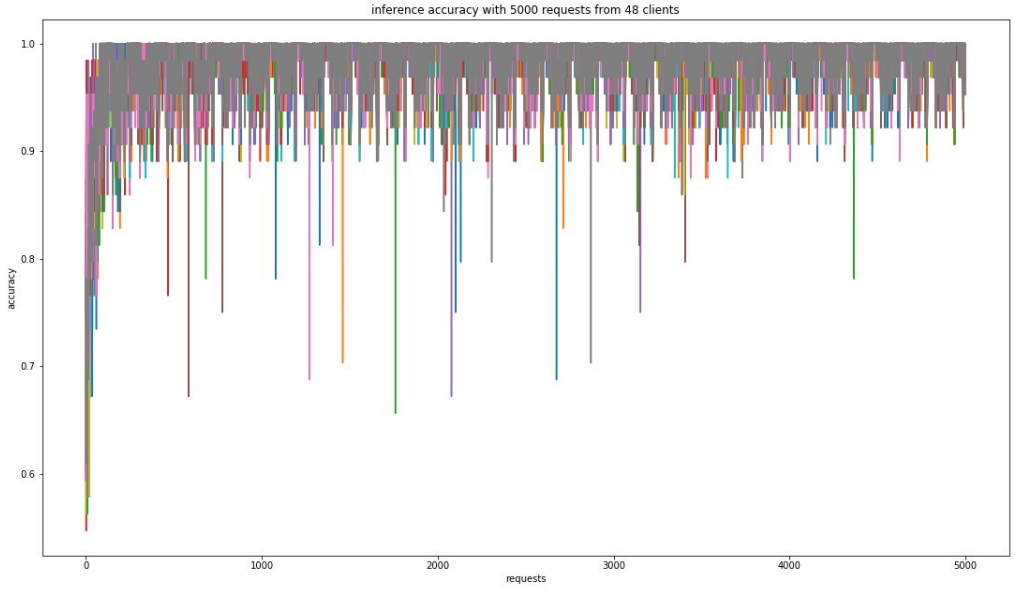
Furthermore, as DNNs and other machine learning models require higher dimensionality and become more complex, the time required to train them will increase as well. BytePS and other

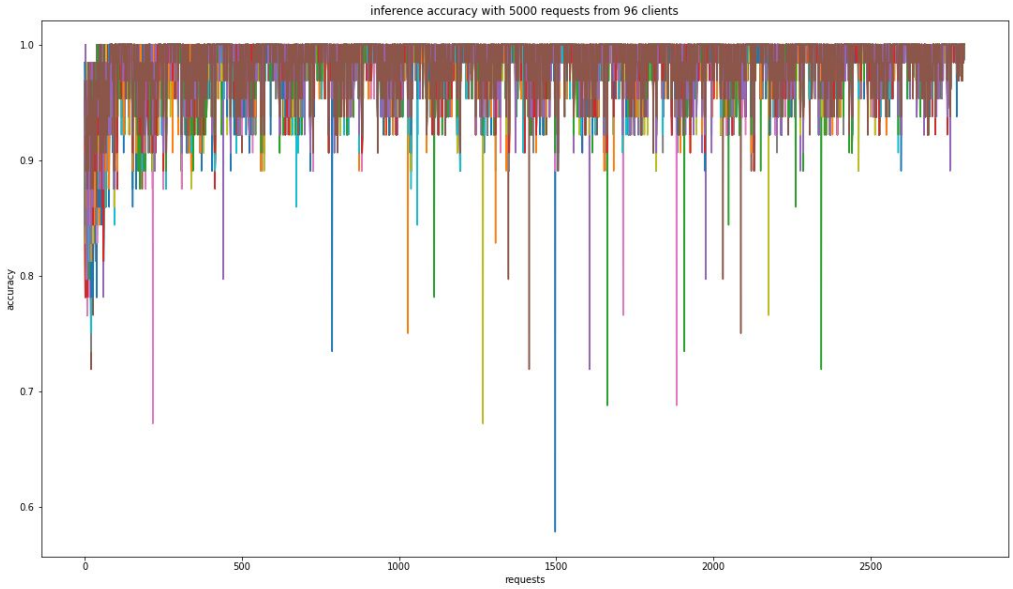
forms of distributed training algorithms will become more commonplace and integral to reducing these training times. However, reducing training times through distributing the work alone can only scale horizontally so much. As a result, having a means of querying and performing live inferences on these models as they train in a low-latency, effective way will be an important requirement of these systems. Extracting inferences from models as early as possible in the training process makes an entire class of machine learning models that may not have been viable previously, especially those ideal under continuous training conditions, now feasible options to pursue.

## Supplementary Figures

Inference Accuracy figures for experiments running on AWS machines




inference accuracy with 5000 requests from 96 clients
accuracy
requests
1.0
0.9
0.8
0.7
0.6
0
500
1000
1500
2000
2500