The Scalable Machine Learning with JAX

Introduction

Hi, this is Sai Htaung Kham from Applied Machine Learning Department from ZOZO NEXT. At ZOZO NEXT, we have many research algorithms under development and JAX is one of the libraries that has been used to facilitate our research. For those who are new to JAX, JAX is a Python library designed for high-performance numerical computation. If you are familiar with NumPy, you can think of JAX as similar to NumPy but a more powerful library. Unlike NumPy, JAX can run on multi-GPUs, multi-TPUs, and with Autograd which is very useful for Machine Learning research.

If you are already familiar with the NumPy library, JAX is very easy to adopt as it has mirrored most of the NumPy APIs. With Autograd, we can automatically differentiate native Python and NumPy functions. For the detailed functionalities of JAX, please refer to the official Github repository of JAX.

Why JAX in the first place?

When it comes to building a machine learning algorithm, most of us will quickly move to and focus on reliable ML frameworks such as Tensorflow and PyTorch. This is true and a good decision as mature ML frameworks have a mature ecosystem that is easy to integrate and maintain. At our lab, we do have many algorithms implemented using those frameworks. However, some algorithms were implemented in a pure Python programming language with the NumPy library. For example, we have an embedding algorithm designed by our researchers and engineers for internal use. As it was implemented in Python, it has some bottlenecks in computation runtimes. As you may notice, without using the framework, the hyperparameters updates and housekeeping jobs are done manually. There is a big bottleneck in the process and it is hard for us to try out new ideas which prevent us to move forward quickly. Moreover, we cannot scale the library and our training process as it is bounded to a CPU device. Even though we have successfully figured out to run the algorithm with multi-process using the shared memory architecture, it would be nice if we could scale out horizontally rather than vertically, running on multi hosts and devices such as GPUs, TPUs, etc.

So we consider finding a way to port the program to a different framework for high scalability and maintainability. The following are the characteristics of why JAX suits our purposes.

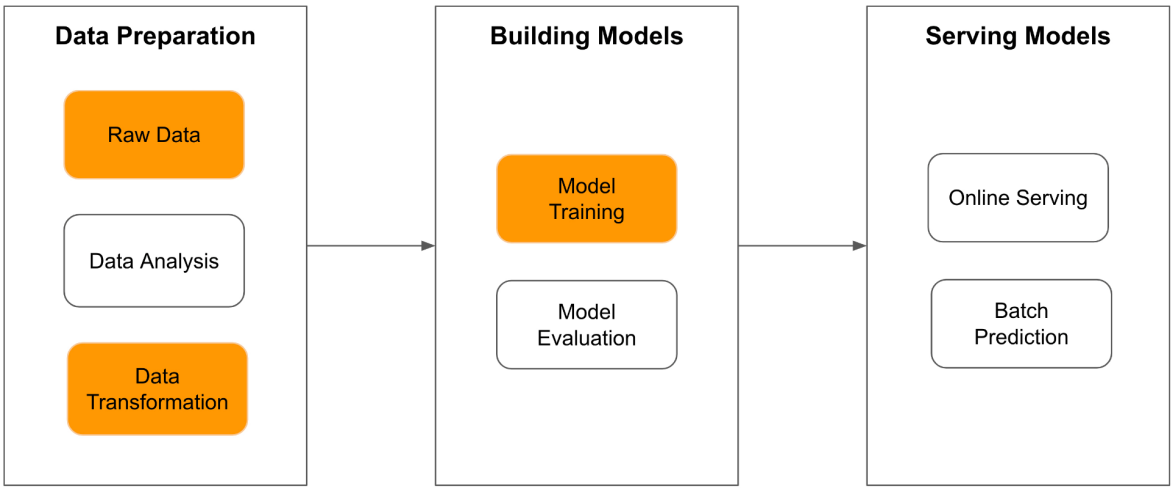
(1) Horizontal scalability with SPMD (single-program-multiple-data) architecture
(2) NumPy APIs mirrored
(3) Compatible with Python Programming Language
(4) Autograd support
(5) Have open-sourced an ecosystem of libraries around JAX (Flax, Haiku, etc)

What to expect in this blog?

To show how it is easy to code in JAX, in this blog, we will discuss how we can use the JAX library to perform Machine Learning and more importantly how to make it scale across the infrastructure that we have. When we talk about building Machine Learning models, we generally focus on how to create a model that could learn well and do well on real-world data.

During learning the theory and practicing the ML problems, we usually use the small data to reinforce our understanding. However, when we do ML problems in the real world with the actual data, we will face many difficulties such as the volume of the data, the size of the memory to hold the model, and the scalability of the machine learning training and evaluation. Thankfully with the revolution and pricing of today's cloud computing, scalable machine learning is accessible to everyone.

The typical Machine Learning Pipeline



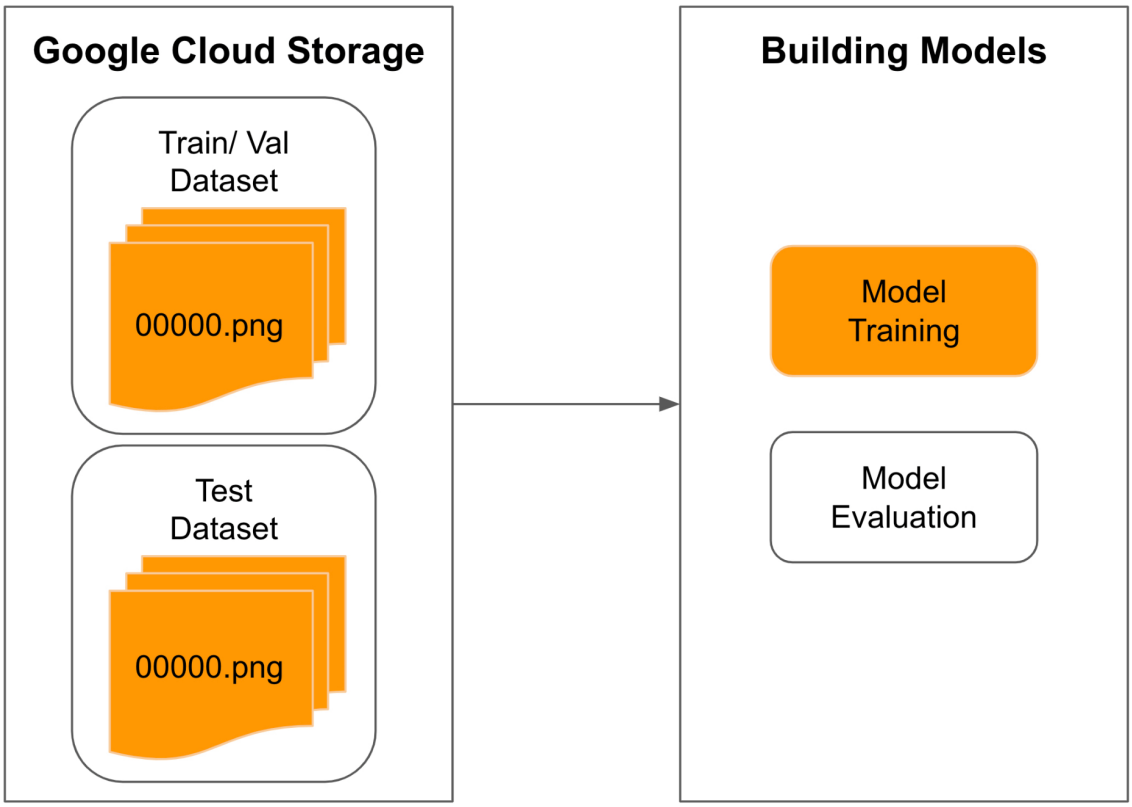
Typical machine learning pipeline

In this blog, we will mainly focus on the performance of the JAX library and how to make it scales on cloud computing. The typical machine learning pipeline consists of many stages and in the blog, we will mainly focus on data preparation and model training.

The Data, is it really big? When, Where, and How?

Before we begin, every engineer understands that data and quality data transformation is the heart of the success of a machine learning project. In the real world when we deal with the data for a specific ML project, it is rare to see a volume of the data that could fit and do the ETL (extract, transform and load) process on a single machine. At our lab, we widely rely on the cloud computing resources such as Google Cloud and AWS, which usually use cloud storage and cloud data warehouse to manage our data. For this blog, we will demonstrate how we scalably trained the multilayer perception with JAX on the famous MNIST handwritten digit dataset and discuss the results that we found.

Host the data on the cloud infrastructure



The integration of the data storage and machine that hosts the algorithm

I am chocking. Help needed!

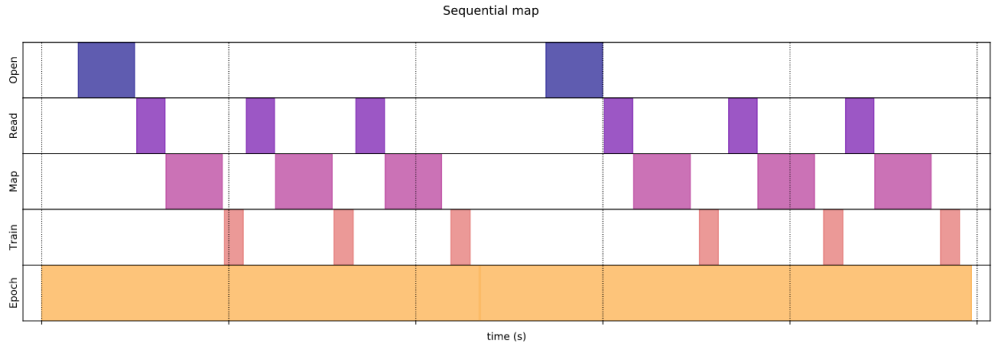
Cloud storage helps us a lot in storing a large amount of data that could not fit on a single machine. However, we still need to find an efficient way to read the data from the storage. One of the problems that most ML engineers discover during the training with GPU devices is that the GPU devices are not fully utilized and the training process is taking longer than it should be. Please observe the following profiling diagram of a TensorFlow model. Did you notice something?

The model profiling (Ref: https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras)

If you observe closely, you will notice that the GPU device is in an idle state most of the time while fetching the data from the disk. In general, we would like to keep the GPU device busy and underutilization most of the time. This shows that we have a bottleneck in our data input pipeline.

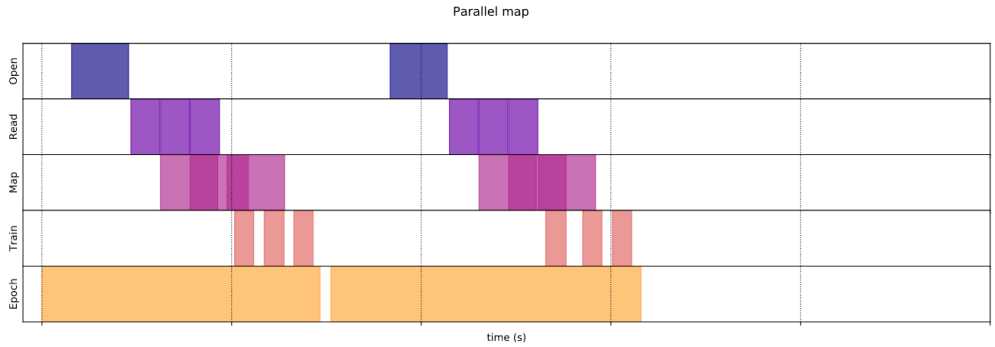
TF Data is that a Hero?

To alleviate the data input pipeline bottleneck, we will use the useful tool provided by TensorFlow APIs named TF Data. With the traditional method, we read the data from the disk sequentially as shown in the following diagram. The mapping process is just the transformation of the data such as normalizing, image augmentation, etc.



Fetching data sequentially to the model (Ref: https://www.tensorflow.org/guide/data_performance)

However, this method introduces the bottleneck as we noticed the training process is needed to wait for the data and leads GPU devices in an idle stage. We may prefer parallel reading and data transformation so that we could reduce the gaps during the training process like the following diagram.



Effective data transformation with TF Data Pipeline (Ref: https://www.tensorflow.org/guide/data_performance)

The TF Data pipeline components are reusable. We could apply them to the training and serving phase. The TF Data library utilizes the power of the host CPU cores to process the data in parallel. It is important to keep in mind that the better the CPU, the faster the data reading and preprocessing.

The other options that we could explore for the data preprocessing pipeline are to use the Apache Bean and TFX Transformation. However, we will not discuss these technologies in this blog. We will use the TF Data along with JAX for scalable machine learning.

Let it burns, let it burns on scale

With the effective data preprocessing pipeline in hand, we could move to the model training and evaluation steps. Some of the useful libraries of JAX are vmap and pmap. The vmap allows us to map our function into the vectorization manner without changing our code which speeds up our process while pmap distributes our function over multi-devices for distributed processing. In this blog, we used the vmap and pmap to facilitate the faster training process on multi GPU devices.

```python
import numpy as np
import jax.numpy as jnp
import jax

def convolve(x, w):
  output = []
  for i in range(1, len(x)-1):
    output.append(jnp.dot(x[i-1:i+2], w))
  return jnp.array(output)

x = np.arange(5)
w = np.array([3., 1., 3.])
batch_size = 10
xs = np.arange(5 * batch_size).reshape(-1, 5)
ws = np.stack([w] * batch_size)
print(f"The shape of the x and w : {xs.shape, ws.shape}")

print("Process each sample.")
for sample in xs:
    print(convolve(sample, w))

print("Auto-vectorization with vmap:")
print(jax.vmap(convolve)(xs, ws))
```

Auto-vectorization with vmap

```
The shape of the x and w : ((10, 5), (10, 3))
Process each sample.
[ 7. 14. 21.]
[42. 49. 56.]
[77. 84. 91.]
[112. 119. 126.]
[147. 154. 161.]
[182. 189. 196.]
[217. 224. 231.]
[252. 259. 266.]
[287. 294. 301.]
[322. 329. 336.]
Auto-vectorization with vmap:
[[  7.  14.  21.]
 [ 42.  49.  56.]
 [ 77.  84.  91.]
 [112. 119. 126.]
 [147. 154. 161.]
 [182. 189. 196.]
 [217. 224. 231.]
 [252. 259. 266.]
 [287. 294. 301.]
 [322. 329. 336.]]
```

Results of the comparison of vmap and per sample processing

With the power of auto-vectorization, we noticed that we don't need to change any bit of our code rather than wrapping our function under vmap API. This is very useful for us, especially during batch processing. There are still more features of vmap and don't forget to check it out at https://jax.readthedocs.io/en/latest/jax-101/03-vectorization.html.

The usage of pmap is quite similar to vmap. However, the pmap provides the collective ops, allowing our programs to communicate over multi-devices and perform the collective operations (MapReduce) such as collective sum, collective permutation, collective mean, etc. This API makes our program scale horizontally.

```
@partial(jax.pmap, axis_name="num_devices")
def update(params: Params, x: jnp.ndarray, y: jnp.ndarray) -> Tuple[Params, jnp.ndarray]:
    loss, grads = jax.value_and_grad(loss_func)(params, x, y)

    grads = jax.lax.pmean(grads, axis_name="num_devices")
    loss = jax.lax.pmean(loss, axis_name="num_devices")

    new_params = jax.tree_multimap(
        lambda param, g: param - g * step_size, params, grads
    )

    return new_params, loss
```

Appling the pmap on multi-devices

In the above code sample, we run the collective ops over the results returned from the loss_func function on different devices, perform the collective-mean operation and update our hyperparameters. We can run this piece of code on any machine without worrying about the number of accelerator devices we have. It will automatically scale out and manage by JAX in the background. However, we need to handle and match the device dimension according to the number of accelerator devices.

Time for actions

In this blog, we used the JAX library and built up the simple multilayer perceptron for MNIST handwritten digit dataset.

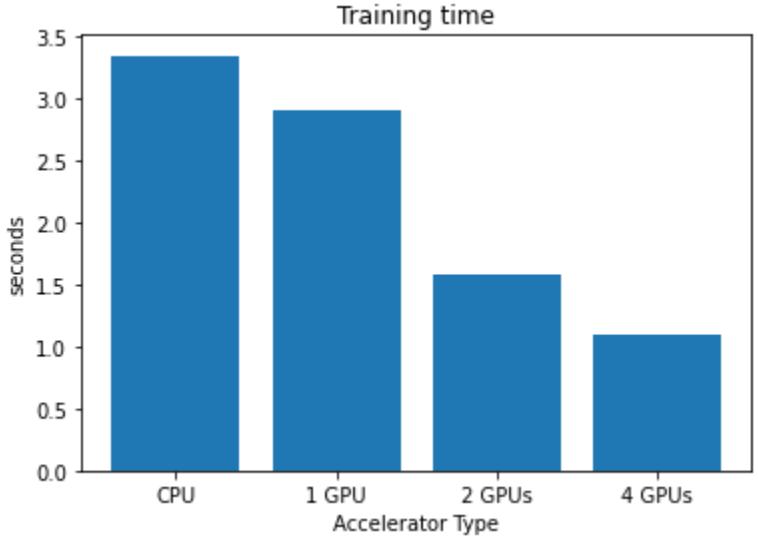
The infrastructure we used:

Hardware Name	Type
CPU	Intel(R) Xeon(R) CPU @ 2.30GHz 16 Cores
Memory	60 GB
GPUs	NVIDIA Tesla T4 (1, 2, 4 devices)
Storage	Google Cloud Storage

The MLP Parameters:

Type	Values
MLP layers	[784, 512, 10]
Learning rate	0.01
Number of epochs	50
Batch size	32
Input image dimension	28x28x1
Number of classes	10
Training image count	48000
Validation image count	12000

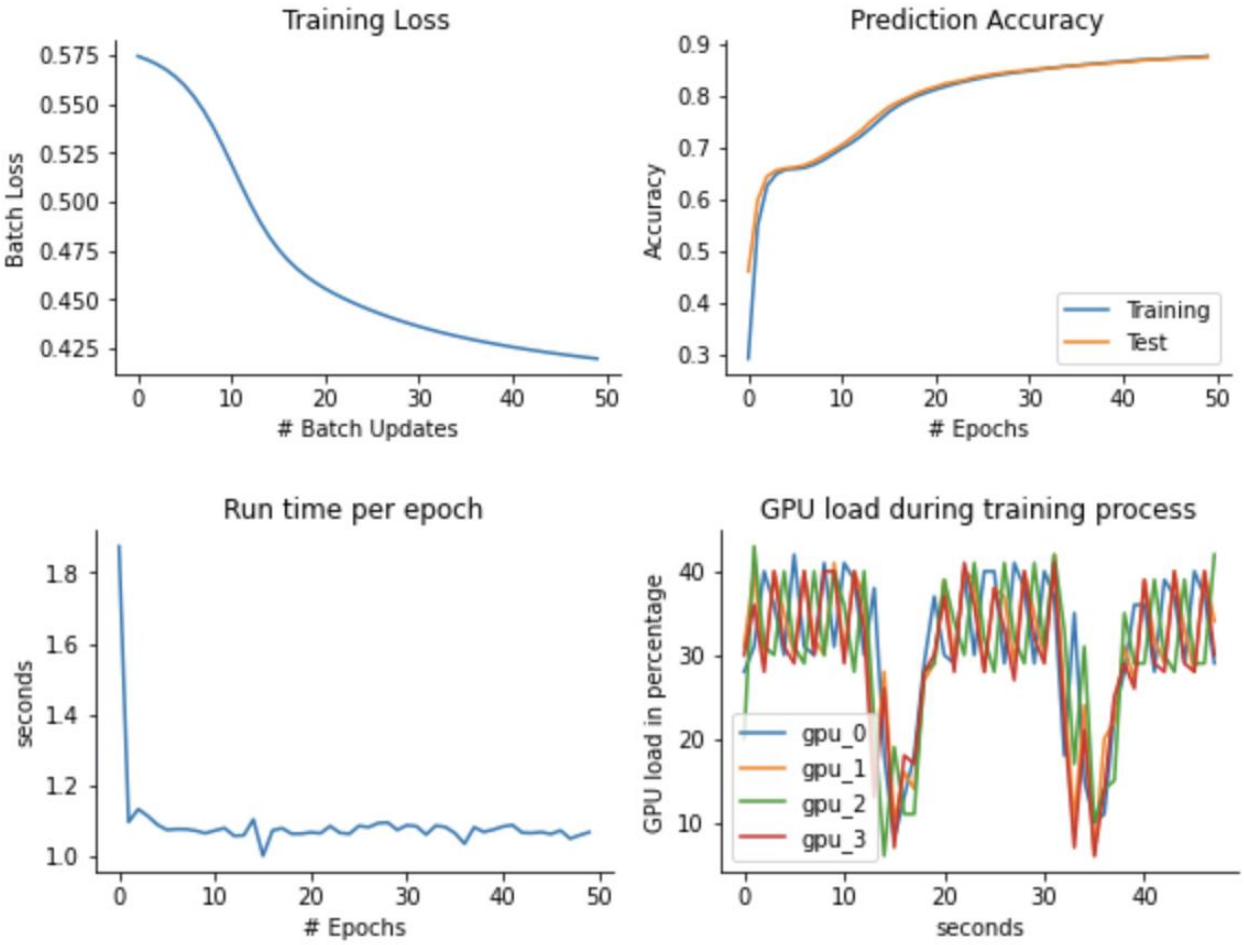
The following diagrams show the runtime comparison of the algorithm over different accelerators.



Average runtime over different accelerators for each epoch

We could observe that the MLP algorithm on CPU took around 3.34 seconds for each epoch while 4 GPUs took 1.09 seconds. This diagram shows that the more parallel processing we perform, the better runtime we have for this specific algorithm.

The following diagrams show the performance of the model training and runtime on each epoch with 4 GPU accelerators.



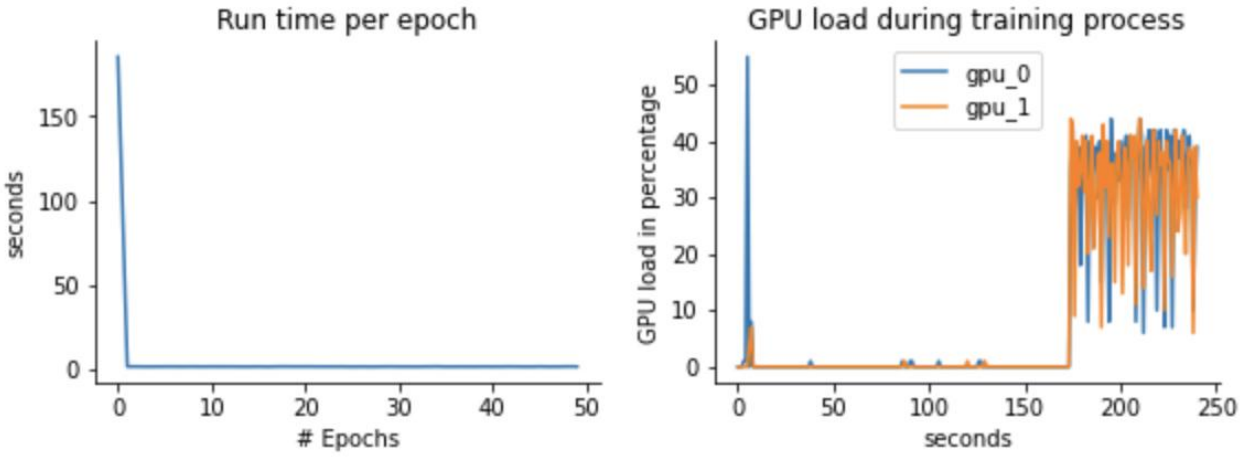
The training and runtime of each epoch on 4 GPUs

From the above diagrams, we observed that the model is learning well on the training dataset and doing well on the validation dataset. Moreover, the training processes are evenly distributed on all 4 GPU devices. It took around 1.8 seconds to finish the first epoch and around 1.09 seconds on the following epochs. For the first epoch, the program needs to remotely read the images from the cloud storage and do the preprocessing on them according to the pipeline data transformation. Later on, we use the cache function of the pipeline to cache the data locally so that it is ready for the following epochs which significantly reduces the runtime.

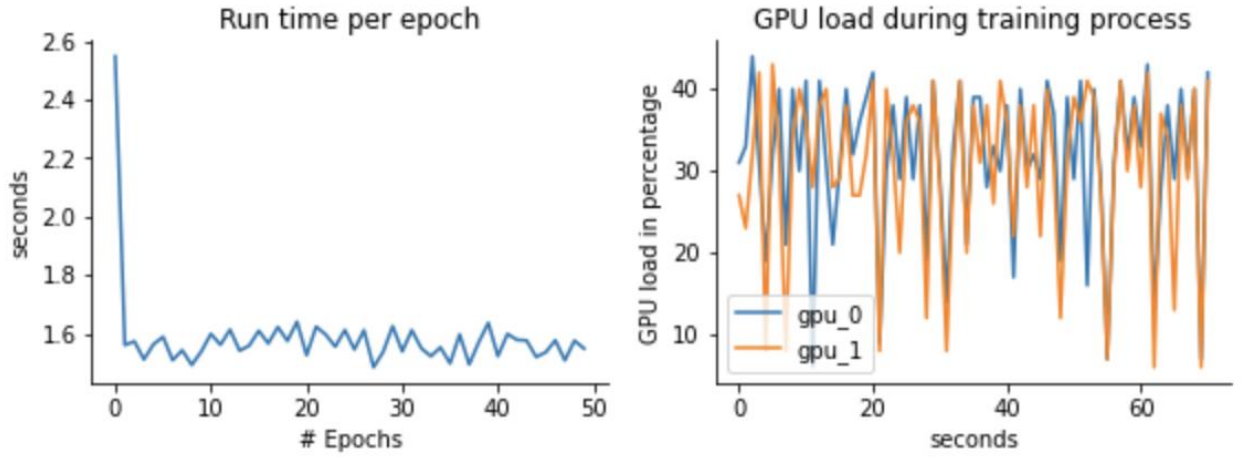
From the GPU usage perspective, we could see that the GPUs are busy from the beginning and has some peak around 1.87 seconds which is the end of the first epoch of the training. This tells us the GPUs (especially the gpu_2) are holding some processes simultaneously with the data loading and transformation from the pipeline. We can see that the training process is starting in parallel with the data pipeline loading the data from the remote

storage devices. The GPU devices are at their peak of around 40% and it is reasonable as the MLP layers are quite small to utilize all the GPUs at 100%.

One interesting fact we discovered is that if the location of the cloud storage is different from the machine where we host our algorithms, the remote data fetching could add extra time to the first epoch. This is normally the specification of the cloud infrastructure where it needs to download the data to the edge location where the user accesses them. After the first time accessing from the host machine, the data is stored on the edge location and it significantly improves the training runtime.



In case the training machine and cloud storage are on different regions or locations

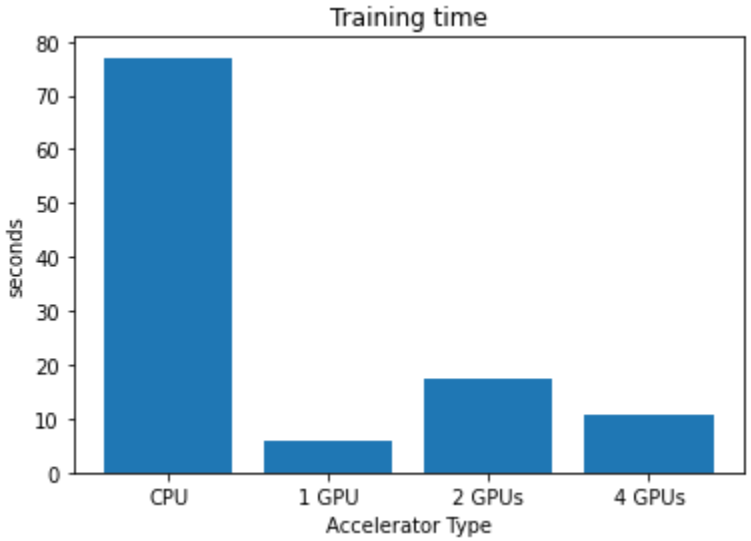


After the first access, due to the cache on the edge location, the runtime has improved

To verify that the bigger the calculation, the higher the GPU devices utilization, we have tested on the larger images with larger MLP layers. We keep the hardware specification the same as the previous experiment on the MNIST handwritten dataset.

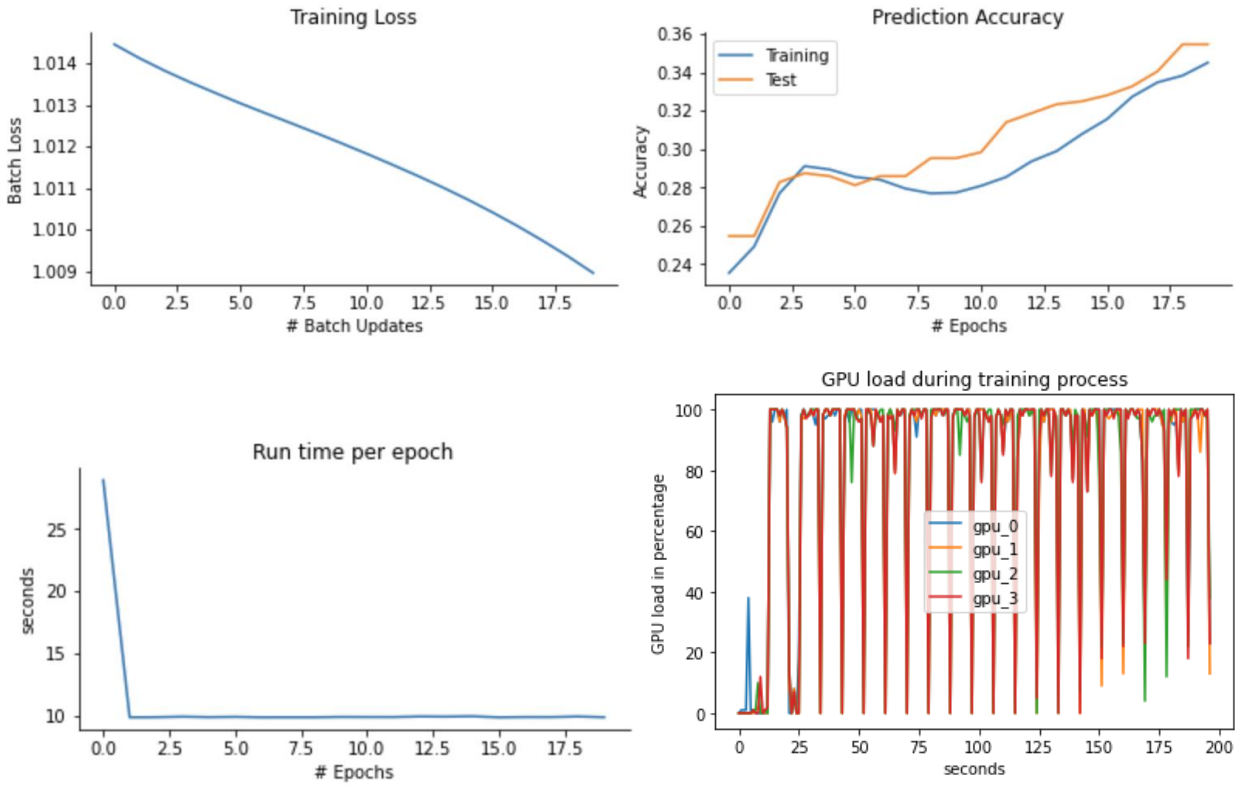
Type	Values
MLP layers	[150528, 2048, 1024, 512, 5]
Learning rate	0.01
Number of epochs	20
Batch size	32
Input image dimension	224x224x3
Number of classes	5
Training image count	2936
Validation image count	734

The following diagrams show the runtime comparison of the algorithm over different accelerators.



Average runtime over different accelerators for each epoch

It is interesting to observe that the training on a single GPU yields the best runtime performance in this scenario. It is almost 12 times faster on a single GPU than a CPU.



The training and runtime of each epoch on 4 GPUs

As expected the more layers and neural are added to the MLP layers, the more costly computation is triggered on the GPU devices. We can observe that during the first epoch (0-28 seconds), the data preprocess happening on the host and training steps on GPUs are carried out simultaneously. Of course, as it is feeding the raw pixels into the MLP layers, we could not expect much on the learning of the model. It is preferable to use a convolutional neural network for better results.

Summary

In conclusion, we observed that using the TF Data library with parallel processing and caching facilitates faster training while keeping the GPU devices busy. We have discussed the usage of cloud storage from the big vendors such as Google and AWS to store our data remotely while ensuring that fetching the data from the cloud storage should be fast. TF Data allows us to read the data remotely not only from cloud storage but also BigQuery, BigTable, etc. Please refer to the documentation for detailed usage.

We also demonstrated the useful function of JAX such as vmap and pmap regarding multi-device processing. JAX is easy to use if we are familiar with NumPy as most of the NumPy APIs are mirrored in JAX. Moreover, Autograd automatically allows us to find the differentiation on native Python and NumPy functions. If we develop our program in a manner that is compatible with pmap usage, we could run this program anywhere regardless of CPUs or multi GPU devices as JAX will take care of the background process for us.

In my opinion, JAX is very flexible to use and easy to code for quickly coming up with the proof of concept. However, if we are aiming to develop the machine learning algorithms for production with multiple platforms, I would say, JAX is not mature yet. It is better to look over other frameworks such as Tensorflow or PyTorch which have a strong eco-system with widely adopted.

Moreover, to keep things simple, we adopted a simple MLP algorithm to demonstrate the scalable infrastructure with JAX. We could use complex and advanced models of JAX to solve the ML problems. It is just a matter of swapping the code and everything discussed in the blog is pretty much the same. I would encourage to use the JAX framework for deep learning such as Flax and Haiku which have implemented most of the building blocks of ML algorithms we need today. Please don't forget to check the JAX official Github repository. Have a nice hands-on with JAX and hopping you enjoy reading this blog.

To keep the blog simple, I have omitted the coding and put everything in a notebook. Please check it out.