SMARTIA

Performance comparison between MXNet and TensorFlow

Smartia Ltd, Bristol and Bath Science Park, Bristol, BS16 7FR, UK

29 Jul 2019

Contents

1 Introduction....2
1.1 Manufacturing the Future 2
1.1.1 Out on the Edge....2
1.2 The Frameworks....3
1.2.1 TensorFlow 3
1.2.2 MXNet....3
2 Datasets....4
3 Methodology 8
4 Experiments....9
4.1 Setting up the test environment....9
4.2 Ways to run benchmark scripts 11
5 Results 13
6 Conclusion 17
6.1 Training Speed 17
6.2 GPU utilisation ratio....17
6.3 GPU memory usage 17
References....18

In confidence. © Smartia Ltd 2019

1 Introduction

These days everything from fridges to running shoes are being fitted with sensors and have internet connectivity. Torrents of information from all these 'sensorised' devices flow in a massive and growing internet of things (IoT). Having access to data from these devices is all well and good (it's cool to see how fast and how far you run every day for example) but if we add machine learning (ML) and artificial intelligence (AI) to the mix we can get real insights into the data and transform how we do things.

In industrial settings, the potential for ML and AI is massive. Indeed, the term *industrial* internet of things (IIoT) has been coined to cover this emerging field.

1.1 Manufacturing the Future

Giving machines 'intelligence' by leveraging sensor data and using machine learning, promises to improve processes and operations as well as providing brand new revenue streams. The best thing is that all the promise and potential applies whether you are a multinational giant or a one-person outfit operating out of a small industrial unit.

The hard part is to realise this potential. This is where having the right tools is critical.

1.1.1 Out on the Edge

To bring machine learning (ML) to industrial settings we could simply run our inference as normal, on conventional computing systems that sit in perfectly controlled server rooms or similar. However, what is more likely is that machine learning models will have to perform their inference in close proximity to their target machines; in other words, they will need to operate at the 'edge'.

Edge processing is a hot topic in IIoT right now. Running ML models combined with space and power restrictions as well as harsh environments will mean those edge devices will need to have good processing power in a compact and hardy form factor. The software used will therefore have to cope with these restrictions.

In this article, we are going to take a look at two popular ML frameworks. The aim is to see how their behaviour and performance can influence their use in industrial settings at the edge.

In confidence. © Smartia Ltd 2019

1.2 The Frameworks

Currently, there are a myriad of frameworks allowing us to develop ML models that make industrial applications smarter and far more intelligent. Though these frameworks are designed to be general ML platforms, the inherent differences of their designs, architectures, and implementations lead to a potential variance of ML performance on Graphic Processing Units (GPUs).

1.2.1 TensorFlow

TensorFlow is an end-to-end open-source deep learning framework. It has a comprehensive, flexible ecosystem of tools, libraries and community resources. There are three main features of this framework: easy model building, robust ML production anywhere and powerful experimentation for research. Moreover, it has been adopted by several tech giants such as Google, Intel, Twitter, and Coca Cola. Community engagement is quite high as almost everyone in the machine learning (ML) community is aware of it. To date, there are 61,000 repositories, 870,000 commits and 5,000 wiki pages related to TensorFlow while there are only 2,000 repositories, 41,000 commits, and 436 wiki pages for MXNet (discussed below). However, TensorFlow can hog a lot of GPU as it tries to allocate all available GPU memory for itself.

1.2.2 MXNet

MXNet is known to be a flexible and efficient library for deep learning. It is developed with many different programming languages thus it can support a wide variety of languages such as C++, Python, Matlab, and R. MXNet is lightweight because its source code is almost 40 times lighter than the source code in TensorFlow. Moreover, it is also optimised for GPU memory usage because it can perform dynamic allocation of memory based on actual requirements. Based on these advantages, we decided to take a deeper look at its difference in performance with TensorFlow.

In confidence. © Smartia Ltd 2019

2 Datasets

Fashion-MNIST is a good starting dataset for us to do benchmarks on, specifically on the performance of ML frameworks using a Central Processing Unit (CPU) or a GPU. A general overview of this dataset can be found in Figure 1. It consists of a training dataset of 60,000 examples and 10,000 examples in its test set. Each example is a 28*28 grayscale image, associated with a label from 10 classes. This dataset will be used to get performance results with a CPU or GPU. Both MXNet and TensorFlow have included Fashion-MNIST as a built-in dataset.

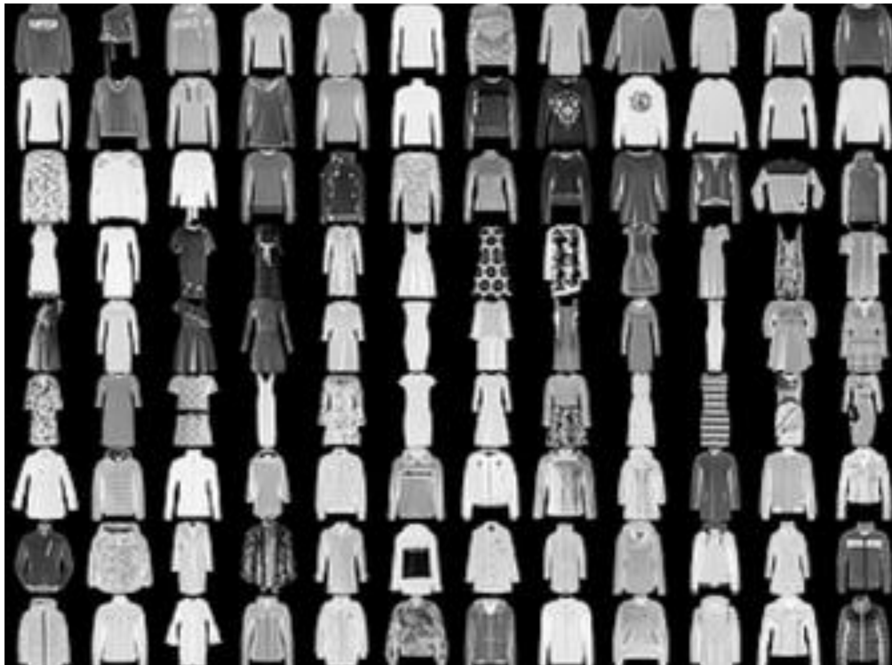

Figure 1: Image examples from Fashion-MNIST dataset, image is taken from fashion-mnist.

After using the Fashion-MNIST dataset, we will use three datasets benchmarked by keras-apache-mxnet[2] to run experiments on a GPU. keras-apache-mxnet can help us use Keras, which is high-level API running on top of other frameworks, with MXNet support as well as TensorFlow. Thus, we can easily use MXNet or TensorFlow as the backend framework using similar benchmark scripts with this type of Keras.

CIFAR-10 dataset includes 50,000 training images and 10,000 test images in 10 classes. Each example is a 32*32 colour image. Also, some example images are shown in Figure 2. It is commonly used to train machine learning and computer vision algorithms. Keras also includes CIFAR-10 as its built-in dataset.

In confidence. © Smartia Ltd 2019

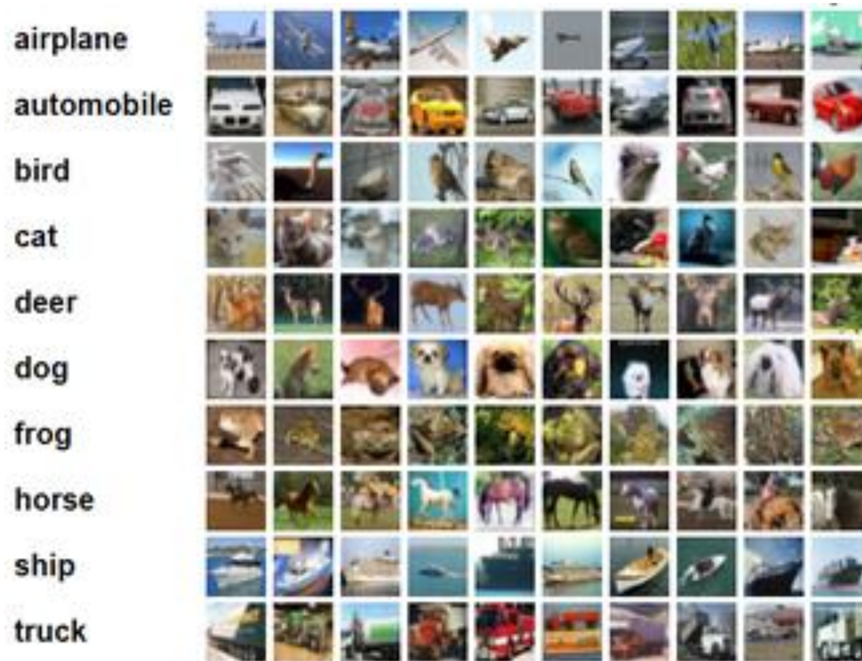

Figure 2: CIFAR-10 image examples, image is taken from CIFAR-10 dataset.

ImageNet is an image dataset that includes more than 14 million images with annotations. In this experiment, we downloaded the ILSVRC2012 version which has about 1.3 million images. In Figure 3, you can get a sense of example images in one synset in ImageNet. This dataset can be downloaded using a downloader script provided by TensorFlow. You can use the following two steps to download ILSVRC2012 ImageNet dataset.

1. Create an account with image-net.org and generate a username and access_key
2. Use download_imagenet.sh from TensorFlow to download the raw images for train and validation.

In confidence. © Smartia Ltd 2019

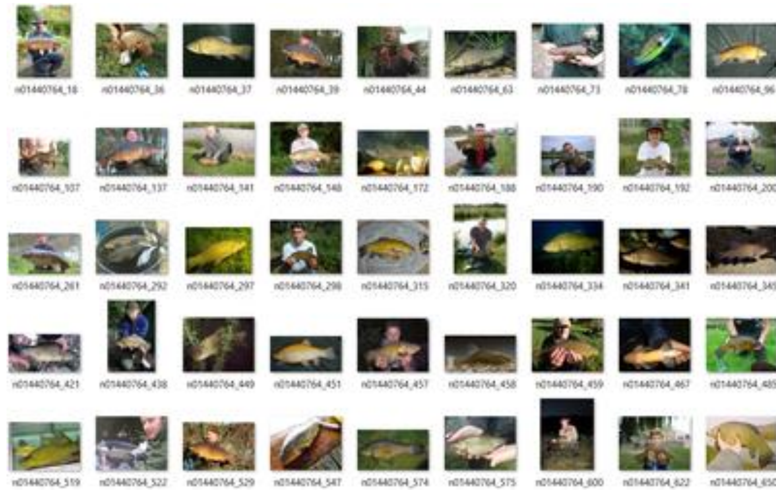

Figure 3: A screenshot of ImageNet original images from one synset: "tench, Tinca tinca".

Besides these real datasets, synthetic data is also used. This data is defined in keras-apache-mxnet and it has 1,000 (256*256) samples in 1,000 classes. Each value is formed by random numbers ranging from 0 to 255.

Table 1 summarises the information in the three datasets as well as a list of some hyper-parameters for training. These hyper-parameters are defined by benchmark scripts in keras-apache-mxnet. In terms of batch size, the default value for all datasets is 32. However, we also use 16 as it is valuable while we train synthetic and CIFAR-10. For ImageNet, we only use 8 as a batch size because it is quite a large dataset.

In confidence. © Smartia Ltd 2019

	Samples	Channels	Resolution	Classes	Batch Size	Optimiser	Learning Rate
mnist_fashion	60000	3	28*28	10	32	Adam	0.001
Synthetic Data (Random Data)	1000	3	256*256	1000	16, 32	RMSprop	0.0001
CIFAR-10	60000	3	32*32	10	16, 32	Adam	0.001
ImageNet (ILSVRC2012)	1, 281, 167	3	High Resolution varied from images to images	1000	8	Adam	0.001

Table 1: Datasets information: mnist_fashion, synthetic data, CIFAR-10, and ImageNet dataset. We also list the default hyperparameters such as batch_size, optimiser and learning rate while using these datasets for training.

In confidence. © Smartia Ltd 2019

3 Methodology

To test performances of MXNet and TensorFlow to train the models, four mentioned datasets will be used to benchmark these two ML frameworks. To compare training speeds with a CPU, fully connected neural networks (NNs), as well as convolutional neural networks (CNNs), are implemented.

We use different architectures to build the model for different datasets, below is the code for a fully connected neural network (NN) architecture:

```
model=keras.Sequential([
  keras.layers.Flatten(input_shape=(28,28)),
    keras.layers.Dense(128,activation=tf.nn.relu),
    keras.layers.Dense(10,activation=tf.nn.softmax)
])
```

The first “Flatten” layer transforms the format of the images from a 2d-array (of 28 by 28 pixels) to a 1d-array of 28*28 = 784 pixels. This layer does not learn as it only reformats the data. After this operation, two layers are added. The first layer has 128 nodes and the second is a 10-node softmax layer which returns an array of 10 probability scores that sum to 1.

The architecture of a CNN will include 2 Conv2D layers. The first layer will have 64 nodes and the second has 32 nodes. With kernel_size equals to 3, we will have a 3*3 filter matrix. Between Conv2D layers and the dense layer, there is also a 'Flatten' layer. In the final output layer, it is a 10-node softmax layer.

```
model = Sequential()
model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

As keras-apache-mxnet has provided benchmark results for three datasets, it is convenient to do experiments based on their scripts. In addition, we customised the batch size parameter to further test performances between the two ML frameworks. ResNet56 networks are used to build the training models as it is a built-in training network which can be easily accessed from Keras.

In confidence. © Smartia Ltd 2019

4 Experiments

In this section, our goal is to monitor the performance of the training process using MXNet and TensorFlow while running training models with the CPU and GPU. Setting up the environment for the CPU version is quite straightforward. We just need to install the CPU versions of MXNet and TensorFlow. Examples of NN can be found in TensorFlow NN tutorial, and CNN models can be built following the Keras CNN tutorial[1]. However, we replace the MNIST dataset with Fashion-MNIST in the Keras CNN model.

To run models with the GPU, we need to install more packages such as CUDA and cuDNN along with the GPU versions of MXNet and TensorFlow. CUDA can boost up NVIDIA GPUs to solve complex computational problems in a more efficient way by leveraging the parallel compute engine. cuDNN is a GPU-accelerated library, specifically for deep neural networks. The following two sections will introduce the installation procedure.

The first part (4.1) will be a guideline about how to set up our test environment using the required packages. The second part (4.2) includes specific commands for running the benchmark scripts. After setting up the environment, the average training speed for each epoch is monitored. Our metric of measuring performances is "img/sec"(number of images processed per second).

4.1 Setting up the test environment

This whole article is aimed at general readers rather than the hardcore researchers. With that in mind, we use a DELL G3 15 gaming laptop, which has a GTX 1060 graphics card, to perform the experiments. The specifications of this computer are shown in Table 2.

Name	**Detail**
CPU	Intel Core i7-8750H 2.20GHz
MEMORY	16.0 GB
Disk	WDC 1TB / SSD 256 GB
GPU	NVIDIA GeForce GTX 1060
OS	Microsoft Windows 10 Home
Shell	Windows PowerShell

Table 2: Computer configuration

In confidence. © Smartia Ltd 2019

Here are the six steps to set up the test environment based on the above specifications.

1. Install python and pip in Windows. Easy Guide Link.
2. Open Windows PowerShell and install pipenv

    pip install pipenv

3. Create a new folder for the experiments in PowerShell

    mkdir performance_test; cd performance_test

4. Create a new virtual environment under the new folder using "pipenv shell" command

    pipenv shell

5. Install the required packages described in the *Library versions* (see table below).
6. Install CUDA 10.0 and configure cuDNN on Windows.

We use keras-apache-mxnet[2] to install Keras, it is necessary to install it along with MXNet. If we need to test the running models on the CPU, then CPU versions of MXNet and TensorFlow should be installed. Once the GPU versions of these two frameworks are installed, they will supersede the CPU versions when the tests are run. The required packages for the experiments are listed in Table 3.

Framework	Version	Installation
Keras	2.2.4.1	pip install keras-mxnet Note: This package requires installing MXNet first.
MXNet (CPU)	1.4.0	pip install mxnet
MXNet (GPU)	1.4.0	pip install mxnet-cu100
Tensorflow (CPU)	1.13.1	pip install tensorflow
Tensorflow (GPU)	1.13.1	pip install tensorflow-gpu
CUDA	10.0	Download Link
cuDNN	7.5	Download Link

Table 3: Library versions: A list of packages which we use for benchmarking.

In confidence. © Smartia Ltd 2019

4.2 Ways to run benchmark scripts

In this part, we will introduce the steps of running benchmark scripts for three datasets.

Ways to download benchmark scripts:

1. Git clone the Keras-Apache-MXNet repo:

   git clone https://github.com/awslabs/keras-apache-mxnet.git

2. Go to the keras-apache-mxnet-master\benchmark\scripts folder

   cd keras-apache-mxnet-master\benchmark\scripts

Before we start running the scripts, we need to enable the parameter, batch_size. Batch_size is a number which defines how much data is required for each iteration. Using a larger batch size will require more memory while smaller batch sizes require less. This feature is important because the memory is limited thus using a suitable batch size can utilise the memory usage. In this case, we need to enable "batch_size" as an argument in the script. For example, benchmark_resnet needs to be modified with an extra argument "batch_size".

Here is the example code for adding "batch_size" as a new argument in benchmark_resnet.py.

```
parser.add_argument('--batch_size', default=32, type=int, help='Number of batch_size')

# Replace line 78 with the following example code in benchmark_resnet.py
batch_size = args.batch_size * num_gpus if num_gpus > 0 else args.batch_size
```

Run benchmark script for CIFAR-10 (Example code).

```
python benchmark_resnet.py --dataset cifar10 --version 1 --layers 56 --gpus 1 --epoch 20
--batch_size 32
```

Run benchmark script for Synthetic data (Example code). Note: Current Path is the directory of the scripts in the benchmark, e.g. ''C:\Users\junzh\python_tests\mxnet_test\keras-apache-mxnet-master\benchmark\scripts":

```
python run_benchmark.py --pwd=['Current Path'] --mode=gpu_config --
model_name=resnet50 --dry_run=True --inference=False --epochs=20
```

Run imageNet script. Note: data_path is the training data folder of imagenet, e.g. "D:\imageNetData\tensorflow_imagenet\imagenet_data\train"

In confidence. © Smartia Ltd 2019

python benchmark_resnet.py --dataset imagenet --version 1 --layers 56 --gpus 1 --epoch 20 --train_mode fit_generator --data_path ['ImageNet train data path'] --batch_size 8

To switch the backend from MXNet to TensorFlow or conversely, you can update the *$HOME/.keras/keras.json* to set the backend and image_data_format:

For TensorFlow backend benchmarks, set backend: TensorFlow and image_data_format: channels_last.

For MXNet backend benchmarks, set backend: mxnet and image_data_format: channels_first.

It is strongly suggested that we set channel_first for MXNet because using channel_last with MXNet will greatly slow the training speed down in GPU mode. This framework prefers channel_first due to performance reasons on the GPU. More information can be found in this documentation.

In confidence. © Smartia Ltd 2019

5 Results

In this section, we present the results of our experiments.

In Table 4, MXNet is shown to be about 30% slower than TensorFlow if we build the training model using one neural network layer with 128 nodes. The training speed dramatically decreases if we use a CNN to build the model and run it again. However, there is no difference between TensorFlow and MXNet if we use a GPU to train the model using CNN. In this instance both need around 10 seconds to train one epoch of the mnist_fashion dataset. One possible reason for this is that mnist_fashion is a small dataset and the model is rather simple as it only has 2 hidden layers.

Computer Model	GPUs*	Computing Model	Batch Size	Keras-MXNet (img/sec)	Keras-Tensorflow (img/sec)	Speed Ratio (Speed of MXNet / Speed of Tensorflow)
DELL G3 15	0	One neural network layer with 128 nodes	32	3571	5000	71%
DELL G3 15	0	Two Conv2D layers (64 nodes and 32 nodes)	32	300	344	87%
DELL G3 15	1	Two Conv2D layers (64 nodes and 32 nodes)	32	4545	4545	100%

Table 4: Results of performance when running fully connected neural networks and CNN on a CPU or a GPU using the mnist_fashion dataset. * If GPU is 0, it means it uses a CPU (Intel Core i7-8750) to generate the benchmarks.

In confidence. © Smartia Ltd 2019

	Computer Model	GPUs	Batch Size	Speed Ratio (Speed of MXNet / Speed of Tensorflow)	Keras-MXNet Speed (img/sec)	GPU utilisation ratio while running MXNet	Keras-Tensorflow Speed (img/sec)	GPU utilisation ratio while running Tensorflow
Synthetic Data	DELL G3 15	1	16	100%	14	2%	14	62%
Synthetic Data	DELL G3 15	1	32	82%	41	2%	50	40%
CIFAR-10	DELL G3 15	1	16	89%	229	1%	257	38%
CIFAR-10	DELL G3 15	1	32	78%	357	2%	454	44%
ImageNet	DELL G3 15	1	8	10%	3.2	1%	32	20%**
ImageNet	DELL G3 15	1	16	--	Out of Memory	--	Out of Memory	--

Table 5: Training speed results and GPU utilisation ratio for three datasets based on different batch sizes using ResNet-50 network.

*** For the training of ImageNet, fit_generator is used to run the model. To use this, we set steps_per_epoch to 10000. It is estimated that 110 hours are needed for one epoch if we use MXNet while about 10 hours can be used to finish one epoch using TensorFlow.**

**** This value fluctuates from 0 - 30% during training.**

In confidence. © Smartia Ltd 2019

In Table 5, we can see that the smaller batch sizes result in lower speed. For synthetic data, the speed decreases from 41 img/sec to 14 img/sec if the batch size changes from 32 to 16. In terms of ImageNet, the model can't be run if the batch size is 16 or above.

MXNet is generally slower than TensorFlow by around 20% when training synthetic data and CIFAR-10 using a 32 batch size. Moreover, if the dataset is too large like ImageNet, the training speed using MXNet is ten times slower than TensorFlow.

There is a huge difference in the GPU utilisation ratio between the two frameworks. 2% is the common case for MXNet while the ratio will go from 40% to 60% if we use synthetic data or CIFAR-10. Training ImageNet, the ratio for MXNet is about 1% while this ratio fluctuates from 0 to 30%.

In confidence. © Smartia Ltd 2019

For clarity, Figure 4 is a graph containing information from the above table:

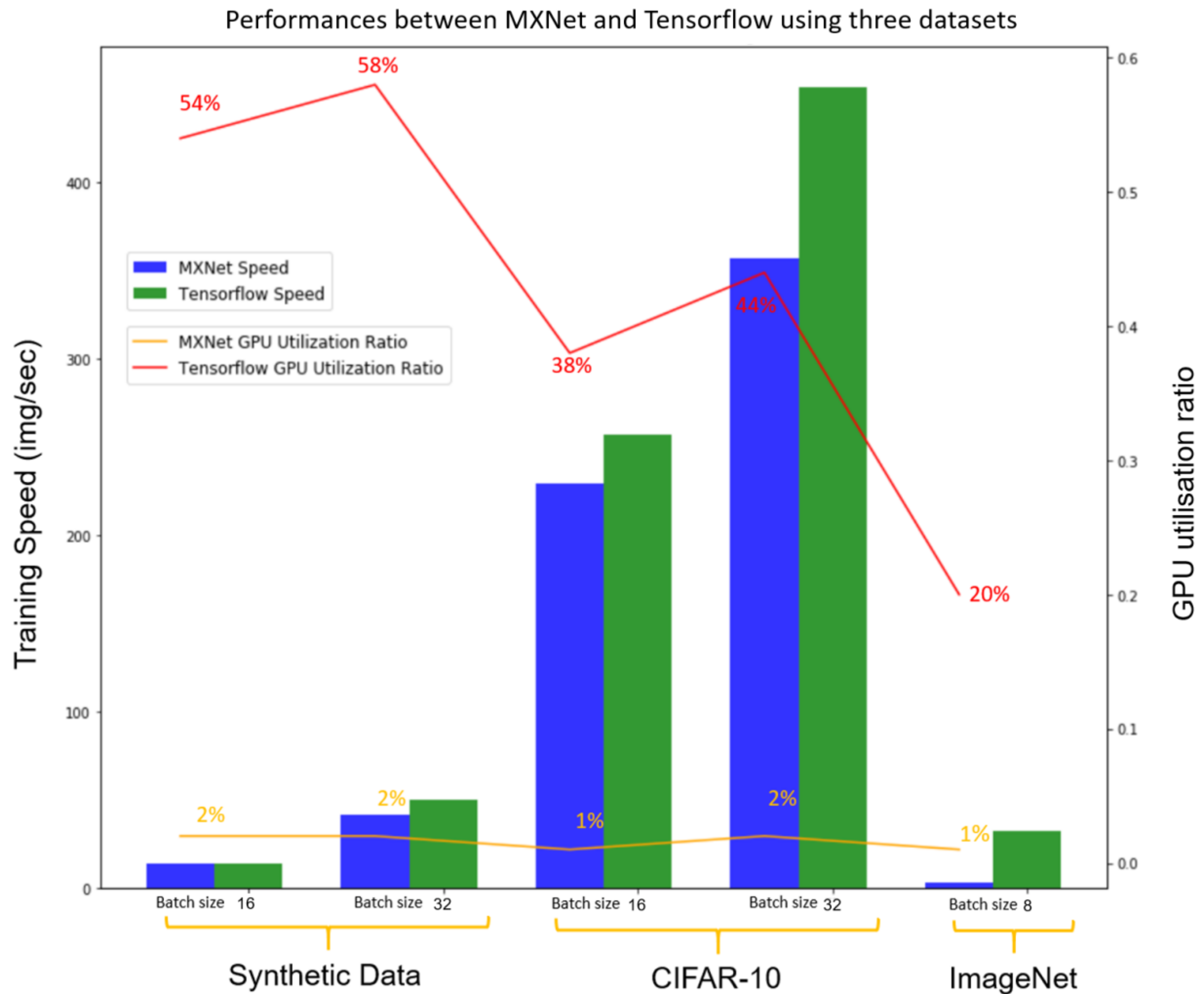

Figure 4: Differences between MXNet and TensorFlow in training speed and GPU utilisation ratio using three datasets. Training speed from MXNet is generally slower than TensorFlow in all datasets. However, MXNet is much more efficient than TensorFlow as it only occupies 2% of the GPU while TensorFlow's processes use more than 20% of the GPU.

In confidence. © Smartia Ltd 2019

6 Conclusion

6.1 Training Speed

As we have noticed the training speed is slower with MXNet than TensorFlow using current configuration settings. This result is also confirmed in another blog [5]. MXNet is believed to be more efficient and faster according to official benchmark results and a blog posted in Medium.com. However, in the scripts in which MXNet has a higher training speed, they used multiple GPUs to boost the training process. This difference raises a question: Is MXNet more efficient when there are multiple devices such as GPUs? Further experiments need to be designed to test MXNet on a multiple CPU or GPU scenario.

6.2 GPU utilisation ratio

On the other hand, one of the striking differences was the GPU utilisation ratio. Whereas MXNet only occupied 2% on the GPU, TensorFlow occupied close to 40% - 60%. Does this behaviour mean that we could have more tasks on the GPU while training the model using MXNet? This blog [3] takes a look at the differences in memory usage between the two ML frameworks, it found that TensorFlow allocates as much memory as possible, even though its memory footprint is similar to MXNet without lowering the training speed. In contrast, the experiment proves that MXNet aggressively reduces memory footprint and reuses memory space whenever possible.

6.3 GPU memory usage

We also have a look at memory usage based on the information in this blog [4]. We can set the fraction of GPU memory to be allocated for TensorFlow by using the following example commands:

gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.333) sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))

Although we decrease the fraction of gpu_memory_fraction from 0.99 to 0.05, the training speeds remain the same for both when we train using CIFAR-10 with a batch size of 32.

In confidence. © Smartia Ltd 2019

References

1. Keras CNN tutorial
2. Keras Apache MXNet
3. TensorFlow vs MXNet
4. Memory Usage Between TensorFlow and MXNet
5. Comprehensive evaluation between TensorFlow, PyTorch, and MXNet

In confidence. © Smartia Ltd 2019