Cluster Computing (2021) 24:2017–2038
https://doi.org/10.1007/s10586-021-03240-4

DLBench: a comprehensive experimental evaluation of deep learning frameworks

Radwa Elshawi[1] · Abdul Wahab[1] · Ahmed Barnawi[2] · Sherif Sakr[1]

Received: 17 January 2020 / Revised: 7 January 2021 / Accepted: 13 January 2021 / Published online: 7 February 2021

Abstract
Deep Learning (DL) has achieved remarkable progress over the last decade on various tasks such as image recognition, speech recognition, and natural language processing. In general, three main crucial aspects fueled this progress: the increasing availability of large amount of digitized data, the increasing availability of affordable parallel and powerful computing resources (e.g., GPU) and the growing number of open source deep learning frameworks that facilitate and ease the development process of deep learning architectures. In practice, the increasing popularity of deep learning frameworks calls for benchmarking studies that can effectively evaluate and understand the performance characteristics of these systems. In this paper, we conduct an extensive experimental evaluation and analysis of six popular deep learning frameworks, namely, *TensorFlow*, *MXNet*, *PyTorch*, *Theano*, *Chainer*, and *Keras*, using three types of DL architectures Convolutional Neural Networks (CNN), Faster Region-based Convolutional Neural Networks (Faster R-CNN), and Long Short Term Memory (LSTM). Our experimental evaluation considers different aspects for its comparison including accuracy, training time, convergence and resource consumption patterns. Our experiments have been conducted on both CPU and GPU environments using different datasets. We report and analyze the performance characteristics of the studied frameworks. In addition, we report a set of insights and important lessons that we have learned from conducting our experiments.

Keywords Deep learning · Experimental evaluation · CNN · LSTM

1 Introduction

Nowadays, we are witnessing an explosion of interest in Artificial Intelligence (AI)-based systems across governments, industries and research communities with a yearly spending figure of around 12.5 billion US dollars [47]. A central driver for this explosion is the advent and increasing popularity of Deep Learning (DL) techniques that are capable of learning task-specific representation of the input data, automating what used to be the most tedious development task which is that of feature engineering. In general, deep learning techniques represent a subset of artificial intelligence methodologies that are based on artificial neural networks (ANN) which are mainly inspired by the neuron structure of the human brain [6]. It is described as *deep* because it has more than one layer of nonlinear feature transformation. In practice, the main advantage of deep learning over the traditional machine learning techniques is their ability for automatic feature extraction which allows learning complex functions to be mapped from the input space to the output space without much human intervention. In particular, it consists of multiple layers, nodes, weights and optimization algorithms. Due to the increasing availability of labeled data, computing power, better optimization algorithms, and better neural net models and architectures, deep learning

✉ Sherif Sakr
sherif.sakr@ut.ee

Radwa Elshawi
radwa.elshawi@ut.ee

Abdul Wahab
abdul.wahab@ut.ee

Ahmed Barnawi
ambarnawi@kau.edu.sa

[1] University of Tartu, Tartu, Estonia

[2] King Abdulaziz University, Jeddah, Saudi Arabia

techniques have started to outperform humans in some domains such as image recognition and classification. Therefore, deep learning applications on big data are gaining increasing popularity in various domains including natural language processing, medical diagnosis, speech recognition and computer vision [2, 11, 19, 23, 26]. Recently, we have been witnessing an increasing growth in the number of open source deep learning frameworks. Examples include *TensorFlow* [1], *MXNet* [8], *Chainer* [45], *Torch* [12], *PyTorch* [34], *Theano* [5], *CNTK* [39], *Caffe* [22], and *Keras* [9] (Fig. 1). In practice, different frameworks focus on different aspects and use different techniques to facilitate, parallelize and optimize the training and deployment of the deep learning models.

Convolutional Neural Networks (CNNs) is a popular deep learning technique that has shown significant performance gains in several domains such as object detection, disease diagnosis and autonomous driving [20, 41], however such networks are computationally expensive due to the model complexity and the huge amount of hyperparameters used that need to be trained over large datasets. When the size of the dataset is relatively small, most of the classification algorithms such as decision trees, random forests and logistic regression have been shown to achieve comparable performance. However when the size of the data is huge, `AlexNet` [26] shows that training CNNs on millions of images from ImageNet outperforms all the previous work done in image classification and hence concluded that using large size training dataset improves the performance of classification tasks.

Faster Region-based convolutional Neural Networks (Fatser R-CNN) is considered the state-of-the-art object detection algorithm which was introduced by Ren et al. [37]. Faster R-CNNs are the main driven behind advances in object detection [48]. Fatser R-CNN consists of two main modules which are Regional Proposal Network (RPN) and Fast R-CNN detector. The RPN is a fully convolutional network for generating object proposals that will be fed into the second module. The second module is the Fast R-CNN detector whose purpose is to refine the proposals. The key idea is to share the same convolutional layers for the RPN and Fast R-CNN detector up to their own fully connected layers. Thus, the image only passes through the CNN once to produce and then refine object proposals.

Fig. 1 Timeline of deep learning frameworks

Long Short Term Memory (LSTM) is another popular deep learning technique which represents a special type of Recurrent Neural Network (RNN) that is capable of learning long-term dependencies [38]. An LSTM cell contains a memory that enables the storage of previous sequences. Each cell has three types of gates to control and protect the state of the cells: input gate, output gate and forget gate. The forget gate is to decide what information to discard from each LSTM cell. The input gate decides the update of the memory state based on the input values and the output gate decides what to output based on the input and the memory of the LSTM cell.

Graphics Processing Units (GPUs) were originally designed for rendering graphics in real time. However, recently, GPUs have been increasingly used for general purpose computation that requires a high data parallel architecture, such as deep learning computation. In principle, each GPU is composed of thousands of cores. Thus, GPUs can process a large number of data points in parallel which leads to higher computational throughput. Training deep learning models is computationally expensive and time consuming process due to the need for a tremendous volume of data, and leveraging scalable computation resources can speed up the training process significantly [15, 24]. Recently, research effort has been focused on speeding up the training process. One popular way for speeding up this process is to use specialized graphical processors such as GPU and Tensor Processing Unit (TPU). As per Amdahl's law, in a particular computation task, the non-parallelizable portion may limit the computation speedup [18]. For example, if the non-parallelizable portion of a task is equal to 50% then reducing the computation time to almost zero will result only in the increase of speed by a factor of two. Hence to speed up the training time, the non-parallelizable computation portions should be seriously addressed.

In general, choosing a DL framework for a particular task is a challenging problem for domain experts. We argue that benchmarking DL frameworks should consider performance comparison from three main dimensions: (1) how computational environment (CPU, GPU) may impact the performance; (2) how different types and variety of datasets may impact on performance; and (3) how different deep learning architectures may impact the performance. Most of current benchmarking efforts for DL frameworks have been focused mainly on studying the effect of different CPU-GPU configurations on the performance of

different deep learning frameworks on standard datasets [3, 10, 40, 42]. Very few of existing efforts, to the best of our knowledge, have been devoted to study the effectiveness of default configurations recommended by each DL framework with respect to different datasets and different DL architectures. We argue that effective benchmarking of DL frameworks requires an in-depth understanding of all of the above three dimensions.

In this paper, we present design considerations, metrics and insights towards benchmarking DL software frameworks through a comparative study of six popular deep learning frameworks. This work is an extension of our initial work [29] that mainly focused on comparing the performance of DL frameworks for CNN architectures. In particular, in this work, we follow a holistic approach to design and conduct a comparative study of six DL frameworks, namely *TensorFlow*, *MXNet*, *PyTorch*, *Theano*, *Chainer*, and *Keras*, focusing on comparing their performance in terms of training time, accuracy, convergence, CPU and memory usages on both CPU and GPU environments. In addition, we study the impact of different deep learning architectures (CNN, Faster R-CNN, and LSTM) on both the performance and system resource consumption of DL frameworks using different datasets. In particular, for evaluating the performance of CNN architecture, we use four datasets, namely, `MNIST`, `CIFAR-10`, `CIFAR-100` [25] and `SVHN` [33]. For evaluating the performance of Faster R-CNN architecture, we use `VOC2012` [14]. For evaluating the performance of LSTM architecture, we use three datasets, namely, `IMDB Reviews` [28], `Penn Treebank` [30], and `Many things: English to Spanish`[1]. For ensuring repeatability as one of the main targets of this work, we provide access to the source codes and the detailed results for the experiments of our study[2].

The remainder of this paper is organized as follows. We discuss the related work in Sect. 2. Section 3 provides an overview of the different deep learning frameworks that have been considered in this study. Section 4 describes the details of our experimental setup in terms of used datasets, hardware configurations and software configurations. Section 5 provides the detailed results of our experiments and lessons learned before we conclude the paper in Sect. 6.

2 Related work

Some research efforts have attempted to tackle the challenge of benchmarking deep learning frameworks and comparing different neural network hardware and libraries [53]. For example, the `DeepBench` project[3] focuses on the benchmarking fundamental neural networks operations such as dense matrix multiplications, convolutions and communication on different hardware platforms using different neural network libraries. `DAWNBench` [10] is a benchmark that focuses on end-to-end training time of deep learning model to achieve certain accuracy on difference deep learning platforms including `TensorFlow` and `PyTorch` using image classification datasets including *CIFAR10*, *ImageNet* and question answering on SQuAD [36], showing differences across models, software and hardware. Awan et al. [3] compare between CPU and GPU for multi-node training using OSU-Caffe [44] and Intel-Caffe [21]. Authors provide the following key insights: (1) Convolutions account for the majority of time consumed in DNN training, (2) GPU-based training continues to deliver excellent performance across generations of GPU hardware and software, and (3) Recent CPU-based optimizations like MKL-DNN [31] and OpenMP-based thread parallelism leads to significant speed-ups over under-optimized designs. Shams et al. [40] analyze the performance of three different frameworks, Caffe, TensorFlow, and Apache SINGA, over several hardware environments. More specifically, authors provide analysis of the frameworks' performance over different hardware environments in terms of speed and scaling. Wang and Guo [49] compare the accuracy of the same CNN model on three different frameworks, TensorFlow, Caffe, and PyTorch. Results show that the PyTorch based models tend to obtain the best performance among these three frameworks because of its better weight initialization methods and data preprocessing steps, followed by TensorFlow and then Caffe. In conclusion, using the same CPU-GPU configurations, no single DL framework outperforms others on all performance metrics on the different datasets.

Bahrampour et al. [4] evaluated the training and interface performance of different deep learning frameworks including `Caffe`, `Neon`, `TensorFlow`, `Theano`, and `Torch` on a single CPU/GPU environment using *MNIST* and *ImageNet* datasets. Wu et al. [51] evaluated the performance of four deep learning frameworks including `Caffe`, `Torch`, `TensorFlow` and `Theano` on different selection of CPU-GPU configurations on three popular datasets: *MNIST*, *CIFAR-10*, and *ImageNet*. In addition, authors conducted comparative measurement study on the resource consumption patterns on the four frameworks and their performance and accuracy implications, including CPU and memory consumption, and their correlations to varying settings of hyper-parameters under different configuration combinations of hardware, parallel computing libraries. Zou et al. [55] evaluated the performance of four

[1] https://www.manythings.org/bilingual/.

[2] https://github.com/DataSystemsGroupUT/DLBench.

[3] https://github.com/baidu-research/DeepBench.

deep learning frameworks including Caffe, MXNet, TensorFlow and Troch on the *ILSVRC-2012* dataset which is a subset of the ImageNet dataset, however, this study lacks the empirical evaluation for the frameworks used. Liu et al. [27] evaluated five deep learning frameworks including Caffe2[4], Chainer, Microsoft Cognitive Toolkit (CNTK), MXNet, and Tensorflow across multiple GPUs and multiple nodes on two datasets, *CIFAR-10* and *ImageNet*. Shi et al. [42] benchmarked several deep learning frameworks including TensorFlow, Caffe, CNTK, and Torch on CPU and GPU with the main focus on the running time performance with three different types of neural networks including feed-forward neural networks (FCNs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Thus, to the best of our knowledge, our study is the first study to benchmark six popular deep learning frameworks (*TensorFlow*, *MXNet*, *PyTorch*, *Theano*, *Chainer* and *Keras*) from different performance aspects including accuracy, modeling time and resource consumption on both of CPU- and GPU-based environments. Some recent efforts [40, 54] provide end-to-end DL benchmarking by considering only the training phase or a particular DL task, however, no study considers a holistic approach to study the impact of hardware configurations, and default hyper-parameters on the performance of DL frameworks on different deep learning architectures with respect to both accuracy, training time, and resource consumption.

3 Reference deep learning frameworks

As deep learning techniques have been gaining increasing popularity, a lot of academic and industrial organizations (e.g., Berkeley Vision and Learning Center, Facebook AI Research, Google Brain) have focused on developing frameworks to enable the experimentation of with deep-neural networks in a user-friendly way. Most of the deep learning frameworks such, as PyTorch, Torch, Caffe, Keras, TensorFlow, Theano, and MXNet adopt a similar software architecture and provide APIs to allow users to easily configure deep neural network models. Most of the current deep learning frameworks are implemented on the top of widely used parallel computing libraries such as OpenBlas [52], cuBLAS [32], NCCL [32] and OpenMP [13]. Most of the deep learning frameworks offer some of the well-known neural networks models such AlexNet [26] and VGG [43] and Resnet [17] as user-configurable options. In this section, we give an overview on the frameworks considered in this study.

3.1 TensorFlow

TensorFlow[5] is an open source library for high-performance computation and large-scale machine learning across different platforms including CPU, GPU and distributed processing [1]. TensorFlow, developed by Google Brain team in Google's AI organization, was released as an open source project in 2015. It provides a data flow model that allows mutable state and cyclic computation graph. TensorFlow supports different types of architectures due to its auto differentiation and parameter sharing capabilities. TensorFlow supports parallelism through the parallel execution of data flow graph model using multiple computational resources that collaborate to update shared parameters. The computation in TensorFlow is modeled as directed acyclic graph where nodes represent operations. Values that flow along the edges of the graph are called *Tensors* that are represented as a multi-dimensional array. An operation can take zero or more tensors as input and produce zero or more tensors as output. An operation is valid as long the graph which the operation is part of is valid.

3.2 MXNet

MXNet[6] is an open source deep learning framework founded as a collaboration between Carnegie Mellon University, Washington University and Microsoft. It is a scalable framework that allows training deep neural networks using different programming languages including C++, Python, MATLAB, JavaScript, R, Julia and, Scala. MXNet supports data-parallelism on multiple CPUs or GPUs and allows model-parallelism as well. MXNet supports two different modes of training; synchronous and asynchronous training [8]. MXNet provides primitive fault tolerance operations through save and load: save stores the model's parameters to a checkpoint file and load restores the model's parameters from a checkpoint file. MXNet supports both declarative programming and imperative programming.

3.3 Theano

Theano[7] is an open source Python library for fast large-scale computations that can run on different computing platforms including CPU and GPU [7]. Theano has been developed by researchers and developers from Montreal University. Theano is a fundamental mathematical expression library that facilitates building deep learning

[4] https://caffe2.ai/.

[5] https://www.tensorflow.org/.

[6] https://mxnet.apache.org/.

[7] https://deeplearning.net/software/theano//.

models. Different libraries have been developed on the top of Theano such as Keras which is tailored for building deep learning models and provides the building blocks for efficient experimentation of deep learning models. Computations in Theano are expressed using Numpy-esque syntax. Theano works by creating a symbolic representation of the operations which are translated to C++ and then compiling them into dynamically loaded Python molecules. Theano supports both data parallelism and model parallelism.

3.4 PyTorch

`PyTorch`[8] has been introduced by Facebook's AI research group in October 2016 [35]. PyTorch is a Python-based deep learning framework which facilitates building deep learning models through an easy to use API. Unlike most of the other popular deep learning frameworks, which use static computation graphs, PyTorch uses dynamic computation, which allows greater flexibility in building complex architectures.

3.5 Chainer

`Chainer`[9] is an open-source deep learning framework, implemented in Python. The development of Chainer is led by researchers and developers from Tokyo University [46]. Chainer provides automatic differentiation APIs for building and training neural networks. Chainer's approach is based on the "define-by-run" technique which enables building the computational graph during training and allows the user to change the graph at each iteration. Chainer is a flexible framework as it provides an imperative API in Python and NumPy. Both CPU and GPU computations are supported by Chainer.

3.6 Keras

`Keras`[10] is an open source neural networks framework developed by François Cholle, a member of the Google AI team. Keras is considered as a meta-framework that interacts with other frameworks. In particular, it can run on the top of TensorFlow and Theano [7]. It is implemented in Python and provides high-level neural networks APIs for developing deep learning models. Instead of handling low-level operations (differentiation and tensor manipulation), Keras relies on a specialized library that serves as its back-end engine. Keras minimizes the number of actions required by a user for a specific action. An important feature of Keras is its ease of use without sacrificing flexibility. Keras enables the users to implement their models as if they were implemented on the base frameworks (such as TensorFlow, Theano, MXNet).

Table 1 summarizes the main features of the frameworks under test in our study.

[8] https://pytorch.org/.

[9] https://chainer.org/.

[10] https://keras.io/.

4 Experimental setup

In this section, we start by introducing the reference models and datasets used for CNN, LSTM, and Faster R-CNN. Next, we describe the hardware and software resources used for conducting the experiments. Finally, we introduce the metrics used to evaluate the performance of the deep learning models.

4.1 Reference models and datasets for CNN

We selected the most popular datasets used in different deep learning tasks. For CNNs, we selected four different datasets: `MNIST`, `CIFAR-10`, `CIFAR-100` and `SVHN`. Figure 2 shows the architecture of the CNN associated with each of `MNIST`, `CIFAR-10`, `CIFAR-100` and `SVHN`. The description of each dataset and the structure of its associated CNN model is detailed as follows.

MNIST The MNIST Dataset contains 70,000 images of handwritten digits. The training set consists of 60,000 examples (86% of the original dataset) while the test set consists of 10,000 examples (14% of the original dataset). The dataset has 10 classes, the 10 numerical digits. The CNN model structure of the MNIST dataset (Fig. 2a) consists of two consecutive `conv2D` layers, having 32 and 64 filters respectively, with `ReLU` activation function. Next, we added a `max-pooling` layer followed by `dropout` layer (keep-prob = 0.75). Then we used a layer to flatten the data. In order to have a densely-connected NN layer, we used a `dense` layer. In order to reduce the overfitting, we used another `dropout` layer (keep-prob = 0.5). Finally, we used a dense layer with 10 outputs to represent labels with a `softmax` activation function. The network is trained for 15 epochs.

CIFAR-10 The dataset consists of 60,000 colour images in 10 classes, with 6000 images per class. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The training set consists of 50,000 examples (83% of the original dataset) while the test set consists of 10,000 examples (17% of the original dataset). The architecture of the chosen CNN model (Fig. 2b) consists of two consecutive `conv2D` layers having 32 filters with `ReLU` activation function followed by a `maxpooling` layer and `dropout` layer (keep-prob = 0.75). In addition, another two consecutive

Table 1 Summary of the main properties of the deep learning frameworks used our study as of 4/12/2019

	TensorFlow	Keras	PyTorch	MXNet	Theano	Chainer
Release date	2016	2015	2017	2015	2010	2015
Core language	C++	Python, R	C++, Python	C++	C++	Python
API	C++, Python	Python	Python	C++, Python, R Scala, Clojure Javascript, Web-UI	Python	Python
Data parallelism	✔	✔	✔	✔	✔	✔
Model parallelism	✔	✔	✔	✔	✔	✔
Programming paradigm	Imperative	Imperative	Imperative	Imperative declarative	Imperative	Imperative
Fault tolerance	Checkpoint-and -recovery	Checkpoint- and -resume	Checkpoint- and -resume	Checkpoint-and -resume	Checkpoint- and -resume	Checkpoint- and -resume
Multi GPU	✔	✔	✔	✔	✔	✔
Popularity (# stars on Github)	138k	45.8k	34.2k	18.1k	9k	5.2k

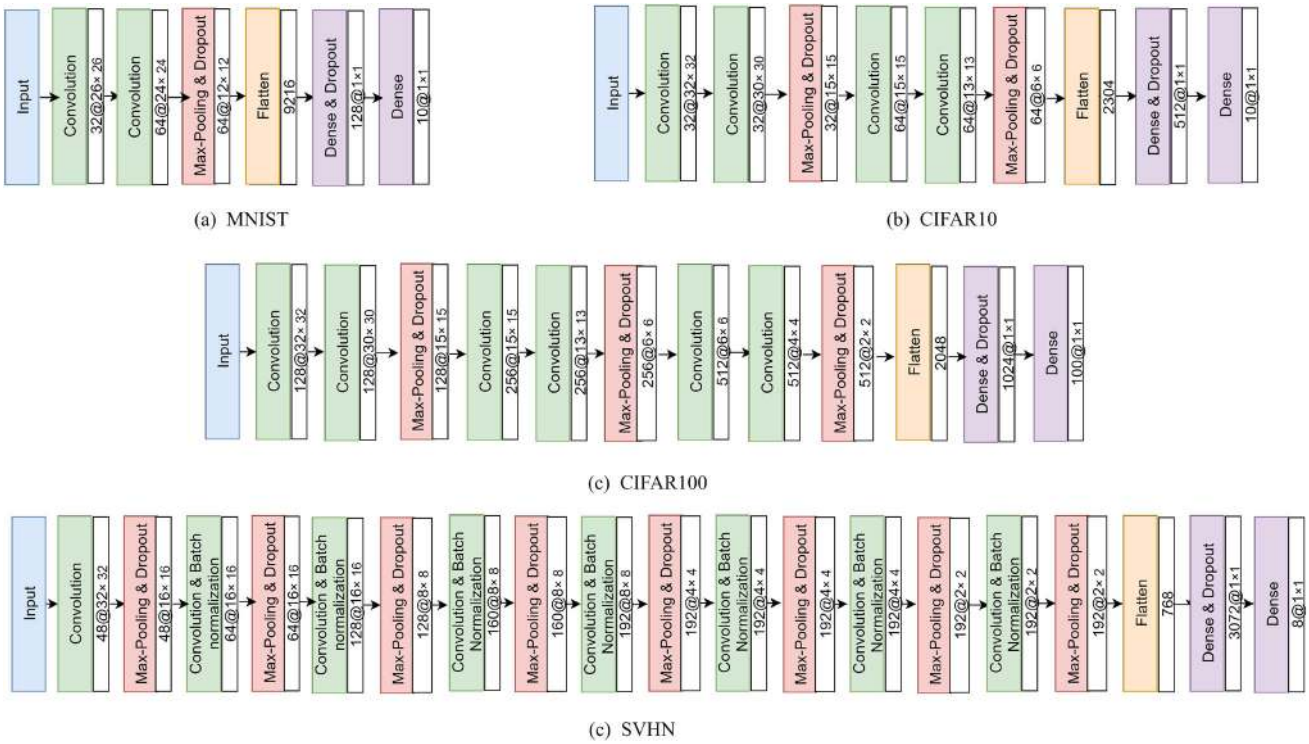

Fig. 2 The architecture of the CNN used with each of `MNIST`, `CIFAR-10`, `CIFAR-100` and `SVHN`

`conv2D` layers having 64 filters with `RELU` activation function were added, followed by a `maxpooling` layer and `dropout` layer (keep-prob = 0.75). A `flatten` layer followed by a `dense` layer having `ReLU` as the activation function was used. The last layers used were a `dropout` (keep-prob = 0.5) layer followed by a `dense` layer having a `softmax` activation function. The network is trained for 100 epochs.

CIFAR-100 This dataset is just like the CIFAR-10, except that it consists of 60,000 colour images in 100 classes, with 600 images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. For example, the classes aquarium fish, flatfish, ray, shark, trout all belong to the superclass fish. The training set consists of 50,000 examples (83% of the original dataset) while the test set consists of 10,000 examples (17% of the original dataset). The architecture of the chosen CNN model (Fig. 2c) consists of two consecutive `conv2D` layers having 128 filters with `ReLU` activation function, followed by a `maxpooling` layer and `dropout` layer

(keep-prob = 0.9). Next, another two consecutive `conv2D` layers having 256 filters with `RELU` activation function were used. Next, a `maxpooling` layer and `dropout` layer (keep-prob = 0.75) was added. Then, we add another two consecutive `conv2D` layers having 512 filters with `RELU` activation function, followed by a `maxpooling` layer and `dropout` layer (keep-prob = 0.5). After that, we used a `flatten` layer followed by a `dense` layer having `ReLU` as the activation function. The last layers used were `dropout` (keep-prob = 0.5) layer followed by a `dense` layer having `softmax` activation function. The network is trained for 200 epochs.

SVHN It is a dataset for Street View House Numbers. The dataset contains 10 classes with a total of 99,289 images in which 73,257 digits (74% of the original dataset) are used for training and 26,032 digits (26% of the original dataset) are used for testing. The architecture of the model for this dataset (Fig. 2d) consists of a `conv2D` layer, having 48 filters with `ReLU` activation function, `max-pooling` layer and `dropout` layer (keep-prob = 0.8). After that, there exists 7 blocks each of which consists of a `conv2D` layer with filters (64,128,160,192,192,192,192) followed by `ReLU` activation and a `maxpooling` layer followed by a `dropout` layer (keep-prob = 0.8). Finally, we used a `dense` layer with `ReLU`, `dropout` (keep-prob = 0.5) and a `dense` layer having a `softmax` activation function. The network is trained for 100 epochs.

4.2 Reference models and datasets for LSTM

Figure 3 shows the architecture of the LSTM model associated with each of `IMDB Reviews`, `Penn Treebank`, and `Many things: English to Spanish`. The description of each dataset and the structure of its associated LSTM model is detailed as follows.

IMDB Reviews This is a dataset for binary sentiment classification. This dataset is intended to serve as a benchmark for sentiment classification. The training dataset contains 25,000 highly polar movie reviews (50% of the original dataset), and the testing dataset contains 25,000 reviews (50% of the original dataset). A review is encoded as a sequence of word indexes (integers) by overall frequency in the dataset (Fig. 3a). All sequences are padded to a length of 500 with a vocabulary size trimmed to 5000. The network architecture used on this dataset consists of an `embedding` of size 32, LSTM with hidden size 100 for processing sentences and a `dense` layer to take in the last hidden state of LSTM and produce a single `sigmoid` activation. It is also important to supply actual lengths of padded sequences into the LSTM, so that the last state is not diminished by the paddings at the end of a sequence. The network is trained for 50 epochs with the `Adam` optimizer, learning rate 10^{-3} and a batch size 64.

Penn Treebank The dataset is large and diverse and contains one million words from Wall Street Journal. The words are annotated in Treebank II style which encodes them to a tree based structure giving their syntactic and semantic relevance. The dataset is commonly used for the language modeling task, where the goal is to learn a probabilistic model for generating text based on previous words. The training dataset contains 1,088,220 examples (92% of the original dataset), and the testing dataset contains 59,118 examples (8% of the original dataset). The architecture used on this dataset (Fig. 3b) consists of an `embedding` of size 128, two `LSTM` layers with `hidden` size of 1024, a `Dropout` layer with rate 0.5 and a Dense

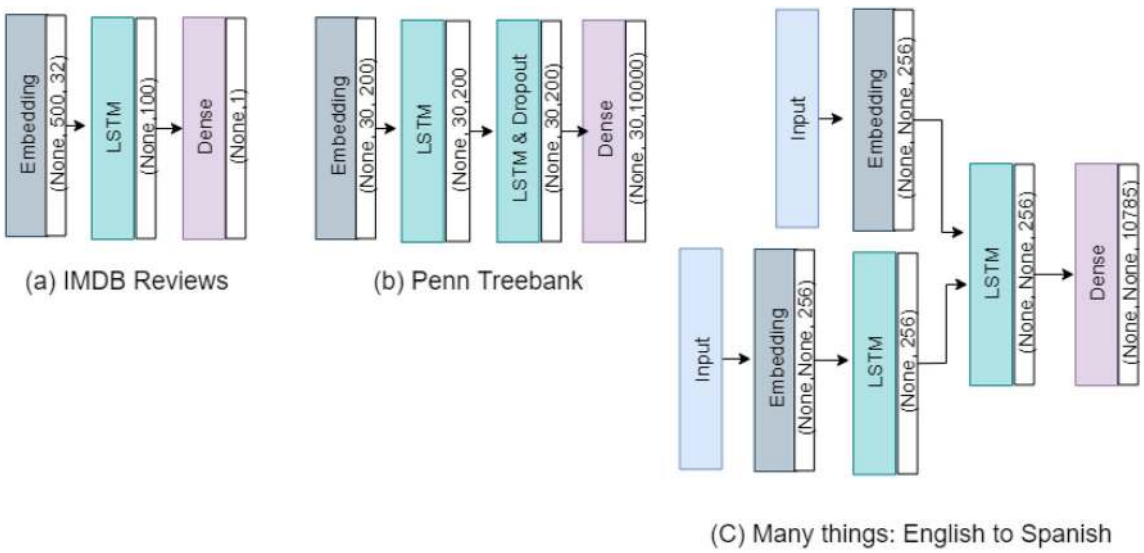

Fig. 3 The architecture of the LSTM used with each of `IMDB Reviews`, `Penn Treebank` and `Many things: English to Spanish`

layer with `softmax` activation over the vocabulary dimensionality to predict the next word at each timestep. The model is optimized for 50 epochs using the `Adadelta` optimizer and a batch size 20.

Many things: English to Spanish The dataset contains 100,000 sentence pairs in English and Spanish from the Tatoeba Project. This project consist of a large database of example sentences translated into many languages. Those translations have been prepared by native speakers of their respective languages, so most of the sentences are error-free. The training dataset contains 80,000 examples (80% of the original dataset), and the testing dataset contains 20,000 examples (20% of the original dataset). The machine translation task is a sequence-to-sequence generation task, in which the encoder-decoder architecture is used for the task. The architecture used (Fig. 3c) consists of two Input layers, two `embedding` layers of size 256, two `LSTM` layers with hidden size of 256, and `dense` layer with `softmax` activation over the vocabulary dimensionality to predict the next word at each time step. The model is optimized for 100 epochs using the `AdamDelta` optimizer and the batch size is 128.

4.3 Reference model and dataset for regional-CNN

For the regional-CNN, we used Faster R-CNN [37] on `VOC2012` dataset.

VOC2012 This dataset contains the data from the PASCAL Visual Object Classes Challenge 2012 corresponding to the Classification and Detection competitions. The dataset contains 11,540 images where each image contains a set of objects, out of 20 different classes. In the classification task, the goal is to predict the set of labels contained in the image. The training dataset contains 5717 examples (50% of the original dataset), and the testing dataset contains 5823 images (50% of the original dataset). For this datset, we use Faster R-CNN which consists of two stages. In the first stage, the input images are processed using the feature extractor. The result of this process is a map of features that is fed to the next stage in which it consists of two main networks. The first one is the RPN, which is mainly responsible for generating regions (called a region proposal), on the basis of which the second network performs structure detection. This is directed at those regions that most likely contain objects. The `ReLU` activation function was used in order to train Faster R-CNN. The network is trained for 100 epochs with `Adam` optimizer and learning rate of 10^{-4}. We also use a weight decay parameter of 5×10^{-4}. We use ResNet-50 [17] as a feature extractor.

4.4 Hardware and Software Resources

We conducted our experiments on two hardware environments: a CPU environment and a GPU environment. The CPU environment runs on CentOS release 7.5.1804 with a total of 64 cores of Intel Xeon Processor (Skylake, IBRS) @ 2.00GHz;240 GB DIMM memory; and 240 GB SSD data storage. The GPU experiments are performed on a single machine running on Debian GNU/Linux 9 (stretch) with an 8 core Intel(R) Xeon(R) CPU @ 2.00GHz; NVIDIA Tesla P4;36 GB DIMM memory; and 300 GB SSD data storage. 'psutil' library of Python along with 'subprocess' and 'memory-profiler' python modules were used for monitoring and logging the system resource utilization values of the experiments. For all DL frameworks, we used `CUDA` 10.0, and `cuDNN` 7.3. All the frameworks have been used with their default configuration settings. Table 2 lists the versions of the deep learning frameworks considered in this study on both the CPU and GPU environments.

4.5 Evaluation metrics

We introduce the metrics we used to evaluate the performance of the different deep learning models.

- *Training time* It is the time spent on building a DNN model over the training dataset through an iterative process.
- *Prediction accuracy* The accuracy metric measures the utility of the trained DNN model at testing phase.
- *CPU utilization* This metric quantifies how frequently CPU is utilized during the training of the deep learning models. This utilization is measured as the average utilization of all CPU cores as shown in Eq. (1). The higher the value of the average utilization, the higher CPU utilization is during the training of a deep learning model.

$$CPU_{AvgUtilization} = \frac{\sum_i^n (CPU_{CoreUtilization}{}^i)}{n} \quad (1)$$

where n is the total number of CPU cores for training a deep learning model, i is the index of the CPU core, and $CPU_{CoreUtilization}$ is the utilization of a single CPU core and is defined in Eq. (2).

$$CPU_{CoreUtilization} = \frac{T^C_{active} \times 100}{T_{total}}\% \quad (2)$$

In Eq. (2), T_{total} denotes the total training time, and T^C_{active} indicates the active time of the CPU core.

- *GPU utilization* This metric quantifies how frequently GPU is utilized during the training of deep learning models. This metric is defined in Eq. (3).

Table 2 The versions of deep learning frameworks included in this study on CPU and GPU environments

Framework	CPU version	GPU version
TensorFlow	1.11.0	1.11.0
Keras	2.2.4 on TensorFlow version 1.11.0	2.2.4 on TensorFlow version 1.11.0
PyTorch	0.4.1	0.4.1
MXNet	1.3.0	1.3.0
Theano	1.0.2	1.0.2
Chainer	4.5.0	4.5.2

$$GPU_{Utilization} = \frac{T^{G}_{active} \times 100}{T_{total}}\% \quad (3)$$

where T^{G}_{active} indicates the active time of the GPU.

- *Memory usage* This metric is defined as the average memory usage during the training process.

5 Experimental results

Our experiments aim to examine the following: (1) the impact of the default configuration on the time and accuracy of each DL framework using different DL architectures on different datasets, (2) how well each DL framework utilize resources using different deep learning architectures on both GPU and CPU environments. The Wilcoxon signed-rank test [50] was conducted to determine if a statistically significant difference in terms of accuracy, training time and resource consumption exists between the different DL frameworks. We present the experimental results in two subsections; CPU results and GPU results on different datasets using different deep learning architectures. We mainly focus on accuracy, running time, convergence and resource consumptions. For the CPU-based experiments on CNN, we use only three datasets: *MNIST*, *CIFAR-10*, and *CIFAR-100*. We excluded the *SVHN* dataset from the CPU-based experiments as all frameworks spent more than 24 h for processing its associated model. For the GPU-based experiments on CNN, we used the four datasets: *MNIST*, *CIFAR-10*, *CIFAR-100*, and *SVHN*. For both CPU and GPU experiments on LSTM, we used *IMDB Reviews*, *Penn Treebank*, and *Many things: English to Spanish* datasets. For both CPU and GPU experiments on Faster R-CNN, we used *VOC2012*. In practice, the processing time and accuracy can be slightly different from one run to another based on the random initialization technique used by the DL framework. Thus, we conducted 5 runs for each experiment where the reported results represent the average of them. Due to space limitation, we report here the most important results of our experiments. For the detailed results, we refer the readers to our project repository.

5.1 Accuracy

Figure 4 shows the testing accuracy of six deep learning frameworks using their own default configuration on CNN and LSTM architectures. Results show that there is no single deep learning framework outperforms the accuracy of all other frameworks across all datasets on different DL architectures. For CNN on *MNIST*, all the deep learning frameworks achieve comparable accuracy around 98% except `Chainer` achieves 96.4%. For CNN on *MNIST*, the differences in accuracy between all frameworks are not statistically significant. For *CIFAR-10* dataset, `TensorFlow`, `Keras`, `MXNet` and `Theano` come in the first place achieving a comparable accuracy of 80%, followed by `Chainer` (73%), while `PyTorch` comes in the last place (72%), as shown in Fig. 4a. For CNN on *CIFAR-10*, the differences in accuracy between `Keras`, `TensorFlow`, `MXNet`, and `Theano` are not statistically significant, while the differences in accuracy between each of `PyTorch` and `Chainer` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05). For *CIFAR-100*, `Keras` achieves the highest accuracy of 53.8% for 200 epochs with 24 h time limit, while `Chainer` achieves the lowest accuracy of 28.3% on CNN architecture, as shown in Fig. 4a. For CNN on *CIFAR-100*, the differences in accuracy between all frameworks except between `TensorFlow` and `MxNet` are statistically significant with more than 95% level of confidence (p value <0.05). For LSTM on *IMDB Reviews*, `Keras`, `Pytorch`, `TensorFlow`, `Chainer` and `MXNet` achieve comparable accuracy (between 87 and 88%), while `Theano` achieves the lowest accuracy of 50%, as shown in Fig. 4b. For LSTM on *IMDB Reviews*, the differences in accuracy between `Theano` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05), while the differences in accuracy between the rest of the frameworks are not statistically significant. All frameworks on *Penn Treebank* dataset achieve a comparable low performance between 17 and 21.7%, as shown in Fig. 4b. For *Penn Treebank* on LSTM, the differences in accuracy between `MXNet` and the rest of the frameworks are statistically significant with more than

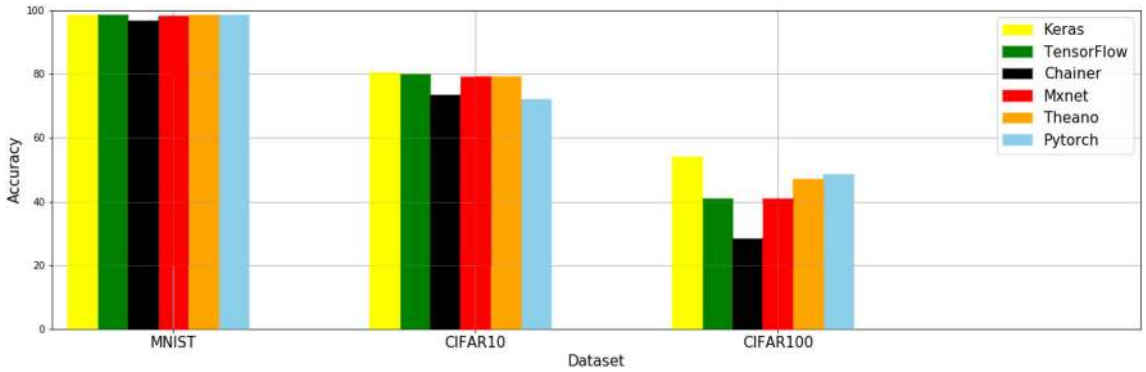

Fig. 4 Accuracy of deep learning frameworks across different datasets on CPU environment using CNN and LSTM architectures

95% level of confidence (p value <0.05), while the differences in accuracy between the rest of the frameworks are not statistically significant. For *Many things* dataset, `Chainer` achieves the highest accuracy of 99.7% while `Keras` achieves the lowest accuracy of 73.3% on LSTM architecture. For *Many things* on LSTM, the differences in accuracy between all frameworks are statistically significant with more than 95% level of confidence (p value <0.05). For Faster R-CNN, `TensorFlow`, `MXNet`, and `Theano` achieve comparable accuracy of 63%, while `Keras` and `Chainer` achieve accuracy of 62% and 53%, respectively. `Pytorch` achieves the lowest accuracy of 51%. For Faster R-CNN architecture on VOC2012, the differences in accuracy between each of `PyTorch` and `Chainer` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05). The differences in accuracy between `Keras`, `TensorFlow`, `MXNet`, and `Theano` are not statistically significant.

In summary, all deep learning frameworks achieve the highest accuracy on the sparse gray-scale *MNIST* dataset due to its low entropy that allows the deep learning frameworks to be able to learn easier. On average, the default setting of `Keras` on the CNN architecture achieves relatively higher performance than the default setting of the other frameworks. On average, such difference in accuracy between `Keras` and other frameworks on the CNN architecture is not statistically significant. On average, the default configuration of `Chainer` on LSTM architecture achieves better performance than the configuration of the other frameworks. On average, the difference in accuracy between `Chainer` and other frameworks on the LSTM architecture is statistically significant with more than 95% level of confidence (p value <0.05).

5.2 Training time

Figure 5 shows the training time of six deep learning frameworks using their own default configuration on CNN and LSTM architectures. Figure 5a shows that `Chainer` has the highest training time across all the datasets on CNN. `Chainer` takes 1 h and 30 min on MNIST, 13 h and 44 min on CIFAR-10 and more than 24 h on CIFAR-100. The differences in training time between `Chainer` and all other frameworks on the CNN architecture are statistically significant with almost 100% level of confidence. `Keras` has the smallest training time on *MNIST* (6 min), *CIFAR-10* (1 h and 12 min) and *CIFAR-100* (5 h and 48 min) on CNN architecture. Such difference in training time between `Keras` and the rest of the frameworks on the CNN architecture are statistically significant with almost 99% level of confidence. `TensorFlow` spent the second smallest running time on the CNN architecture across all datasets. Such difference in training time between `TensorFlow` and the rest of the frameworks on the CNN architecture are statistically significant with almost 99% level of confidence. For the LSTM architecture, `TensorFlow` takes the smallest training time on *Penn Treebank* and *Many things*. For the LSTM architecture, the differences in training time between `TensorFlow` and the rest of the frameworks on *Penn Treebank* and *Many things* are statistically significant with almost 100% level of confidence except for the differences between `PyTorch` and the rest of the frameworks on *Many things* which are not statistically significant. Both `Pytorch` and `Tensorflow` show similar training time on *IMDB Reviews* dataset (around 4 h) on the LSTM architecture. `Theano` has the longest training time on LSTM across all the datasets (11 h and 6 min on *IMDB Reviews*, more than 24 h on both *Penn Treebank* and *Many things*). Such differences in training time between `Theano` and the rest of the frameworks on the LSTM architecture are statistically significant with almost 100% level of confidence. For the Faster R-CNN on VOC2012, the training time of `Keras`, `Chainer`, and `Pytorch` takes more than 24 h. The training time of `MXNet` is 21 h and 5 min while and `TensorFlow` 19 h and 7 min. For the Faster R-CNN, the differences in training time between all frameworks except

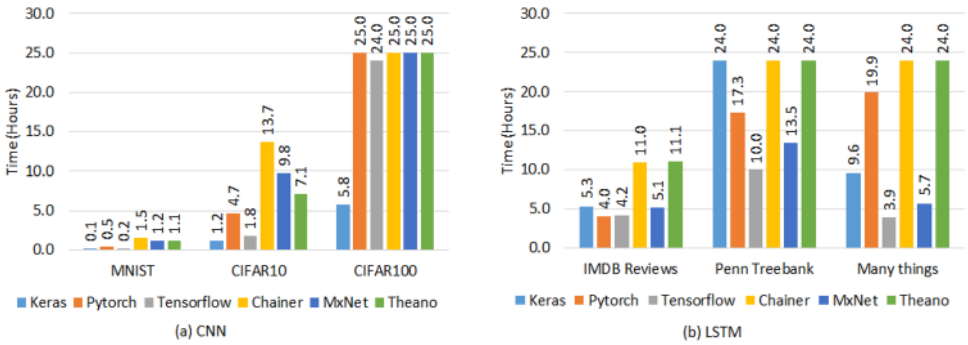

Fig. 5 Training time of deep learning frameworks across different datasets on CPU environment using CNN and LSTM architectures

between `TensorFlow` are the rest of the frameworks are not statistically significant.

In summary, we conclude that longer training time by DL frameworks does not necessarily contribute to better accuracy. For example, for both CNN and Faster R-CNN, `Chainer` spent the longest training while achieves considerable lower performance than other frameworks, such as `TensorFlow`. Also, `Keras` spent the smallest training time on CNN while on average achieves higher accuracy than other framework on CNN.

5.3 Resource consumption

Figure 6 shows the mean CPU consumption for different frameworks on CNN and LSTM architectures during training at 1-s interval. For the CNN architecture, the results show that `MXNet` has the lowest CPU usage across all the datasets, while `Pytorch` has the highest CPU usage on *MNIST* and *CIFAR10* and `Keras` has the highest CPU usage on *CIFAR100*. `Theano` has the second lowest CPU consumption on *MNIST* and *CIFAR10* while `Chainer` has the second lowest on *CIFAR100*, as shown in Fig. 6a. For CNN architecture, the differences in CPU consumption between all frameworks on all datasets are statistically significant with more than 95% level of confidence (p value <0.05). For the LSTM architecture, `Theano` has the lowest CPU usage on *Many things* and *IMDB Reviews* datasets, while `Keras` has the lowest CPU consumption on *Penn Treebank*. `Pytorch` has the highest CPU consumption on *Penn Treebank* and *IMDB Reviews*, while `Keras` has the highest CPU consumption on *Many things*, as shown in Fig. 6b. For LSTM, the differences in CPU consumption between all frameworks on all datasets are statistically significant with more than 95% level of confidence (p value <0.05) except the difference between `Chainer` and `MXNet` on *Penn Treebank* which is not significant. For Faster R-CNN, the mean CPU consumption during training at 1-s for different frameworks is shown in Fig. 7a. `Pytorch` has the highest memory consumption while `MXNet` has the lowest memory consumption. For Faster R-CNN, `Keras`, `TensorFlow`, and `Theano` have comparable memory consumption as shown in Fig. 7a. For Faster R-CNN, the differences in CPU consumption between all frameworks are statistically significant with more than 95% level of confidence (p value <0.05).

Figure 8 shows the memory consumption for different frameworks on CNN and LSTM architectures. `Tensorflow` has the highest memory consumption across all datatsets on CNN architecture, while `MXNet` has the lowest memory consumption on *CIFAR10* (403MB)and *CIFAR100* (751MB), as shown in Fig. 8a. For CNN architecture, `Keras`, `Chainer`, and `Theano` have comparable memory consumption across all the datasets. For LSTM architecture, `Chainer` has the lowest memory consumption on *Penn Treebank* (396.2MB) and *Many things* (1065.4MB), as shown in Fig. 8b. For LSTM architecture, `Pytorch` and `Chainer` have comparable memory consumption on *IMDB Reviews* and *Penn*

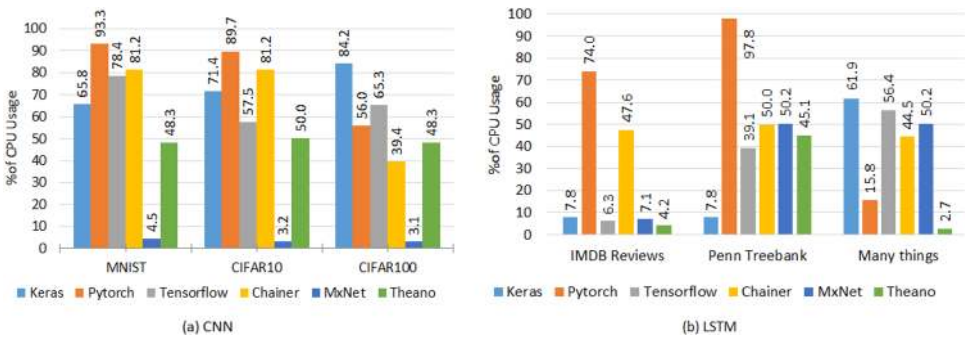

Fig. 6 Mean CPU consumption of the different deep learning frameworks on CPU environments using CNN and LSTM architectures

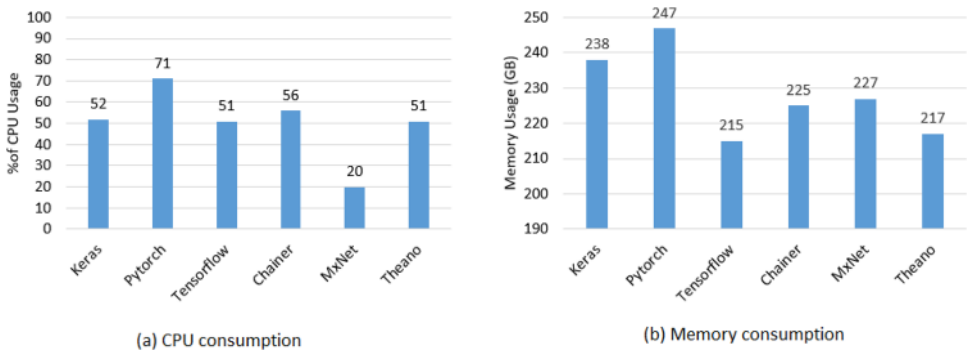

Fig. 7 CPU and memory consumption of the different deep learning frameworks on CPU environments using Faster R-CNN architecture on VOC2012

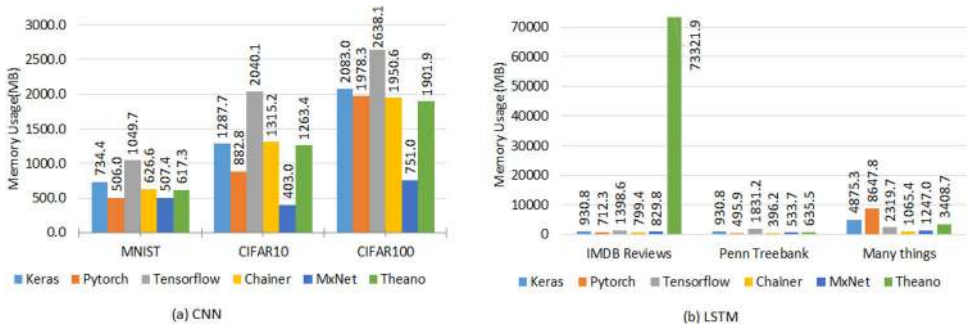

Fig. 8 Memory consumption of the different deep learning frameworks on CPU environments using CNN and LSTM architectures

Treebank. `TensorFlow` has the highest memory consumption on *Penn Treebank* (1831.2MB), while `Pytorch` has the highest memory consumption on *Many things* (8647.8MB). It is notable that `Theano` has the highest memory consumption on *IMDB Reviews* dataset; more than 50*x* the consumption of the second highest DL framework. The main reason behind such huge memory consumption is that `Theano` suffers from a memory leak problem and amount of memory consumed increases significantly over time during training. For Faster R-CNN, the memory consumption for different frameworks is shown in Fig. 7b. `TensorFlow` has the lowest memory consumption (215GB), followed by `Theano` (217GB), while `Chainer` and `MXNet` have a comparable memory consumption. `PyTorch` has the highest memory consumption of 247GB, as shown in Fig. 7b. For CNN, LSTM, and Faster R-CNN, the differences in memory consumption between all frameworks on all datasets are statistically significant with more than 95% level of confidence (p value <0.05).

In Summary, we conclude that higher resource consumption(CPU and memory) may not result in shorter training time and better accuracy. For example, `Pytorch` has the highest CPU consumption while comes in the third place in terms of training time across most of the datasets on CNN, LSTM, and Faster R-CNN architectures.

5.4 Convergence

Figure 9 shows the impact of varying the number of epochs on the performance of the deep learning frameworks on the CNN architecture. The results show that the testing accuracy increases as the number of epochs increases. For the CNN architecture on *MNIST* dataset, `PyTorch`, `TensorFlow`, `Theano`, `MXNet` and `Keras` reach their peak accuracy at around 12 to 14 epochs, while `Chainer` takes larger number of epochs to reach its peak accuracy. For the CNN architecture on *CIFAR-10* dataset, `PyTorch` and `Chainer` reach their peak accuracy at around 60 epochs, while the rest of the frameworks reach their peak accuracy between the 80th of 90th epochs. For the CNN architecture on *CIFAR-100* dataset, `Keras` and `TensorFlow` reach their peak accuracy at around 80 epochs. Figure 10 shows the impact of varying the number of epochs on the performance of the deep learning frameworks on the LSTM architecture. Overall, for `PyTorch`, `TensorFlow`, `Chainer`, `MXNet` and `Keras` on *IMDB Reviews*, the accuracy first increases rapidly to reach the peak value at around 10th epoch and then stays stable or slightly drops especially in `MXNet`. Figure 10a) shows that `Theano` experiences more accuracy fluctuations with the peak accuracy of 90% at the 19th epoch, followed by a significant drop in the accuracy at the 20th epoch. Figure 10b shows that on *Penn Treebank*, `Theano`, `TensorFlow`, `Chainer`, `MXNet` and `Keras` reach their peak accuracy between the 10th and 20th epochs, while `PyTorch`

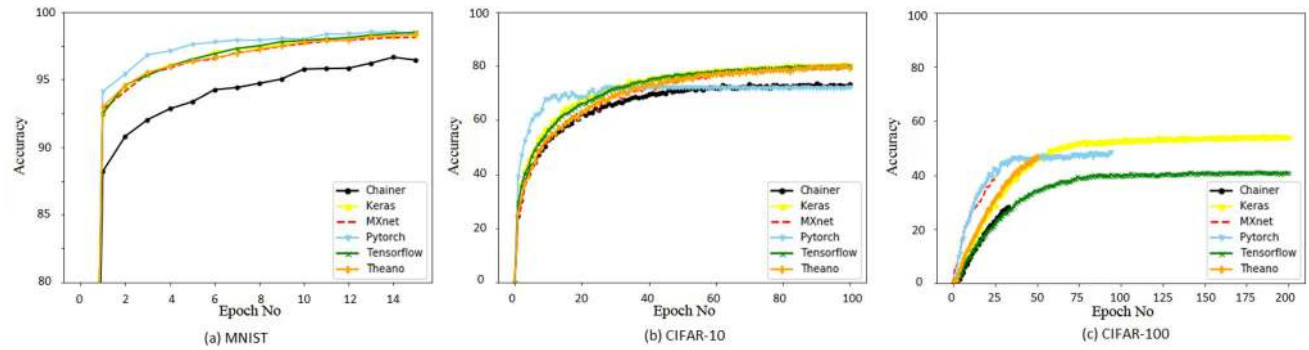

Fig. 9 Convergence of CNN on CIFAR-10, CIFAR-100 and MNIST for deep learning frameworks running on CPU

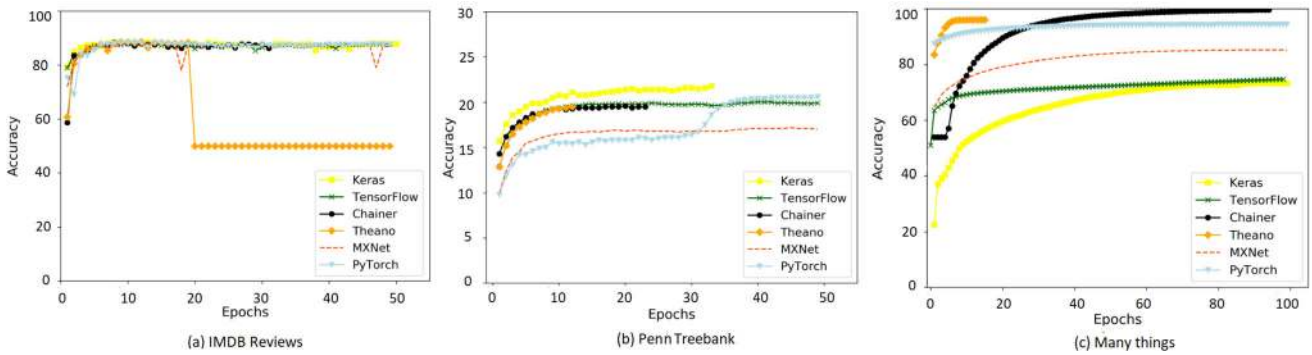

Fig. 10 Convergence of LSTM on IMDB Reviews, Penn Treebank and Many things for deep learning frameworks running on CPU

reaches its peak accuracy of 20% at 37th epochs. Figure 10c shows that on *Many things*, `PyTorch`, `TensorFlow` and `Theano` reach their peak accuracy at early epochs (30th epoch), while `Keras`, `MXNet` and `Chainer` reach their peak accuracy between the 60th and 80th epochs. Figure 11 shows the accuracy converging curves of VOC2012 for different deep learning frameworks on Faster R-CNN architecture on the CPU environment. Figure 11 shows that `PyTorch`, `TensorFlow`, `MXNet`, and `theano` reach their peak accuracy at around the 40th epochs, while `Keras` and `Chainer` reach their peak accuracy of 62% at the 37th epochs and 53% at the 20th epochs, respectively after 24 h time limit.

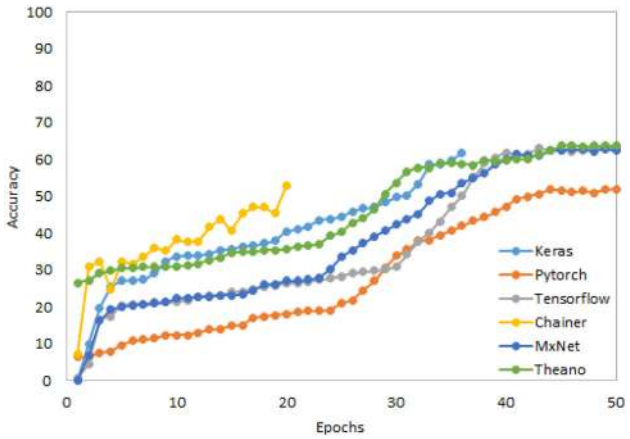

Fig. 11 Convergence of VOC2012 for deep learning frameworks running on CPU

In summary, we conclude that the impact of the number of epochs on the CNN architecture confirms that the training time is proportional to the number of the epochs independently of the dataset or DL framework choices. Generally for LSTM, CNN, and Faster R-CNN architectures, increasing the number of epochs is associated with an increase in model accuracy for most frameworks. However, we noticed that no single framework is able to reach its peak accuracy in earlier epochs than other frameworks across all datasets using different architectures.

5.5 Results of GPU-based experiments

5.5.1 Accuracy

Figure 12 shows the testing accuracy achieved by the different deep learning frameworks on CNN and LSTM architectures using GPU environment. As shown in Fig. 12a, for the *MNIST* and *SVHN* datasets on the CNN, all the deep learning frameworks achieve a comparable accuracy of around 98% and 97%, respectively. For *MNIST* and *CIFAR-10* on CNN, there is no notable accuracy change from running on a CPU or GPU environment. For

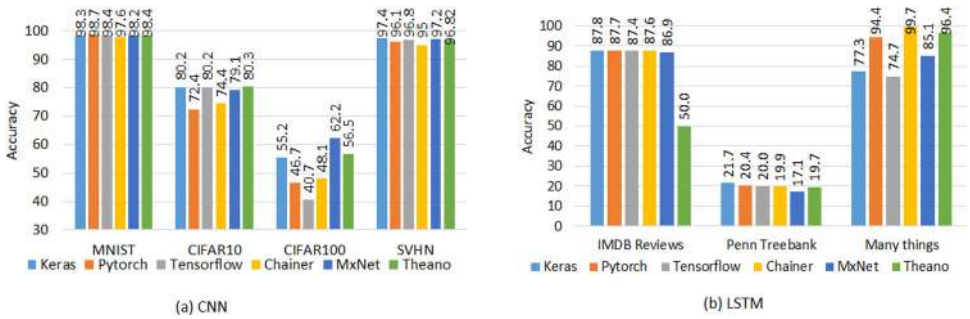

Fig. 12 Accuracy of deep learning frameworks across different datasets on single GPU using CNN and LSTM architectures

CIFAR-100 on CNN, `MXNet` outperforms all DL frameworks by achieving an accuracy of 62.2% while `TensorFlow` achieves the lowest accuracy of 40.7%. For *CIFAR-100* on CNN, the differences in accuracy between each of `MXNet` and `TensorFlow` and the rest of the DL frameworks are statistically significant with more than 95% level of confidence (p value <0.05). Figure 12b shows that `PyTorch`, `TensorFlow`, `Chainer`, `MXNet` and `Keras` on *IMDB Reviews* achieve comparable performance of between 87 and 88%, while `Theano` achieve the lowest accuracy of 50%. For *IMDB Reviews* on LSTM, the differences in accuracy between `Theano` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05), while the differences in accuracy between the rest of the frameworks are not statistically significant. All the frameworks achieve comparable performance of 17% and 22% on the *Penn Treebank* dataset, as shown in Fig. 12b. `Chainer` achieves the highest accuracy of 99.7% on *Many things* dataset, followed by `Theano` (96.4%) and `PyTorch` (94.4%). For *Many things* on LSTM, the differences in accuracy between `Chainer` and each of `Theano` and `PyTorch` are not statistically significant, while the differences in accuracy between `Chainer` and the rest of the frameworks are statistically significant with level of confidence between 97 and 99% (p value <0.05). `Tensorflow` achieves the lowest accuracy of 74.7% on *Many things*, as shown in Fig. 12b. For Faster R-CNN, `TensorFlow`, `MXNet`, and `Theano` achieve comparable accuracy of 63.3% and there is no notable accuracy change from running on a CPU or GPU environment. `Keras` and `Chainer` achieve accuracy of 63% and 59%, respectively, while `Pytorch` achieves the lowest accuracy of 51.4%.

In summary, we conclude that there is no notable accuracy change from running on CPU or GPU environments using LSTM, CNN, and Faster R-CNN architectures across all datasets except *CIFAR-100* and *VOC2012*. For *CIFAR-100* on CNN, we witnessed significant performance boost with `Chainer`, `MXNet`, and `Theano`. For *VOC2012* on Faster R-CNN, we witnessed significant performance boost with `Chainer` and `Keras`.

5.5.2 Training time

Figure 13 shows the training time of the different DL frameworks on both CNN and LSTM architectures. For *MNIST* on CNN, `Chainer`, `Tensorflow` and `Keras` have comparable running times (1 min and 3 s), followed by `Theano`, while `MXNet` has the longest training time (6 min and 33 s). The differences in training time between `MXNet` and all other frameworks on the CNN architecture are statistically significant with more than 95% level of confidence (p value <0.05). For *MNIST*, `Chainer` has the highest running time speedup using the GPU over its CPU-based performance, while `PyTorch` achieves the smallest speedup. For *CIFAR-10* on CNN, `Chainer` takes the shortest training time (7 min and 42 s) followed by `Keras` (10 min and 38 s). For *CIFAR-10* on CNN, the differences in training time between `Keras`, `Tensorflow`, `Chainer`, and `Theano` are not statistically significant, while the differences between each of `PyTorch` and `MXNet` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05). `TensorFlow` comes in the third place (10 min and 46 s) while `PyTorch` comes in the last place (43 min and 48 s). For *CIFAR-10*, `Chainer` gains the most benefits from GPU acceleration, while `PyTorch` gains the least. For *CIFAR-100*, `Keras` achieves the shortest training time (1 h and 47 min) followed by `TensorFlow` (1 h and 48 min) while `Theano` comes at the last place (4 h and 42 min). For *CIFAR-100* on CNN, the differences in training time between `Keras`, `Pytorch`, `TensorFlow`, and `Chainer` are not statistically significant, while the differences between each of `Theano` and `MXNet` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05).

For the LSTM architecture, `Theano` has significantly shortened training time on GPU compared to CPU. `Theano` is 11 times faster and more than 15 times faster on *IMDB Reviews* and *Penn Treebank* using the GPU over its CPU-based performance. `Keras` benefits the least from using a GPU compared to other frameworks, as shown in Fig. 13b. It is notable that the CPU training time of `Keras` on *IMDB Reviews* is shorter than that using CPU and GPU.

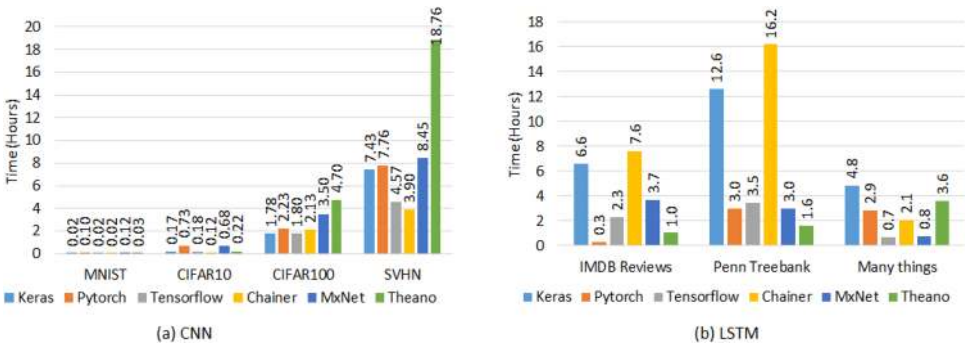

Fig. 13 Training time of deep learning frameworks included in this study on different datasets using CNN and LSTM architectures on GPU environment

We observe a slight improvement using GPU over CPU on `Chainer` by a factor of 1.5 times, more than 1.5 times and more than 11 times on *IMDB Reviews*, *Penn Treebank* and *Many things*, respectively, on LSTM. For the Faster R-CNN on VOC2012, both `PyTorch` and `TensorFlow` achieve the shortest training time of 6 h and 12 min, followed by `MXNet` (7 h and 18 min), while `Chainer` comes in the last place (21 h and 36 min). The differences in training time between each of `Tensorflow`, `Pytorch` and `Chainer` and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05). Both of `Keras` and `Theano` have a comparable training time of 9 h and 36 min. `PyTorch` gains the most benefits from the GPU acceleration while `Chainer` gains the least.

In summary, GPU acceleration shortens the training time by an order of magnitude across most datasets on CNN, Faster R-CNN, and LSTM architectures. However, our empirical evaluation shows that in the case of `Keras` on *IMDB Reviews* using LSTM, CPU without GPU outperforms the CPU with GPU in terms of training time. This observation shows the potential of better deep learning frameworks for a specific hardware configuration platform (CPU only or CPU with GPU).

5.5.3 Resource consumption

Figure 14 shows the mean GPU consumption for different frameworks on CNN and LSTM architectures during training at 1-s interval. The results show that `Theano` on CNN has the highest GPU usage across all of the datasets (Fig. 14a). The differences in GPU consumption between `Theano` and the rest of the frameworks on CNN are statistically significant with more than 95% level of confidence (p value <0.05). `PyTorch` and `MXNet` on CNN have comparable GPU usage on *MNIST*, *CIFAR-10*, and *SVHN*. On CNN, `PyTorch` has the lowest GPU usage on the *MNIST* dataset, followed by `MXNet`. For *CIFAR-10* and *CIFAR-100*, `MXNet` has the lowest GPU usage. For *CIFAR-10* and *CIFAR-100*, the differences in GPU consumption between `MXNet` and the rest of the frameworks on CNN are statistically significant with more than 95% level of confidence (p value <0.05). For *SVHN*, `Keras` has the lowest GPU usage, followed by `TensorFlow`. For *SVHN*, the differences in GPU consumption between each of `Keras` and `TensorFlow` and the rest of the frameworks on CNN are statistically significant with more than 95% level of confidence (p value <0.05). For the LSTM architecture, Fig. 14b shows that `Chainer` has the lowest GPU consumption on all datasets, while `PyTorch` has the highest GPU consumption on *Penn Treebank* (91.6%) and *Many things* (94.6%) and `Theano` has the highest consumption on *IMDB Reviews* (66.6%). For LSTM on all datasets, the differences in GPU consumption between `Chainer` and the rest of the frameworks are statistically significant with a level of confidence between 96 and 98%. For LSTM, the differences in GPU consumption between `PyTorch` and the rest of the frameworks on *Penn Treebank* and *Many things* are statistically significant with more than 95% level of confidence (p value <0.05). For *IMDB*

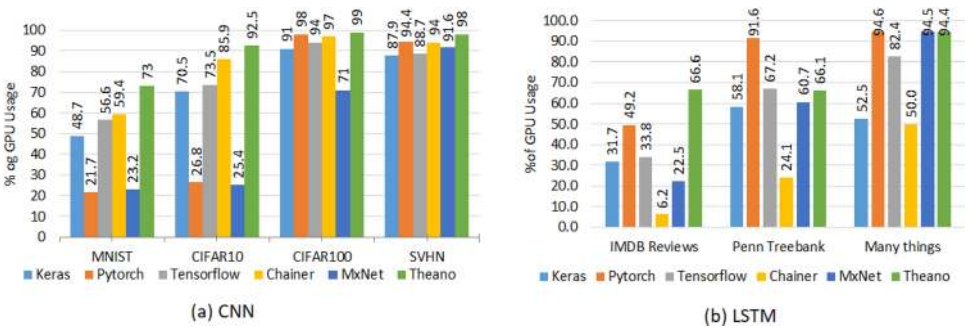

Fig. 14 GPU consumption of the different deep learning frameworks on GPU environment using CNN and LSTM architectures

Reviews on LSTM, the differences in GPU consumption between Theano and the rest of the frameworks are statistically significant with more than 95% level of confidence (p value <0.05). Theano and TensorFlow have comparable GPU consumption on the *Penn Treebank* dataset of between 66 and 67%. Figure 16a shows the mean GPU consumption for different frameworks on Faster R-CNN architecture during training at 1-s interval. The results show that PyTorch has the highest GPU consumption (54%), followed by Chainer (52%), while TensorFlow has the lowest GPU consumption (34%). As shown in Fig. 16a, Keras and MXNet have comparable GPU consumption of 47% and 46%, respectively. The differences in GPU consumption between all frameworks on Faster R-CNN are statistically significant with more than 95% level of confidence (p value <0.05).

Figure 15 illustrates the mean CPU consumption by the different DL frameworks on the GPU environment. The results show that on the CNN architecture, PyTorch has the highest CPU usage across all the datasets on all the DL frameworks, while TensorFlow has the lowest CPU consumption, as shown in Fig. 15a. The differences in CPU consumption between each of PyTorch and TensorFlow and the rest of the frameworks on CNN are statistically significant with a level of confidence between 96 and 98%. For the LSTM architecture, MxNet has the highest CPU consumption on the *Penn Treebank* and *Many things* datasets, while TensorFlow has the highest CPU consumption on *IMDB Reviews* (32.2%), as shown in Fig. 15b. The differences in CPU consumption between MxNet and the rest of the DL frameworks on *Penn Treebank* and *Many things* and between TensorFlow and the rest of the frameworks on *IMDB Reviews* are statistically significant with more than 95% level of confidence (p value <0.05). For the LSTM architecture, Chainer has the lowest CPU consumption on *IMDB Reviews* and *Penn Treebank* datasets, while PyTorch has the lowest CPU consumption on *Many things* (1.5%), as shown in Fig. 15b. For the LSTM architecture, the differences in CPU consumption between Chainer and the rest of the DL frameworks on *IMDB Reviews* and *Penn Treebank* and between PyTorch and the rest of the DL frameworks on *Many things* are statistically significant with more than 95% level of confidence (p value <0.05). Figure 16b shows the mean CPU consumption for different frameworks on Faster R-CNN architecture during training time at 1-s interval. The results show that the MXNet has the highest CPU consumption (18%), followed by Keras (14%) and TensorFlow (13%). PyTorch, Chainer, and Theano achieve the lowest CPU consumption of 12%. For Faster R-CNN, the differences in the CPU consumption between all frameworks are statistically significant with more than 95% level of confidence (p value <0.05)

Figure 17 shows the memory consumption of different DL frameworks using both CNN and LSTM on GPU environment. The results on CNN shows that Chainer has the lowest memory consumption on *MNIST*, *CIFAR-10* and *CIFAR-100*, while PyTorch has the lowest memory consumption on *SVHN* (Fig. 17a). Such differences in memory consumption between Chainer and the other DL frameworks on *MNIST*, *CIFAR-10*, and *CIFAR-100* and between PyTorch and other DL frameworks on *SVHN* are statistically significant with more than 95% level of confidence (p value <0.05). TensorFlow has the highest memory consumption on *MNIST*, *CIFAR-10*, and *CIFAR-100*, while TensorFlow and Keras have the highest memory consumption on *SVHN* (Fig. 17a. For LSTM architecture, Keras and TensorFlow have the highest memory consumption across all the datasets, as shown in Fig. 17b. The differences in the memory consumption between each of Keras and TensorFlow and the rest of the DL frameworks are statistically significant with more than 95% level of confidence (p value <0.05). Chainer has the least memory consumption on *IMDB Reviews*, while Theano has the least consumption on *Penn Treebank*. The differences in memory consumption between Chainer and the rest of the DL frameworks on *IMDB Reviews* and between Theano and other DL frameworks on *Penn Treebank* are statistically significant with more than 95% level of confidence (p value <0.05). Chainer and MXNet have considerably low memory consumption

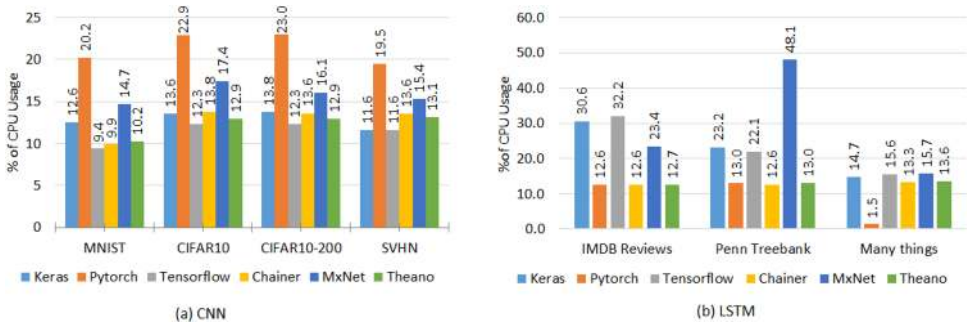

Fig. 15 Mean CPU consumption of the different deep learning frameworks on GPU environment using CNN and LSTM architectures

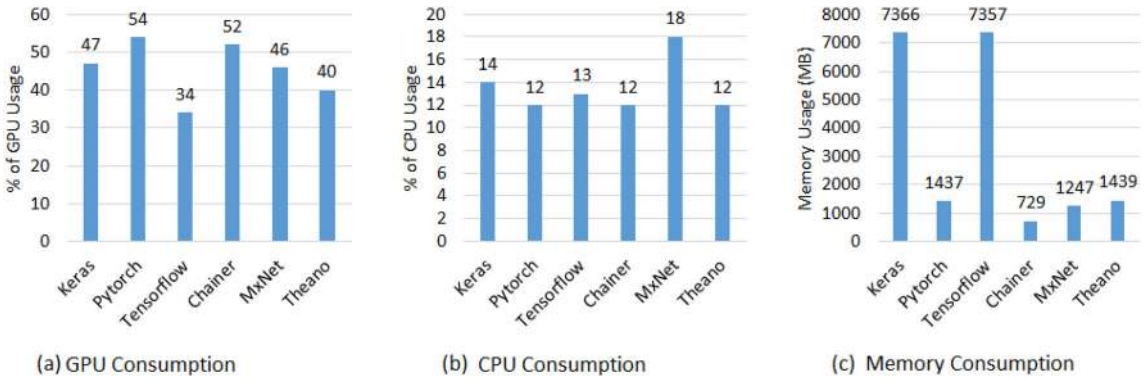

Fig. 16 Mean CPU consumption, memory consumption, and GPU consumption of the different deep learning frameworks on GPU environment using Faster R-CNN architecture

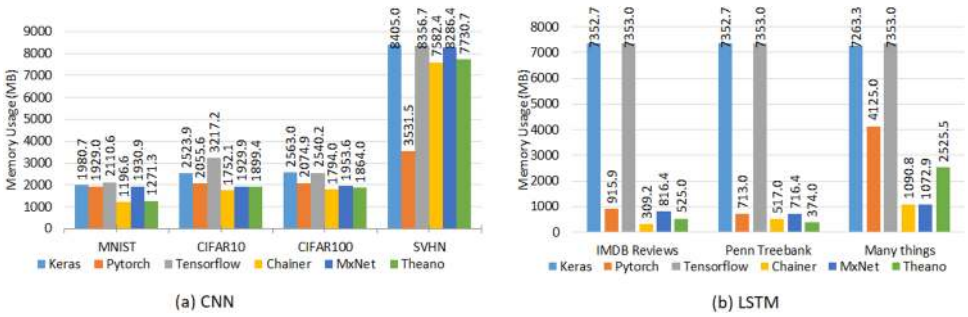

Fig. 17 Memory consumption of the different deep learning frameworks on GPU environment using CNN and LSTM architectures

on *Many things* (1.1GB). Figure 16c shows the memory consumption of different DL frameworks using Faster R-CNN on GPU environment. The results show that `Chainer` has the lowest memory consumption of 729MB, while `Keras` and `TensorFlow` have the highest memory consumption of 7366MB and 7357MB, respectively. For Faster R-CNN, the differences in the memory consumption between all frameworks are statistically significant with more than 95% level of confidence (p value <0.05).

In summary, we conclude that the GPU utilization is generally much higher than the CPU utilization. In most of the times on the CNN architecture, the GPU utilization is close to 100%, while each CPU core utilization ranges from 9.3 to 23%. In addition, when the GPU utilization is high, the CPU utilization tends to be low, and vice versa. This indicates that the workload between the CPU and GPU is not well balanced due to the lack of effective coordination between them.

5.5.4 Convergence

Figure 18 shows the impact of increasing the number of epochs on the performance of the DL frameworks on the CNN architecture on the GPU environment. The results show that the accuracy of `PyTorch` increases rapidly to reach the optimal value earlier than other frameworks on CNN. For *MNIST*, the results show that the accuracy of `Theano`, `MXNet`, `TensorFlow` and `Keras` increase gradually to achieve their peak accuracies at between the 12th and 14th epochs, as shown as shown in Fig. 18a. For *CIFAR-10*, `TensorFlow`, `Keras`, `Theano` and `MXNet` have comparable performance on achieving the peak accuracy between the 80th and 90th epochs, as shown in Fig. 18b. On *CIFAR-100*, all the frameworks achieve their peak accuracy between the 60th and 75th epochs. For the *SVHN* dataset, `Keras`, `MXNet`, `Theano`, `TensorFlow` reaching their peak accuracy early between the 20th and 30th epochs, while `PyTorch` and `Chainer` experience slight drops in the accuracy and reach their peak accuracy between the 40th and 60th epochs. Figure 19 shows the impact of increasing the number of epochs on the performance of the DL frameworks on the LSTM architecture on the GPU environment. The results show that the accuracy of `MXNet`, `Keras`, `TensorFlow`, `PyTorch`, and `Chainer` on the *IMDB* dataset increases rapidly and stays stable or slightly drops, while `Theano` reaches a peak accuracy of 85% and then experiences a significant drop in the performance after the 20th epoch, as shown in Fig. 19a. On the *Penn Treebanks* dataset, `Theano`, `Chainer`, `Keras`, and `TensorFlow` achieve the comparable accuracy of between 20 and 22% between the 10th and 20th epochs, while `PyTorch` takes longer epochs to reach the peak accuracy of 20% between 35th and 40th epoch, as shown in Fig. 19b. On the *Many things* dataset, `PyTorch`

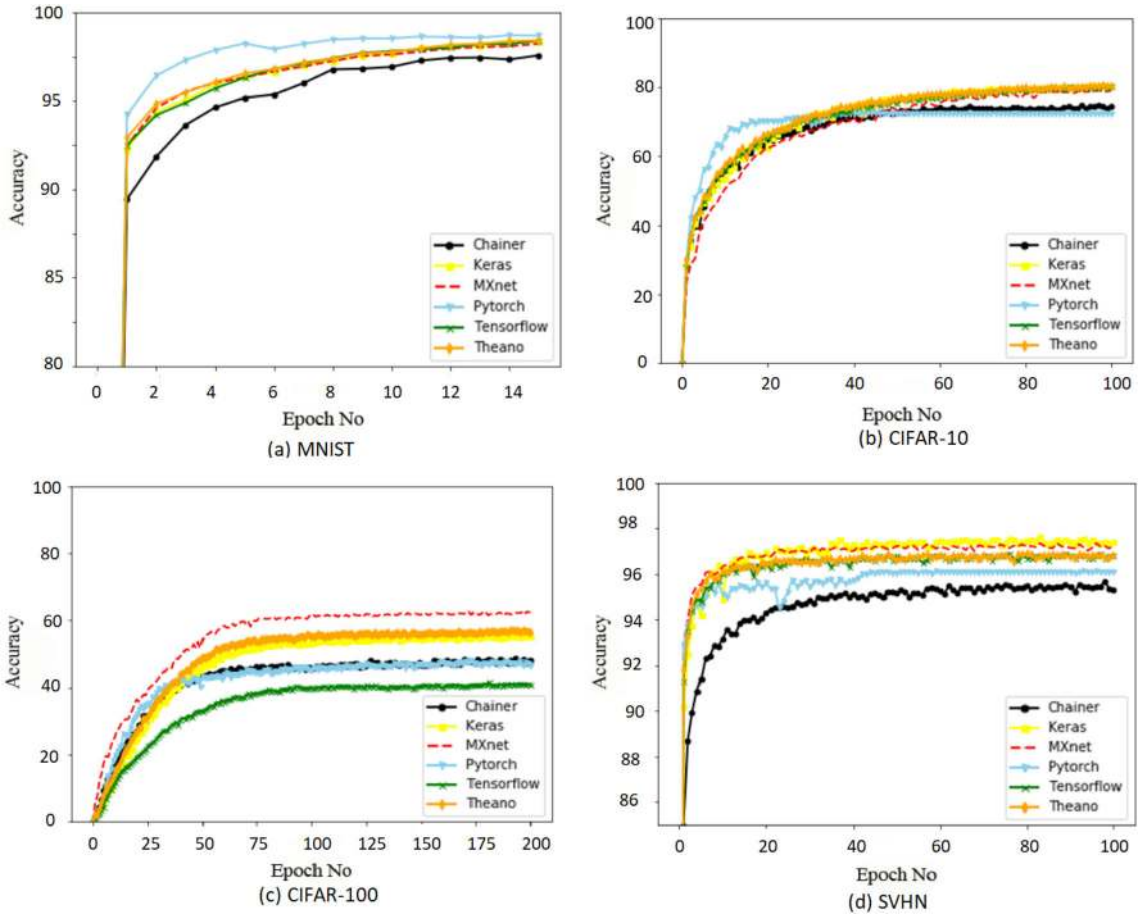

Fig. 18 Convergence of CNN on CIFAR-10, CIFAR-100 and MNIST for deep learning frameworks running on CPU

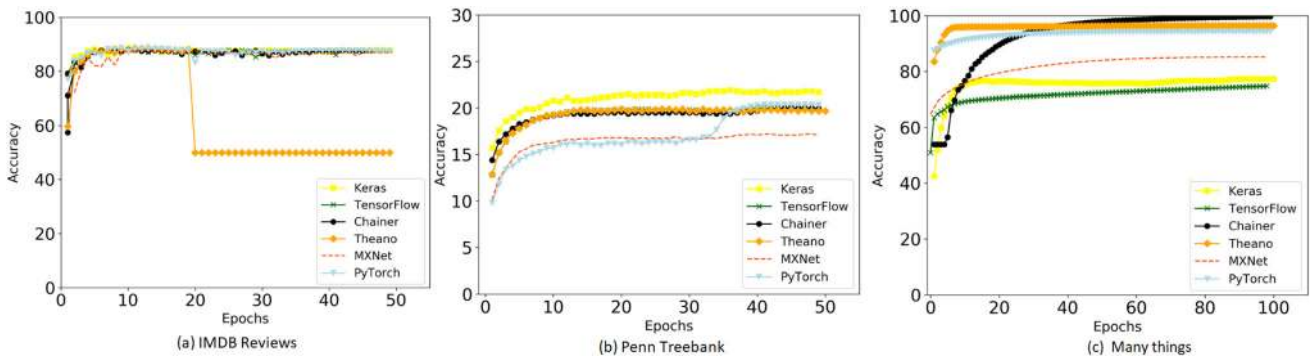

Fig. 19 Convergence of LSTM on IMDB Reviews, Penn Treebank and Many things for deep learning frameworks running on GPU

and `Theano` achieve the comparable peak accuracy of between 94 and 96% at the 20th and 30th epochs, while `Keras` and `TensorFlow` achieve the comparable accuracy of between 94 and 96%. `Chainer` get benefits from the largest number of epochs to reach the highest accuracy across all the frameworks of 99.7% at the 50th epochs, as shown in Fig. 19c. Figure 20 shows the accuracy converging curves for different deep learning frameworks on Faster R-CNN architecture on GPU environment. Figure 20 shows that `Chainer` reaches a peak accuracy of 59% at the 45th epochs and stays stable, while `PyTorch` experiences significant performance jump after the 30th epochs to reach a peak accuracy of 51.4% at the 48th epochs. `TensorFlow`, `Keras`, `MXNet`, and `Theano` achieve comparable peak accuracy of 63% at the 40th epochs, as shown in Fig. 19..

In summary, the Pearson correlation coefficient [16] between the number of epochs and the accuracy is 0.0005,

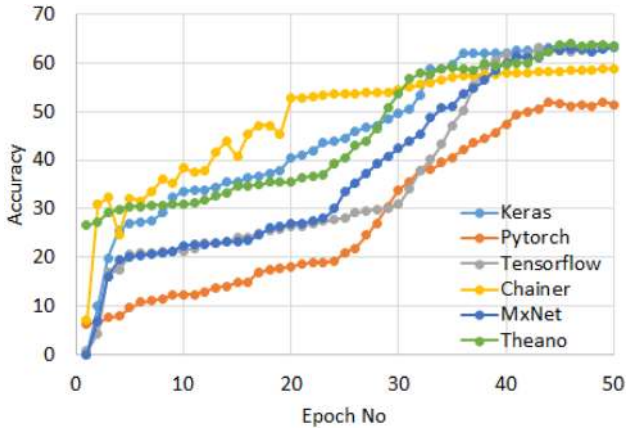

Fig. 20 Convergence of Faster R-CNN on VOC2012 for deep learning frameworks running on GPU

which indicates that there is no linear relationship between the number of epochs and the accuracy. The results on CNN, Faster R-CNN, and LSTM indicate that the training accuracy curve along the number of epochs is almost the same on both CPU and GPU environments.

5.6 Lessons learned

In this section, we report some of the lessons that we have learned during our benchmarking study.

TensorFlow provides the user with the ability to structure every single detail about the neural network layers, `TensorFlow` is considered as a low-level library. It provides more control over all the layers of the network. It has many advanced operations compared to others. In order to load a dataset, we have to do it by ourselves as it does not have a data loader. Building a model using `TensorFlow` is complex and requires the user to go deep into the details of the layers and to structure the dataset. In addition, the user has to explicitly state the bias, weight and input shape of each layer. One of the main limitations we noticed is that we have to write many lines of code compared to the other frameworks. On the other side, `TensorFlow` has a comprehensive documentation set.

Theano configuration is not straightforward. Manual configurations need to be made separately for CPU and GPU based experiments. These configurations range from setting *GPU flags*, *GPU id*, *path to g++* (C++ compiler) etc. These configurations are specified in the '.theanorc' file. Besides `Theano` installation there is a compatible version of `Lasagne` which is a dedicated wrapper library for building and training neural networks on top of `Theano`. In `Lasagne`, we used `pickle` to load the datasets but there exists a library that is easier to get the datasets, named `sklearn`. `Theano` has good documentation that provides examples for using every function in the library. `Lasagne` enables the user to create a custom layer. Many steps needs to be manually managed during the installation for `Theano` and `Lasagne` especially for the GPU environment.

PyTorch is available through Conda and has a smooth installation. `PyTorch` allows us to manipulate tensors, exchange them easily with *NumPy* and perform efficient CPU or GPU calculations and to calculate gradients to apply gradient-based optimization algorithms. `PyTorch` is a comprehensive package containing many sub-packages and most functionalities required in most Machine Learning tasks. Hence, having `PyTorch` alone is sufficient for most Deep Learning tasks and does not require supplementary packages. `PyTorch` has a `utils` package that contains an effective data loader to load the datasets. It also has a package called `torchvision` which contains the popular datasets such as *MNIST*, *CIFAR10*, *SVHN*, and others. We used a `transform` parameter in the data loader which allows us to normalize the datasets. Instead of building a model architecture, `PyTorch` provides the definition for some models such as `Resnet`, `Densenet`, `Resnet` among many others.

MXNet is easy to set-up and has separate installations for CPU and GPU. The GPU versions comes bundled with CUDA, cuDNN. `MXNet` also makes it easier to track, debug, save checkpoints, modify hyper-parameters, such as learning rate or perform early stopping. `MXNet` supports C++ for the optimized backend to get the most of the GPU or CPU available. For the building and training neural networks, scripting options range from Python, R, Scala to JavaScript for user convenience. `MXNet` has a simple API called `Gluon` for deep learning that contains the most known deep learning datasets as *MNIST*, *CIFAR10*, and *CIFAR100*. For example, it provides the `Mxnet.gluon.data.DataLoader` method that has an interesting parameter *last_batch*, which handles the last batch.

Chainer is straight-forward to setup as it is available via pip, however, running models on GPU requires the installation of a separate package called *CuPy*. This package enables CUDA support. In essence, `Chainer` is written purely in Python on top of *NumPy* and *CuPy* Python libraries. In our experience, `Chainer` has been a convenient and easy to use tool in terms of building and training neural networks. `Chainer` supports getting several datasets including *MNIST*, *CIFAR10*, and *SVHN*. `Chainer` has a dataset iterator to loop over the dataset whether in an ordered index or using shuffled order. `Chainer` has many examples of neural nets such as CNN, RNN, DGGAN, and others.

Keras relies on the `TensorFlow` backend. Like `TensorFlow` it needs to be installed separately for GPU and CPU and is available from *Conda* as well. In our experience it has been user friendly, modular, and extensible. `Keras` is an all-inclusive tool and carries a vast

array of functionalities that makes it easy to develop models via scripting. It requires relatively fewer lines of code. `Keras` has a comprehensive and easy to follow documentation. `Keras` has strong built-in functionalities for monitoring training progress and implementing metrics such as the accuracy metric. The layers provided by `Keras` cover almost all requirements to build a specialized neural network. In addition, `Keras` provides a variety of layers to customize your own model. Furthermore, there are many tutorials and resources that could help in designing deep learning models. We used a sequential model which is a linear stack of layers. We used `RELU` and `softmax` activation layers that have already been implemented. For Model optimization, which is one of the two arguments that are required in order to be able to compile any `Keras` model, we used the `stochastic gradient descent` optimizer which includes support for momentum, learning rate decay, and Nesterov momentum. The loss function is the second parameter that is used for compilation. As we are targeting a model for categorical classes, we used `categorical crossentropy` to obtain the target for each image which should be a 10-dimensional vector that is all-zeros except for a one at the index corresponding to the class of this image. `Keras` supports `MSE`, `hinge`, `logcosh`, and many other loss functions.

Recently, the DATA Lab at Texas A&M University released **Auto-Keras**[11] as an open source software library that attempts to automatically search for the architecture and hyperparameters of deep learning models. In order to evaluate this library, we have conducted an experiment on the GPU-based environment using the *CIFAR100* dataset, as it was achieving the lowest accuracy. We made 3 runs using `Auto-Keras` with allocated time budgets of 30 min, 60 min and 24 h to automatically configure the model. The accuracy of the returned models from the 3 runs were 48%, 52% and 54%, respectively. The results show the lack of effectiveness of the auto-tuning technique of the library as it could not outperform the manually designed model executed by `Keras` (55%). In general, auto tuning of deep learning models represents a significant research direction with a big room for improvement.

6 Conclusion and future work

Although the concepts of Artificial Neural Network (ANN) are not new as they originated around the late 1940s, they were, however, difficult to be used because of the intensive need for computational resources and the lack of the amounts of data which is required to effectively train this type of algorithms. Recently, the increasing availability of deep learning frameworks and the ability to use GPUs for performing parallel intensive calculations have paved the way to effectively use the modern deep learning models. Thus, currently, deep learning is revolutionizing the technology industry. For example, modern machine translation tools and computer-based personal assistants (e.g., Alexa) are all powered by deep learning techniques. In practice, it is expected that the applications of deep learning techniques will continue growing as they are increasingly reaching various application domains such as robotics, pharmaceuticals and energy among many others. To this end, we developed `DLBench`, an extensive experimental study for evaluating the performance characteristics of six popular deep learning frameworks on CNN, Fatser R-CNN, and LSTM architectures. The results of our experiments have shown some interesting characteristics of the performance of the evaluated frameworks. In addition, our analysis for the detailed results has provided a set of useful insights.

As a future work, we plan to extend our benchmark to include more frameworks and test different parameter settings for each framework. In addition, we plan to test the influence of more architecture parameterizations sensitivities as we only focused on the influence of ranging epochs in this work.

Acknowledgements The work of Sherif Sakr and Abdul Wahab is funded by the European Regional Development Funds via the Mobilitas Plus programme (Grant MOBTT75). The work of Radwa Elshawi is funded by the European Regional Development Funds via the Mobilitas Plus programme (MOBJD341). The authors would like to thank the students Nesma Mahmoud, Yousef Essam, and Hassan Eldeeb for their involvement on some of the experiments of this work.

References

1. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. OSDI **16**, 265–283 (2016)
2. Abugabah, A., AlZubi, A.A., Al-Obeidat, F.N., Alarifi, A., Alwadain, A.: Data mining techniques for analyzing healthcare conditions of urban space-person lung using meta-heuristic

[11] https://autokeras.com/.

optimized neural networks. Clust. Comput. **23**(3), 1781–1794 (2020)
3. Awan, A.A., Subramoni, H., Panda, D.K.: An in-depth performance characterization of CPU-and GPU-based DNN training on modern architectures. In: Proceedings of the Machine Learning on HPC Environments, p. 8. ACM, (2017)
4. Bahrampour, S., Ramakrishnan, N., Schott, L., Shah, M.: Comparative study of caffe, neon, theano, and torch for deep learning (2016)
5. Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., Bengio, Y.: Theano: new features and speed improvements. arXiv preprint arXiv:1211.5590 (2012)
6. Bengio, Y., et al.: Learning deep architectures for AI. Found. Trends Mach. Learn. **2**(1), 1–127 (2009)
7. Bergstra, J., et al.: Theano: A CPU and GPU math compiler in python. In: Proc. 9th Python in Science Conf, vol. 1 (2010)
8. Chen, T., et al.: Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274 (2015)
9. Chollet, F., et al.: Keras: The python deep learning library. Astrophysics Source Code Library (2018)
10. Coleman, C., et al.: Dawnbench: an end-to-end deep learning benchmark and competition. Training (2017)
11. Collobert, R., et al.: Natural language processing (almost) from scratch. J. Mach. Learn. Res. **12**, 2493–2537 (2011)
12. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: a matlab-like environment for machine learning. In: BigLearn, NIPS workshop, number EPFL-CONF-192376 (2011)
13. Dagum, L., Menon, R.: Openmp: an industry-standard API for shared-memory programming. Comput. Sci. Eng. **1**, 46–55 (1998)
14. Everingham, M., Eslami, S.A., Van Gool, L., Williams, C.K., Winn, J., Zisserman, A.: The pascal visual object classes challenge: a retrospective. Int. J. Comput. Vis. **111**(1), 98–136 (2015)
15. Geng, X., Zhang, H., Zhao, Z., Ma, H.: Interference-aware parallelization for deep learning workload in GPU cluster. Clust. Comput. **23**(4), 2689–2702 (2020)
16. Hauke, J., Kossowski, T.: Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. Quaestiones Geographicae **30**(2), 87–93 (2011)
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
18. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**(7), 33–38 (2008)
19. Hinton, G., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97 (2012)
20. Huval, B., Wang, T., Tandon, S., Kiske, J., Song, W., Pazhayampallil, J., Andriluka, M., Rajpurkar, P., Migimatsu, T., Cheng-Yue, R., et al.: An empirical evaluation of deep learning on highway driving. arXiv preprint arXiv:1504.01716 (2015)
21. Intel caffe. (2017)
22. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: convolutional architecture for fast feature embedding. In: Proceedings of the 22nd ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
23. Jiang, Z., Gao, S.: An intelligent recommendation approach for online advertising based on hybrid deep neural network and parallel computing. Clust. Comput. **23**(3), 1987–2000 (2020)
24. Kim, Y., Lee, J., Kim, J.-S., Jei, H., Roh, H.: Comprehensive techniques of multi-GPU memory optimization for deep learning acceleration. Clust. Comput. **23**(3), 2193–2204 (2020)
25. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images. Technical report, Citeseer (2009)
26. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
27. Liu, J., Dutta, J., Li, N., Kurup, U., Shah, M.: Usability study of distributed deep learning frameworks for convolutional neural networks (2018)
28. Maas, A.L., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y., Potts, C.: Learning word vectors for sentiment analysis. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologiesvol. 1, pp. 142–150. Association for Computational Linguistics (2011)
29. Mahmoud, N., Essam, Y., Shawi, R.E., Sakr, S.: DLBench: an experimental evaluation of deep learning frameworks. In: 2019 IEEE International Congress on Big Data, BigData Congress 2019, Milan, Italy, July 8–13, 2019, pp. 149–156 (2019)
30. Marcus, M., Santorini, B., Marcinkiewicz, M.A.: Building a large annotated corpus of english: the penn treebank (1993)
31. Mkl-dnn for scalable deep learning. (2017)
32. N. Corporation. AI computing leadership from nvidia. In: https://www.nvidia.com/en-us/ (2018)
33. Netzer, Y., et al.: Reading digits in natural images with unsupervised feature learning. In: NIPS Workshop on Deep Learning and Unsupervised Feature Learning (2011)
34. Paszke, A., Gross, S., Chintala, S., Chanan, G.: Tensors and dynamic neural networks in python with strong GPU acceleration (2017)
35. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch (2017)
36. Rajpurkar, P., Zhang, J., Lopyrev, K., Liang, P.: Squad: 100,000+ questions for machine comprehension of text. arXiv preprint arXiv:1606.05250 (2016)
37. Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: Towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems, pp. 91–99 (2015)
38. Sak, H., Senior, A., Beaufays, F.: Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Fifteenth Annual Conference of the International Speech Communication Association (2014)
39. Seide, F., Agarwal, A.: Cntk: Microsoft's open-source deep-learning toolkit. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 2135. ACM (2016)
40. Shams, S., Platania, R., Lee, K., Park, S.-J.: Evaluation of deep learning frameworks over different HPC architectures. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1389–1396. IEEE (2017)
41. Shen, D., Wu, G., Suk, H.-I.: Deep learning in medical image analysis. Annu. Rev. Biomed. Eng. **19**, 221–248 (2017)
42. Shi, S., Wang, Q., Xu, P., Chu, X.: Benchmarking state-of-the-art deep learning software tools. In: IEEE CCBD (2016)
43. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
44. Team, H.: High performance deep learning project. Int. J. Comput. Vis. (2017)
45. Tokui, S., Oono, K., Hido, S., Clayton, J.: Chainer: a next-generation open source framework for deep learning. In: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS) (2015)
46. Tokui, S., Oono, K., Hido, S., Clayton, J.: Chainer: a next-generation open source framework for deep learning. In: NIPS Workshops (2015)
47. T. report. Worldwide semiannual cognitive/artificial intelligence systems spending guide. In: International Data Corporation (2017)

48. Uijlings, J.R., Van De Sande, K.E., Gevers, T., Smeulders, A.W.: Selective search for object recognition. Int. J. Comput. Vis. **104**(2), 154–171 (2013)
49. Wang, Q., Guo, G.: Benchmarking deep learning techniques for face recognition. J. Vis. Commun. Image Represent. **65**, 102663 (2019)
50. Woolson, R.: Wilcoxon signed-rank test. Wiley encyclopedia of clinical trials, pp. 1–3 (2007)
51. Wu, Y., Liu, L., Pu, C., Cao, W., Sahin, S., Wei, W., Zhang, Q.: A comparative measurement study of deep learning as a service framework. IEEE Trans. Serv. Comput. (2019)
52. Xianyi, Z., Qian, W., Saar, W.: Openblas: An Optimized Blas Library. Agosto, Accedido (2016)
53. Yang, C.-T., Liu, J.-C., Chan, Y.-W., Kristiani, E., Kuo, C.-F.: Performance benchmarking of deep learning framework on intel xeon phi. J. Supercomput. 1–25 (2020)
54. Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Phanishayee, A., Schroeder, B., Pekhimenko, G.: Tbd: Benchmarking and analyzing deep neural network training. arXiv preprint arXiv:1803.06905 (2018)
55. Zou, S.-X., Chen, C.-Y., Wu, J.-L., Chou, C.-N., Tsao, C.-C., Tung, K.-C., Lin, T.-W., Sung, C.-L., Chang, E.Y.: Distributed training large-scale deep architectures. In: International Conference on Advanced Data Mining and Applications, pp. 18–32. Springer (2017)

Radwa Elshawi is a Senior Research Fellow in Big Data at Data Systems Group. Radwa received her Ph.D. in Information Technology from Sydney University, Australia in 2013. She received her B.Sc. and M.Sc. degree in Computer Engineering from the Computer Engineering department at Arab Academy for Science and Maritime Transport, Egypt, in 2005 and 2008 respectively. Radwa El Shawi's research interest are Computational Geometry, Big Data and Machine Learning.

Abdul Wahab is an M.Sc. Software Engineering student at the University of Tartu. He graduated with Cum Laude (Şeref Öğrencisi) standing from Bilkent University, Ankara, Turkey in 2018. Originally from Pakistan, he left his home for a better education abroad. Since High School, he has received various scholarships for his studies. Abdul majored in Electrical and Electronics Engineering with a concentration in Data Analytics, Statistical Learning, and Digital Signal Processing. He is passionate about analyzing various Engineering techniques especially in the domain of Data Analytics and Statistical Learning.

Ahmed Barnawi is currently a professor of information and communication technologies at Faculty of Computing and IT (FCIT) in King Abdulaziz University (KAU). He is the managing director of the KAU Cloud computing and Big Data Research group. He acquired his Ph.D. from the University of Bradford, UK, in 2005 and his MSC from UMIST (University of Manchester), UK, in 2001. Prof. Barnawi acted as an associate and Visiting Professors in Canada and Germany. Prof. Barnawi is an active researcher with good research fund awards track. His research interest include Big data, cloud computing, future generation mobile systems, advanced mobile robotic applications and IT infrastructure architecture. He published near to 100 papers in peer reviewed journals.

Sherif Sakr is the Head of Data Systems Group at the Institute of Computer Science, University of Tartu. He received his PhD degree in Computer and Information Science from Konstanz University, Germany in 2007. He received his B.Sc. and M.Sc. degrees in Computer Science from the Information Systems department at the Faculty of Computers and Information in Cairo University, Egypt, in 2000 and 2003 respectively. Prof. Sakr's research interest is data and information management in general, particularly in big data processing systems, big data analytics, data science and big data management in cloud computing platforms. Prof. Sakr has published more than 150 refereed research publications in international journals and conferences. One of his papers has been awarded the Outstanding Paper Excellence Award 2009 of Emerald Literati Network. He is also a winner of CAiSE'19 best paper award. In 2017, he has been appointed to serve as an ACM Distinguished Speaker and as an IEEE Distinguished Speaker. Prof. Sakr is serving as the Editor-in-Chief of the Springer Encyclopedia of Big Data Technologies. He is also serving as a Co-Chair for the European Big Data Value Association (BDVA) TF6-Data Technology Architectures Group.