Kennedy *et al. J Big Data* (2019) 6:16
https://doi.org/10.1186/s40537-019-0179-2

Journal of Big Data

METHODOLOGY Open Access

A parallel and distributed stochastic gradient descent implementation using commodity clusters

Robert K. L. Kennedy[1*], Taghi M. Khoshgoftaar[1], Flavio Villanustre[2] and Timothy Humphrey[2]

*Correspondence:
rkennedy@fau.edu
[1] Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431, USA
Full list of author information is available at the end of the article

Abstract

Deep Learning is an increasingly important subdomain of artificial intelligence, which benefits from training on Big Data. The size and complexity of the model combined with the size of the training dataset makes the training process very computationally and temporally expensive. Accelerating the training process of Deep Learning using cluster computers faces many challenges ranging from distributed optimizers to the large communication overhead specific to systems with off the shelf networking components. In this paper, we present a novel distributed and parallel implementation of stochastic gradient descent (SGD) on a distributed cluster of commodity computers. We use high-performance computing cluster (HPCC) systems as the underlying cluster environment for the implementation. We overview how the HPCC systems platform provides the environment for distributed and parallel Deep Learning, how it provides a facility to work with third party open source libraries such as TensorFlow, and detail our use of third-party libraries and HPCC functionality for implementation. We provide experimental results that validate our work and show that our implementation can scale with respect to both dataset size and the number of compute nodes in the cluster.

Keywords: Parallel stochastic gradient descent, Parallel and distributed processing, Deep learning, Big data, Neural network, Cluster computer, HPCC systems

Introduction

Training neural networks effectively and efficiently is an important component of Deep Learning. Large neural networks can consist of dozens, hundreds or even thousands of layers each with thousands of artificial neurons. Depending on the network's architecture, each of these neurons is connected to a large number of other neurons, where each connection has a trainable weight parameter that determines how the network responds to input signals. In the context of this paper, the effective training of these large complex networks is accomplished through the use of the computationally expensive process of backpropagation. Additionally, neural networks benefit from training on Big Data, as typically more data produces more performant models [1]. For example, the ImageNet database AlexNet was trained on roughly 1.2 million images, and at the time achieved state of the art results [2]. Problems of this magnitude are common and thus researching parallel network optimization on distributed and parallel systems is highly important.

Massively powerful computers are required to efficiently train large neural networks on large amounts of data. Thus, training neural networks in a parallel and distributed manner on large cluster computers or large shared memory supercomputers, is an active area of research.[1] As the models and datasets grow, it is possible to not only improve the hardware of the computer (such as more memory, faster CPU, faster GPU[2], etc.) but to also increase the number of computers simultaneously training a single model. Possible parallel methods for training a neural network include splitting the data across multiple computational nodes and splitting the neural network itself across multiple nodes, known as Data Parallelism and Model Parallelism, respectively. Each have their benefits, but both aim to reduce training time by spreading out the required computations across many different computers.

The distributed systems themselves are an important factor when trying to parallelize the neural network training [1, 3]. For example, a supercomputer can have thousands of individual processors that are connected via specialized high speed interconnects. In contrast, a cluster computer system is comprised of multiple individual computers, each with their own processor, memory, and disk and are connected together via high speed networking interfaces using standard networking protocols. Both of these systems require specialized software to handle the synchronization across nodes (CPU, GPU, or individual computers), the communication between nodes, the distribution of the data, and the distribution and synchronization of processes. In addition to the distributed system, an efficient optimizing algorithm is required to train models with a large number of parameters, as found in large neural networks. SGD, and many of its derivatives, is one such neural network optimizer. It is used extensively in machine learning in part because it is simple to implement and is fast when there are a lot of training instances, as found in Big Data datasets.

In this paper, we present a novel implementation of Parallel SGD on the HPCC systems platform, specifically, a distributed implementation of Parallel SGD. It is both a distributed and parallel approach in that it distributes data and executable code to separate computers (nodes in the cluster), as well as coordinates the training across all nodes towards a single objective. This is in contrast to a distributed only approach where individual nodes train toward different objectives. This would be useful for creating several models at the same time for statistical analysis but not when trying to output a single trained model. Previously, HPCC systems had only more traditional machine Learning algorithms, some Deep Learning algorithms that worked on a single node and a single algorithm that worked in a distributed fashion. Najafabadi et al. [4] proposed a distributed L-BFGS algorithm on HPCC systems but their approach was limited to the capabilities of HPCC systems. Since, Deep Learning is well suited for Big Data analytics [5] and HPCC excels at Big Data processing [6], our approach leverages the capabilities of both HPCC systems and different third-party Python libraries to train single Deep Learning networks using multiple nodes in parallel. HPCC systems is an open source,

[1] In this paper, we distinguish between the "parallel" and "distributed" when concurrent training of a single neural network by different computational resources occurs, and the latter denotes when the computational nodes are physically separated rather than multiple cores in a single physical processor.

[2] Central processing unit (CPU) refers to the main processor of a computer and graphics processing unit (GPU) refers to a discrete graphics card connected to the computer via a high speed bus.

parallel, cluster computing system designed to run on off the shelf commodity servers using standard networking protocols for connections between cluster nodes. HPCC provides its own distributed file system, its own parallel data processing language, called Enterprise Control Language (ecl), ECL compiler, and ECL IDE. Additionally, HPCC provides a facility to integrate with third-party languages, such as Python, which provides an effective and efficient environment for distributed Deep Learning by leveraging TensorFlow, a popular open source Deep Learning library.

The remainder of this paper is as follows. In "Methods" section, we discuss related work in the distribution and parallelization of neural network optimizers. "HPCC systems" section provides some background on the HPCC systems platform, a cluster system, as well as details its capabilities as it relates to the implementation of the parallel SGD algorithm. Next, we present details of the parallel neural network optimizer in "Parallel stochastic gradient descent" section. In "Implementation" section, we present our implementation of Parallel SGD on the HPCC systems Platform. "Results and discussion" section provides experimental results that evaluate and validate the implementation. Finally, in "Conclusion" section, we conclude our work and discuss possible avenues for future work.

Methods

Parallelization strategies

Training a neural network in parallel has two main factors that contribute to the training time. First is the communication cost between nodes and second is the computational time cost. Increasing the number of parallel nodes reduces the system's computational time but at the expense of high communication costs. A balance between the two factors results in decreased training time and is dependent on the architecture of the physical system as well as the methodology of splitting up the computations.

There are six different dimensions of parallelization for neural networks introduced by Nordstrom et al. [7], each with increasing communication costs. In the simplest dimension, training session parallelism, there exists virtually no communication costs. Each node in the system gets an entire copy of the dataset and model, each initialized with different weights. Each of the nodes trains a model and at the end of training, the model that performs the highest is selected. This technique, although rather trivial in concept, can be useful because training neural networks have the propensity to get stuck in local minima. The multiple, initialized models aim to widen the search to hopefully find a global minimum [8]. Another advantage to this approach is its parallelism is unbound, i.e. the increase in nodes and the decrease in training time (for that many different models) is nearly perfectly linear. The next dimension, training example parallelism, also known as Data Parallelism, splits the training data onto multiple nodes. Each node then computes on a smaller data size, reducing training time. This approach is examined in more detail in the coming sections and is the parallelization paradigm used in this paper's work. The final dimension that is suitable for a cluster computer environment is Node Parallelism, which is similar to model parallelism. In model parallelism, the neural network model is divided up and distributed across multiple nodes where each node only trains its portion of the model. For completeness, layer parallelism, weight parallelism, and bit parallelism are the remaining three dimensions. Node parallelism utilizes

pipelining to increase the throughput of the training instances being calculated through the network (either during forwardpropagation or backpropagation). Weight parallelism refers to the simultaneous calculations of all neurons in a given layer. Lastly, Bit parallelism is the bit level parallelization and Nordstorm et al. states it is often taken for granted. However, these three dimensions are outside the scope of this paper and are generally addressed, to some degree, by the optimizations in the neural network libraries used and the optimizations in the underlying computer operating system.

Data parallelism

In general, Data parallelism is the paradigm, during the training phase of creating a machine learning algorithm, where the data used for training is partitioned and distributed evenly across several nodes of a system, along with a copy of the initialized, untrained model, which is also distributed to each node or worker. All workers will concurrently train on their partition of the data and at the end of the training processes, the models are aggregated in some way to have the system produce a model that was trained on the entire dataset. This effectively reduces the amount of work required by each node by a factor of how many workers are used. This method is one of the oldest practical implementations of training an artificial neural network [9]. There are several different data parallel approaches, mainly differing by how the workers' contributions are aggregated. They can be categorized into two main groups: synchronous and asynchronous data parallelism.

Synchronous data parallelism

Synchronous data parallelism, as its name implies, is a training paradigm where the training steps are executed in a synchronous and sequential way. It is identifiable by a series of locking mechanisms during the training runtime. The synchrony ensures model consistency between training steps. Using mini-batch gradient descent as an example on a single machine, the dataset is partitioned into a series of mini-batches where each mini-batch is consumed in sequence until all the data has been consumed (see Algorithm 1). Then the weights for each mini-batch are averaged by summing all the mini-batch's gradients and dividing the total by the number of mini-batches. The locking mechanism in this case is the aggregation after each pass through the data and makes mini-batch SGD synchronous in nature. Even on a single machine, mini-batch SGD has been shown to optimize the training time [10].

Algorithm 1 Mini-batch Stochastic Gradient Descent with Backpropagation

```
for epochs do
    for t=0 to B/D do
        Sample B elements from D
        Compute Gradient with respect to B
        Update Network Weights
    end for
end for
```

A distributed adaptation of mini-batch SGD, where each node in a system computes on a single mini-batch, is also a synchronous approach. Each worker calculates its weight updates on its partition of data and a master node, or parameter server, aggregates the

weights. The locking mechanism comes from the fact that after each pass through of the data, the parameter server has to aggregate the weights before continuing onto the next iteration. If each worker in the system takes exactly the same time to train, the synchrony is not an issue. However, if one or more workers takes longer than the rest, the synchrony becomes a significant bottleneck in the training time. The workers can have varying training times for several reasons such as a cluster system with heterogeneous hardware, slightly different data partition sizes, network latencies, etc. Careful consideration needs to be taken to minimize the chances for this type of bottleneck. Furthermore, this approach requires parameter updates to happen on each mini-batch iteration. This results in large communication costs in a cluster computer.

One of the most challenging aspects of large scale distributed machine learning is the parallelization of neural network training when using SGD [11]. One key challenge is coordinating multiple computational nodes to optimize a single model toward a single goal without increasing overhead to the point of decreasing training time or model performance. There have been many studies that attempt to parallelize SGD [1, 7, 8, 12, 13]. Many of these are designed to use some form of a shared memory computer. Often referred to as a supercomputer, these systems have the benefit of tightly controlled communications between different processing units, either CPU's, CPU cores, or GPU's. However, this paper's work focuses on parallelization methods that are well suited for use on a cluster computer, the main difference being the communication between nodes is not a specialized serial bus, but rather a standard ethernet connection. This presents a high cost of communication due to the slower ethernet connections and standard networking protocols. The network itself introduces a bottleneck. Consider the scenario when all workers need to communicate their updates for aggregation concurrently. The network can become easily saturated, leading to increased training time. Careful consideration of the communication strategy of the parallelization is critical [14].

Parallel stochastic gradient descent

Designing a parallelization method that considers the high communication costs found in cluster computers has been extensively studied [8, 13, 15]. Parallel SGD, introduced by Zinkevich et al. [12] and shown in Algorithms 2 and 3, is one such technique and can be viewed as an improvement on model averaging. Model averaging convergence is dependent on the degree of convexity as a result of regularization. However, regularization decreases with increases in data size which makes it less useful in practice, especially since data sizes continue to grow. Parallel SGD improves upon this by combining the benefits of low network communication of model averaging and that of online learning [12, 16].

In practice, Parallel SGD is a Data Parallel method and is implemented as such. There are two different types of computers (or nodes) used in this optimizer, a parameter server and a worker node. The parameter server is where the Deep Learning model is defined, where the aggregation from the worker nodes occurs and is responsible for communicating the Deep Learning model and training data to the worker nodes. The worker nodes are responsible for training the model on the training data, communicating the changes to the model back to the parameter server and perform the bulk of the computations throughout the training process. A strict implementation of Parallel SGD

uses the SGD optimizer for the localized training. Our implementation uses HPCC systems to handle which computer is the parameter server and which computers are the worker nodes, as well as the communication between them. Specifically, the entire training data is divided into N parts, where N is the number or worker nodes in the cluster, and each part is sent to only one worker node. A single neural network model is then defined and copied to each of the nodes. Importantly, each worker node has a subset of the training data and each has an identical copy of the model. Then each of the worker nodes trains its model on its subset of the training data. This paper uses a mini-batch SGD for the localized training which is an efficient and popular derivative of SGD.

Once all workers are done with their training process, all nodes transmit, to the parameter server, the new model they just trained. Since, training a neural network model modifies just the weights between nodes, it is only the weight changes that are sent to the parameter server. Specifically, each node's weight updates are divided by the number of nodes, see Algorithm 3. Since the weights are aggregated only after all nodes processed their subset of the training data, the communication cost is constant and is kept to a minimum. The result is a new neural network model (incrementally trained) that is then redistributed to the worker nodes and the process then repeats itself for the desired number of iterations, called epochs. It is important to note that this aggregation and redistribution of the model is how this implementation utilized multiple computers to train a single neural network model in a coordinated way as well as how this implementation is a synchronous one . The time required to train is only dependent on the size of the data and the number of epochs selected for training which makes this an ideal method for systems with communication bottlenecks, such as in cluster computing.

Algorithm 2 SGD $(\{c^1, ..., c^m\}, T, \eta, w_0)$

for t=1 **to** T **do**
  Draw $j \in \{1...m\}$ uniformly at random
  $w_t \leftarrow w_{t-1} - \eta \partial_w c^j(w_{t-1})$
end for
return w_T

Algorithm 3 ParallelSGD $(\{c^1, ..., c^m\}, T, \eta, w_0, k)$

for all $i \in \{1, ..., k\}$ **parallel do**
  $v_i = SGD(\{c^1, ..., c^m\}, T, \eta, w_0)$ on each compute node
end for
Aggregate from all nodes $v = \frac{1}{k} \sum_{i=1}^{k} v_i$
return v

HPCC systems

HPCC systems is an open source software platform developed in 2000, and currently in use and maintained by LexisNexis, as a solution for data-intensive computing. HPCC is an integrated system environment that excels at extract, transform, load (ETL) operations, complex analytics, and provides efficient querying of large datasets by a large number of uses with its own data-centric parallel processing language called ECL [6]. It combines the benefits of MapReduce or Hadoop, its high-level languages, and other additions such as HBase and Hive. The system is comprised of newly developed system

software and middleware components that are layered on top of the Linux operating system, also open source. This combination provides an execution environment and distributed file system that is required by data-intensive computing.

Data-intensive computing is defined as software applications that are limited by I/O (input and output) or applications that need to process large amounts of data [17, 18]. These types of applications spend a large percentage of time processing and moving around data; in this case, between nodes in a cluster. A distributed implementation of neural network training would fall under this definition as it needs to both move around data between nodes and process large data. HPCC systems was developed to be a data-intensive computing system. It collocates the data and programs, on each node, to reduce the movement of the data which increases performance and parallelism [6, 18]. It uses a high-level language abstraction (ECL) to make the system machine independent which allows for deployment on different types of systems [19]. As a result, the underlying hardware on which an HPCC system is deployed can be scaled linearly as the data and processing requirements grow [6]. HPCC is designed to run on commodity computing[3] clusters (cluster computer) with each node running a Linux operating system. A cluster computer is a group of individual computers, each connected via a network. This is in contrast to a supercomputer which for the purposes of this paper is a system with multiple processing units using specialized hardware to allow for random-access memory (RAM) memory to be shared. A benefit to the cluster computer is it is relatively easy to add another computer to the system. In a super computer, low-level hardware considerations are needed for adding another computational unit, whereas a cluster computer simply needs another node added to the cluster network. It could prove difficult in some cluster systems to integrate the additional node, but HPCC systems makes it trivial to add another node to the system, provided it is network reachable. This is especially easy and advantageous when using cloud computing, such as Amazon Web Services (AWS) or Microsoft Azure. An additional computer instance simply needs to be created and turned on [20–22] and HPCC easily adds it to the cluster.

The HPCC systems platform excels at both ETL tasks and analytics using its own programming language, ECL. ETL operations are used to read data from external sources, cleaning and pre-processing the data so it is in a consistent format for use in a system, and loading that data into the internal database [6]. ECL adheres to the Dataflow paradigm and similar to many other approaches, HPCC is designed to use a cluster of commodity computers. HPCC systems consists of two cluster types, a rapid data delivery system known as Roxie, and a data refinery known as Thor. Roxie [6] is used for high throughput delivery of data that is prepared by Thor and is out of scope of this paper.

The Thor cluster [6] uses a master/slave design with a single master Thor process and multiple slave Thor processes. HPCC systems is designed to run a cluster of Linux based computers which can be configured to have either single Thor processes per physical node, or multiple Thor processes per physical node and can scale to thousands of processes across thousands of physical nodes[4], as seen in Fig. 1. A Thor cluster uses its own

[3] Commodity computing is defined as a cluster computing system comprised of individual, relatively cheap and easy to obtain computers connected with standard networking protocols.

[4] For the purpose of this paper, the configured system has one Thor process per physical node and the term node is used interchangeably with the term process and worker.

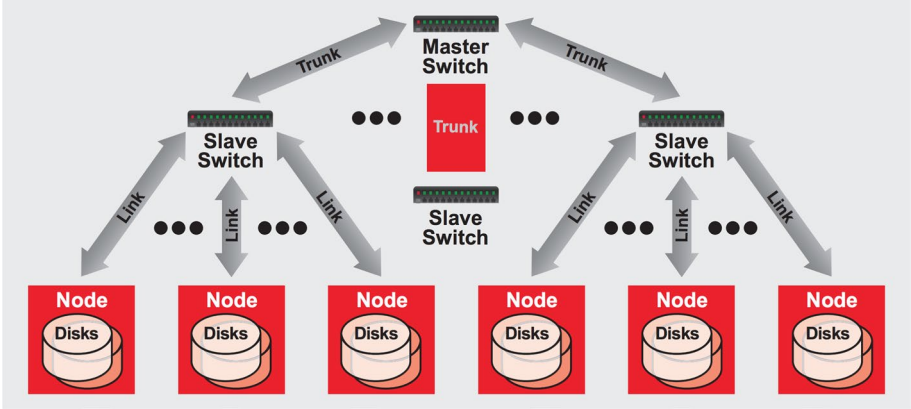

Fig. 1 Visualization of an example HPCC systems cluster

distributed file system (DFS). It can include data from many different types of sources and different formats such as, CSV, XML, numeric, strings, and even Binary Large Object (BLOB). The data is in a record format which is different than the block type found in a MapReduce based system. A dataset consists of multiple records, each of which can be different value types and can be variable in length as well as having nested records. When a dataset is uploaded to a Thor cluster, it is split across all of its processes no matter how big the data. HPCC systems automatically handles exactly how the data is partitioned without user input, meaning it will do its best to keep the distribution, of the data partition sizes, across all nodes the same; i.e. a 400-node system would have the dataset distributed across all nodes into 400 equal or nearly equal parts. It uses the nodes' operating system for physical file storage and does not separate individual records. The master Thor node is responsible for monitoring each slave's processes, handles the network I/O for distributing data and distributes ECL code (which is compiled into C++ for execution) to each slave node for execution.

ECL language

HPCC systems uses its own programming language called ECL [6]. ECL is an implicitly parallel declarative language that compiles into C++ and is based on the Dataflow model. As with all declarative languages, execution is not determined by the order of the statements, but rather the sequence of operations on the data defined by the Dataflow, as described in previous sections. The high-level nature of the language lends itself to high reusability and extensibility and enables users to efficiently generate complex runtimes. HPCC systems includes an Integrated Development Environment (IDE) for ECL and an optimized compiler that compiles ECL into C++, optimized for distributed parallelism across the Thor nodes. As a result, ECL is easily extended by any standard C++ libraries or .DLL's.

ECL has extensive capabilities for data management, filtering, transformation, and has built in functions that facilitate user defined transformations functions that transform records. Some of the built-in functions include: PROJECT, ITERATE, ROLLUP, JOIN, COMBINE, FETCH, NORMALIZE, DENORMALIZE, and PROCESS. In addition, users can embed C++ code directly in line with ECL. Users

are also able to embed other languages, through HPCC plugins, such as R, JavaScript, and Python, and any of their available libraries. This paper's main contributions rely on the combination of ECL and parallel embeddings of other languages and libraries. The Thor cluster allows for these ECL capabilities to function on all of the records distributed across the system or to function on each data partition on each node locally. This is either explicitly defined by the user or implicitly defined by which function is used. This differentiates HPCC from MapReduce where in a MapReduce setting each operation is done locally only. For example, a local sort in HPCC would sort the records on each node individually, without regard to other parts of the dataset on other nodes, a global sort would be performed on the entire distributed dataset and HPCC systems handles all the data transfers between the nodes to accomplish the sort.

TensorFlow

TensorFlow is an open source programming library created by Google [23]. Like ECL, TensorFlow implements the dataflow model that represents programs as directed graphs. Fundamentally, TensorFlow is a symbolic math library that is primarily used for building and training artificial neural networks. It is not only used for development and training of neural network models but also the deployment of the models for practical use.

The Google Brain team created TensorFlow to solve their need for very large-scale models. Its predecessor is DistBelief [1], a software library used for training neural networks internally at Google. DistBelief was designed for large scale distributed training of neural networks which had two algorithms that were asynchronous in nature and were well suited for a shared memory system. DistBelief, and now TensorFlow, has been used at Google for a wide range of research problems such as unsupervised learning, language representation, image recognition, speech recognition, models for playing Go [24], and reinforcement learning among others [25–30]. They have deployed models for practical use in widely used Google products like YouTube, Google Maps, and others [23]. TensorFlow is the second generation machine learning library and it improved upon DistBelief by being more flexible, more performant, and being able to train a wider gamut of models all while being used on heterogeneous systems. TensorFlow is not only the standard machine learning library in use at Google, it is also one of the most popular Deep Learning libraries in the community [31]. Thus, it is the machine learning library used in our work.

TensorFlow is a Python library, written in Python and C, and can be imported and used as any other Python library. A TensorFlow example is provided below for completeness. This TensorFlow logic defines a Multi-Layer Perceptron (MLP) [32] used for classifying the MNIST dataset.

```
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, n_input])
Y = tf.placeholder("float", [None, n_classes])

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'
    ])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
logits = multilayer_perceptron(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
# Initializing the variables
init = tf.global_variables_initializer()
```

Code Example 1 TensorFlow MLP

Implementation

The coding contribution of this paper is twofold. First are the contributions written in Python and second are the ECL contributions that follow a parallelism paradigm, in this case, *Data Parallelism*. When HPCC distributes the data to a worker node, that data is then processed by the embedded Python code. However, initially the data is formatted for HPCC systems in an ECL Record, detailed in "ECL language" section. In order for the Python code to process it, there needs to be a set of functions that interprets between the ECL Record and the python data types (in the embedded Python code) and vice versa. Similarly, since HPCC handles the communication between the nodes, the neural network model itself needs to be interpreted between the embedded Python code and ECL. Additionally, any function parameters and any meta-data that needs to be passed between nodes also needs to be handled by a Python-ECL interpreter. We created, along with the ECL counterparts, various functions to handle these interpretations.

Python

At the beginning of the training processes a neural network needs to be created. We use Python libraries Keras and TensorFlow to define our neural network architecture, create our loss functions, and to train the models. Our implementation is designed to be flexible and extensible and can support any architecture supported by either TensorFlow or Keras. This includes neural networks with various types of dropout layers, fully connected layers, and convolutional layers that would be used to design a neural network architecture. Once a neural network architecture is decided upon, it needs to be represented in code along with the desired optimizer and loss functions. Defining an MLP [32–34] in Keras is shown below. It is important to note the difference between defining a model in Keras and in TensorFlow, with the former being considerably fewer lines of code and consequently, quicker.

```
model = Sequential()
model.add(Dense(100, activation='relu', input_shape=(123,)))
model.add(Dropout(0.5))
model.add(Dense(2, activation='sigmoid'))
model.compile(loss=losses.binary_crossentropy,optimizer=Adam(epsilon
    =1e-08),metrics=['accuracy'])
```

Code Example 2 Defining a Neural Network with Keras

Once the neural network has been defined, it needs to be trained. Following a Parallel SGD approach, the initialized model needs to be distributed to each worker node along with its partition of training data. The embedded Python code, that runs on each node, accepts as parameters the training data[5], model, and meta-data. At the start of the training process, the neural network model will be initialized with random weights. After each iteration of the training process (called epochs), the model becomes further trained. The first interpreter enables the embedded Python code to receive the training data. The training data is stored locally on each node, after being distributed by HPCC, in an ECL Record format with at least one attribute that corresponds to the y value and some arbitrary number of values that corresponds to the x value, or the features which to learn from. Recall that the y value is the variable, or category, that the model is trying to learn. For example, if the model is being trained on images of hand written digits, the y value would be the numeric digit that is visually represented by the image and the x value would be the image in some encoded format. In the case of MNIST (hand written digits database [35]) it is the individual grey scale pixel values represented as an integer between 0 and 255. The interpreter then takes the data passed in by the HPCC Python plugin and converts it into a usable format for training using Keras functions. The final step is to then locally cache this data to disk using the highly optimized HDF5 format [36]. Using this specialized file format to cache data between epochs drastically reduced increased performance between epochs.

[5] The first iteration receives the training data, and every iteration after that uses a locally cached partition of data.

```
model = Sequential.from_config(...) #pass model configuration from
    ECL
model.compile(loss=losses.binary_crossentropy,optimizer=Adam(epsilon
    =1e-08),metrics=['accuracy'])
model.set_weights(...) #sets model weights in preparation of
    training
model.fit(x_dat, y_dat, batch_size=32, epochs=1, shuffle=True,
    verbose=0, validation_split=0.2)
```

Code Example 3 Worker Training

On each worker, as seen in Example 3, the model needs to be defined identically to other workers, then the weights need to be set. The weights will be either the initialized weights, or weights of the partially trained model. In the case of Parallel SGD, all workers start with the same weights. The weights are then returned after training as shown in the last line.

Once the data is in a format consumable by Keras, the model training can start. In Keras, the process of training the neural network is very flexible. Various optimizer settings, such as batch size, number of epochs, logging, among others, are passed into the Keras function by the meta-data handler. The implementation of this handler was designed to be flexible and extensible to be able to handle a wide variety of model types and training scenarios. The Keras function then starts the TensorFlow training on the TensorFlow model created by the Keras Application Programming Interface (API). After the training process, Keras is used to get the model weight updates from the newly trained model. The workers return the updated model weights in the same format as it received it, so it can run in a recursive fashion.

```
updatedWeights = model.get_weights()
return (...) #returns to ECL an object with updatedWeights
```

Code Example 4 shows how the weights are simply returned from the embedded Python to ECL for aggregation.

At this point, all nodes have completed training and the weights need to be aggregated. The way the weights are aggregated is dependent on which parallelization scheme the user defines. The scheme can have the model recursively trained for a number of more epochs, or its performance can be evaluated on unseen test data. To properly evaluate a model, it must be used to make predictions on unseen test data. Unseen test data is simply data that has not been directly used in training the model. A performance metric, depending on the nature of the model, is used to show how well the neural network can generalize. The evaluation simply predicts a class based on the features in the test data and compares them against the ground truth of that data point. Consequently, since this is essentially using the model, at this point the model can easily be saved to disk for later deployment and consumption by HPCC systems.

ECL

Our implementation uses ECL code to partition and distribute the training data to each node, the distribution of the embedded Python code, the execution of the Python code on each node, and the communication of the model between the nodes.[6] As discussed previously, there needs to be an ECL-Pyembed handler to bridge the gap between the two different languages. Thus far, we have only discussed the Python side of the implementation.

We employ the data parallel paradigm for distributed neural network training. The entire dataset is partitioned into smaller sets and is distributed to each node. The number of partitions is dependent on two factors, an individual node's memory and the number of nodes in the system. The dataset is first divided by the number of nodes so that each of the N nodes gets 1/N of the data. If this 1/N of the whole dataset can fit into memory, only N partitions are created. If the underlying hardware results in a memory constraint, further partitions are created on each node. For example, if we have a 10 node cluster, dataset with 100,000 rows, and each node can only fit 5000 rows in memory, there will be 20 data partitions where each node gets 2 of the partitions. This partitioning is done dynamically and provides flexibility when choosing datasets, training tasks, and hardware. Each partition, or partitions, is then communicated from the master node to the slave nodes, adhering to Data Parallelism.

Along with the data partitions, ECL transmits the pyembed code to each node for processing. Once the pyembed code processes its partition, or multiple partitions if memory is limited, the neural network updates are aggregated from the workers to the master node, in the form of an ECL record. Sample ECL code is detailed below. It is important to note that the following code is written only once, and HPCC distributes the code as needed to any necessary Thor nodes.

```
mnist_data_type := RECORD
 INTEGER1 label;
 DATA784 image;
END;
```

Code Example 5 HPCC Record Definition

[6] HPCC systems provides a facility, named pyembed, that allows for the use of embedded Python code.

```
trainingData := (DATASET('~mnist::train',mnist_data_type, THOR);
dTrainingData := DISTRIBUTE(trainingData);

Marked := PROJECT(dTrainingData, TRANSFORM(nodeMarked, SELF.node :=
    Std.system.Thorlib.Node()+1, SELF:=LEFT), LOCAL);
GroupedData := GROUP(Marked, node, LOCAL);

subMarked := PROJECT(GroupedData, TRANSFORM(smallGroup, SELF.grp:=
    COUNTER DIV M, SELF:=LEFT), LOCAL);
subGroup := GROUP(subMarked, grp, LOCAL);
```

Code Example 6 Data and Code Distribution

```
weightUpdates := ROLLUP(subGroup, GROUP, runPy(model, ROWS(LEFT),
    LEFT.grp, 0, FALSE));
```

Code Example 7 Distributed Parallel Processing

Example 5 denotes an ECL record type definition. A record of this type would have some number of rows, for each image in the database, each with two attributes: a one byte integer representing the image class, and a block of hexadecimals 784 bytes long, representing each of the 784 grey-scale pixels in the 28×28 MNIST image.

Example 6 illustrates how ECL can be used to distribute and partition the data and pyembed code from the Thor master node to the Thor slave nodes. The first lines define an ECL dataset from the MNIST file and distributes it across the nodes. Remember that ECL is a declarative language, not an imperative language, so the actual distribution of the data might not happen at this particular point. The next group of code partitions the dataset into N partitions, where N is the number of nodes. The last group of code partitions the data into groups of M, where M is chosen by the programmer to avoid memory constraints.

Example 7 takes the actions denoted in Examples 5 and 6, and performs the distribution and aggregation as outlined by the Data Parallelism paradigm. It can be read as *ROLLUP* executes *runPy* locally, on each node in the cluster, and waits to receive weight updates from all of the nodes before populating *weightUpdates* (which is an ECL record type). *weightUpdates* would then be further processed according to the parallel optimization scheme.

Results and discussion

Medicare Part B dataset

The dataset used in our case studies is a Medicare Part B fraud dataset. It consists of Medicare Part B claims and is used for finding fraudulent physicians. Medicare Part B is a U.S. government program that was created to cover some costs associated with medical procedures, primarily for people age 65 years and older. The dataset is a labeled dataset containing Medicare claims each identified as a fraudulent Medicare claim (by the physician) or not. In general, the claims process is as follows. A physician will perform one or more medical procedures on a patient and the physician then submits a claim to Medicare, rather than directly to the patient, for payment. Additional information can

be found in [37–39]. The original data was released by the Centers for Medicare and Medicaid Services [40] as a result of a new policy by the U.S. Department of Health and Human Services [41] in an attempt to identify Medicare fraud, waste, and abuse. While the original 2015 Medicare Part B data is publicly available, it has been reworked to a high degree by members of our research group, see Bauder et al. [42, 43]. Thus, the dataset used in this paper's experiments is proprietary in nature. It consists of over 3.3 million records each with 29 attributes that are then one-hot encoded for use in our neural networks. There are 15 attributes that correlate to the physicians themselves, such as ID number, name, gender, state, and what type of physician they are. In addition, there are 14 more attributes that correlate to the claim's procedure, such as where the procedure was performed, medical procedure codes, and different statistical metrics for the Medicare payments. These include, average of the charges that the provider submitted for the service, standard deviation of the Medicare payment amount (the difference from the average cost of this procedure), and others [44].

Each record is roughly 850 bytes long in memory, making this a 2.8 gigabyte file. The dataset is then divided into several different size datasets for experimental validation. For example, one of the derived datasets has a 1:1 class ratio imbalance. This equates to roughly 2240 records with a total of roughly 2 megabytes in memory. This is done to generate multiple different datasets of varying size to measure the implementation's scalability, measured in training time, with respect to dataset size and cluster size. A sufficiently large dataset is required to see training time reductions due to the increased communication cost of large clusters. The actual size in bytes to be considered "sufficient" is dependant on the cluster's underlying hardware. Class imbalance has been shown to present unique challenges and negative effects to model performance [45–48].

Random under-sampling (RUS)

The datasets used in this paper's results are derived from the Medicare Part B dataset, see "Medicare part B dataset" section. We created 10 different size datasets by performing random under-sampling (RUS) on the entire Medicare Part B dataset. In this dataset, there are roughly 1400 instances belonging to the minority class, and the rest belong to the majority class. During RUS, a random selection of the majority class is combined with the entire majority class to create a new dataset with a specified class imbalance ratio. For example, if the desired class ratio is 1:1, all of the minority class is combined with a random 1400 instances from the majority class, to produce a dataset total of 2800 instances (1400 minority and 1400 majority). Importantly, during RUS, the distribution of all other attributes is maintained between the original and the newly RUS-created dataset, with respect to the majority class. The minority class is not modified during this procedure. The random under-sampling technique has been shown to improve model classification performance, as compared to training on the entire dataset without RUS [47, 49].

Training time scalability

In this case study, we seek to observe how the training time scales with respect to data size and cluster size. We conduct 50 different experiments of varying data sizes on different sized clusters each with 20 trials for statistical analysis. For each experiment, and

on each cluster size, we train an identical MLP model on increasing dataset sizes and compare the training time of each. As the dataset size increases, we expect to have the training time increase. Additionally, we expect to see an inverse relationship between training time and cluster size.

In addition to any limitations of the individual algorithms, systems, or frameworks, the combination poses the possibility of introducing additional limitations. One such limitation is with memory size. Since the training computations are being executed by a Python interpreter, controlled by HPCC, and the code itself is run through the pyembed, there exists additional memory constraints when training a model as compared to training outside of the HPCC environment. This was considered in the implementation, and a setting is provided to the user that would be dependent on the underlying hardware, neural network being trained, and the data, to overcome the systems shortcomings. Additionally, this adds to the robustness of the implementation since it can easily be adapted to different hardware, different data sizes, and different neural network architectures. However, due to these limitations, we expect our implementation to slow with increases in data sizes up until a node's partition of data is too large for the pyembed memory allocation.

We use the Medicare Part B dataset with Random Undersampling (RUS) to generate our different dataset sizes. First, the whole dataset is divided into two partitions with 80% for training and 20% for testing. RUS is performed on the training partition to generate the training datasets of varying sizes. Each experiment uses the same testing dataset size with an even class ratio (e.g. 1:1 ratio). A RUS ratio is used and is in the form of Minority class to Majority class. Each generated dataset uses the same number of instances from the minority class (1120 instances). For example, the dataset 1:25 would have 1120 minority instances and 28,000 majority instances ($1120 * 25 = 28{,}000$) for a total dataset size of 29,120. Our generated datasets used in this study range from 1:1 (2240 total instances) to 1:2000 (2,241,120 total instances).

In Fig. 2, it can be observed that both the training data size and number of nodes have a large impact on the training time. For a given cluster size, the training time increases with data size. Inversely, for a given dataset size, training time decreases as cluster size increases. Our implementation allows the user to reduce the necessary training time of a model by simply adding nodes to the system. However, it does follow Amdahl's Law[7] [50] in that there is diminishing returns in training time as nodes are added to the cluster. So the benefits of additional nodes would only be realized with sufficient increases in data size. Specifically, a larger cluster should not be slower than a smaller one with enough data.

Consider the scenario where there is either a small training data size or it is not sufficiently large for a certain number of nodes. Additional nodes will in fact increase training time by a constant. Figure 3 illustrates this. Looking at the smallest dataset size, clusters with 8 and 16 nodes are orders of magnitude slower than the other cluster sizes. Interestingly, their training time is constant for several of the smaller data sizes. The dataset size that is large enough to increase training time, for a given node

[7] Amdahl's Law is a formula that defines the maximum theoretical speedup in execution of a fixed task with increases in performance of the computer.

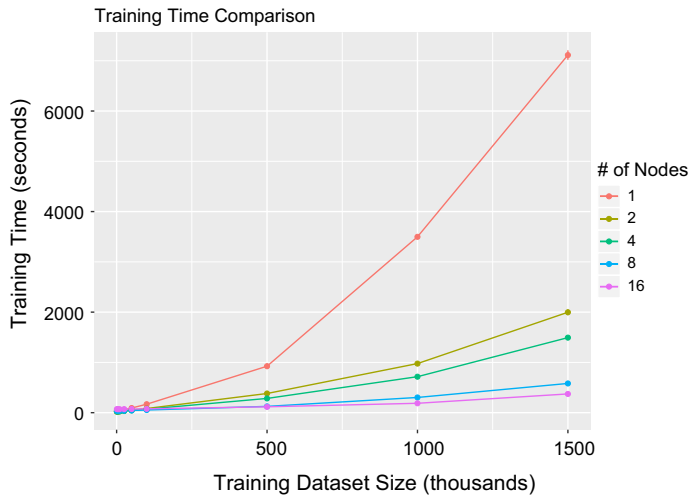

Fig. 2 Visualization of training time with respect to training data size in thousands. Each line represents results from a cluster with N number of nodes. Note that the *X* and *Y* scale are linear

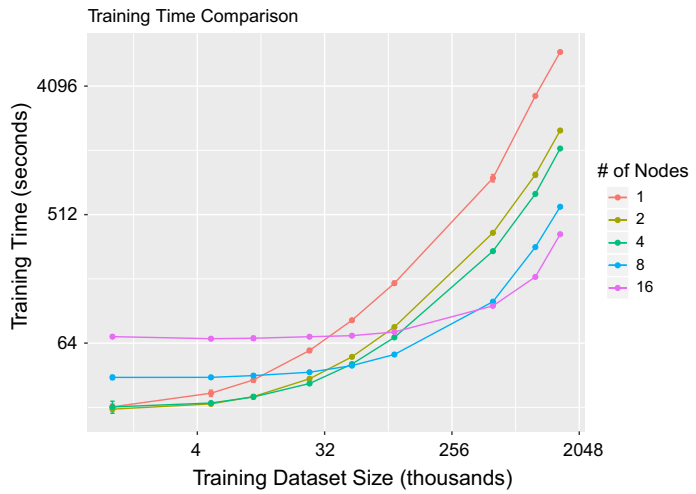

Fig. 3 Visualization of training time with respect to training data size in thousands. Each line represents results from a cluster with N number of nodes. Note that the *X* and *Y* scale are logarithmic, and clearly shows the constant overhead of the larger clusters on small datasets

count, is the same size at which we start to see a benefit of using a cluster of that size. This very clearly shows the situation when the constant communication cost outweighs the benefits of the increased processing power, see "Parallelization strategies" section. It is important to note the *X* scale of Fig. 3 is logarithmic. This provides a second perspective on the results, as well as clearly illustrates the constant communication overhead for each increase in cluster size.

It is interesting to note the difference between the training time vs. cluster size curve between the datasets with 1:2000 and 1:1500 class ratio. Figure 3 illustrates that there is very little, if any, difference between the two training time curves. Comparing any other two sequential datasets (i.e. any two datasets that are closest in size) there is

a significant change in training time when the training size is increased. This is due to the relationship between the physical hardware specifications and the size of the dataset on disk in this paper. Recall that the implementation allows for larger than memory datasets to be trained. To accomplish this, the dataset is partitioned into smaller segments and sequentially trained on by a given node. The small percentage difference between the 1:1500 dataset and the 1:2000, as compared to the percentage differences between other pairs, combined with the fact that on every experiment, those two datasets ended up being on the same size partitions are why their training time curves are so similar. Additionally, the increased communication time, with increased cluster size, can be observed by the upward sloping line segments correlating to the smaller dataset sizes, see Fig. 3. Only after a certain size will the time curve continually slope downward; in this figure, the dataset with 1:500 class imbalance.

Statistical analysis

We grouped the experiments into three distinct groups. The first group presents a dataset with a 1:1 class ratio. This group shows how the implementation performs on a small dataset. The second group presents the datasets with 1:5 and 1:10 class ratios. These two datasets are grouped since they are the smallest datasets in which the addition of worker nodes proves beneficial. The final group presents the datasets with 1:25, 1:50, 1:100, 1:500, 1:1000, 1:1500, and 1:2000 class ratios. These are grouped because each successive dataset has the same trend of reduced communication costs and increased utility of additional nodes as the previous one. The trends are presented in a series of box plots in Additional file 1: Figures 6 and 7. We use Analysis of Variance (ANOVA) [51] and Tukey's Honestly Significant Difference (HSD) tests [52] to determine if dataset size and number of training nodes in the cluster significantly impact model training time [53]. All of the experiments are included in the ANOVA tables and HSD tables. In the fist subsection, "Dataset 1:1 class ratio analysis", we include the ANOVA and HSD tables inline. For the remainder of this paper, all remaining ANOVA and HSD tables are found in Additional file 1.

Dataset 1:1 class ratio analysis

First, we examine the case where the dataset has a 1:1 class ratio, the smallest size used in our paper. This dataset has 1120 majority records and 1120 minority records. We discussed previously that a dataset needs to be sufficiently large to benefit from more nodes and can even run slower if there are too many parallel nodes. In our case study, clusters with 1, 2, and 4 worker nodes all have statistically similar training times for the dataset 1:1. We plot the results of training time vs node count in a box plot format (such as in Fig. 4). Each box depicts the 20 trials for each experiment. Outliers are denoted by single points, and the box itself denotes the mean, median, and 25th and 75th percentiles. The box plot is testing for the significance of the number of nodes in a cluster with respect to training time of a given model trained on a given dataset. Boxes that are similar in Y values denote no significant difference in training time between those two cluster sizes. Boxes that have different Y values, have significant difference in training time between those two cluster sizes. The results are illustrated here, and are numerically analyzed below.

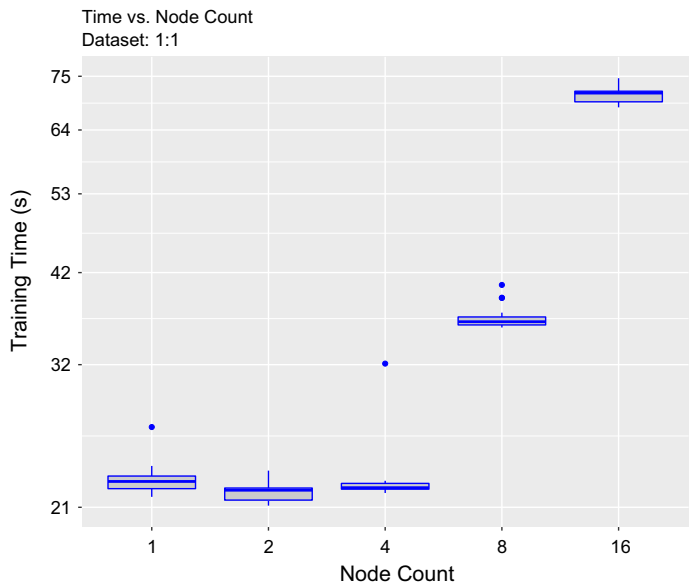

Fig. 4 Depicts a box plot comparing the training time vs. node count for the dataset size 1:1

Table 1 ANOVA Table showing the number of nodes in a cluster is a statistically significant factor regarding training time training time on the 1:1 dataset size

	Df	Sum Sq	Mean Sq	F value	Pr (> F)
Nodes	4	35,419.46	8854.87	3968.24	0.0000
Residuals	95	211.99	2.23		

Table 2 Tukey's HSD test with groupings

	Training time	Groups
16	71.06	a
8	36.74	b
1	22.82	c
4	22.78	c
2	21.98	c

The slowest is the 16 node cluster, second slowest is the 8 node cluster, and all clusters in group "C" have no difference in training time

Training time vs. # nodes, 1:1

As shown in Fig. 4, the cluster with 8 nodes is statistically and significantly slower than the smaller clusters, and the 16 node cluster is slower still. The results are tested with an ANOVA and are presented in Table 1 showing that the number of nodes has a significant effect on training time. In addition to the ANOVA values, we present the Tukey's Honestly Significant Difference test (HSD) [52] to compare and rank the factors. Tables 1 and 2 show that our assumptions from the visualizations in Figs. 3 and 4 are valid.

Datasets: 1:5 and 1:10 class ratio analysis

Next, we examine the case where the dataset size is 1:5 and 1:10, the next smallest data sizes. Recall that the 1:5 dataset is 6720 records long and the 1:10 is 12,320 records long, both relatively small training set sizes. However, the results show that these are the first sufficiently large datasets to benefit from an increase in nodes. Additional file 1: Figure 5 shows a drop in training time when comparing a 1 node cluster to a 2 node cluster, for the 1:5 dataset (a) and the 1:10 (b) dataset, respectively. Similar to the previous dataset size, 1:1, the clusters with larger numbers of nodes are showing a significant increase in training time due to the communication costs. Additional file 1: Tables 3 (1:5) and 5 (1:10) again show the number of nodes is a significant factor in training time. Additional file 1: Tables 4 and 6 show a single node cluster or clusters with 16 or 8 nodes take significantly longer to train on both the 1:5 dataset and the 1:10 dataset. In this case, the 4 node and 2 node clusters perform similarly, for each dataset. Thus, training times would be sped up by adding nodes, with similar results, for both datasets.

Datasets: 1:25, 1:50, 1:100, 1:500, 1:1000, 1:1500, and 1:2000 class ratio analysis

Next, we examine the rest of the datasets (1:25, 50, 100, 500, 1000, 1500, 2000), ranging from a size of 29,120 records to a size of 2,241,120 records. The previous trend of increasing utility per additional node as the dataset grows, continues with these larger datasets. Additional file 1: Figures 6(a) and (b) for datasets 1:25 and 1:50 show that the larger clusters sizes' utility are steadily increasing, though still negative (i.e. larger clusters are slower). Referring to Additional file 1: Table 10 and Figure 6(b), a dataset of at least 57,120 is needed before an 8-node system is worth considering. Though training time for an 8-node and a 4-node cluster are grouped together, indicating they train in the same amount of time, any larger dataset than this would train significantly faster with a cluster of at least 8 nodes, see Additional file 1: Tables 12 and 14. Finally, it can be observed from Additional file 1: Table 16 and Figure 7(a) that a training dataset of 1,121,120, or greater is sufficiently large to warrant the use of a cluster with at least 16 nodes.

Conclusion

Training state-of-the-art neural networks require very large datasets and large amounts of computing power. As the dataset sizes increase and deep neural networks grow in complexity, so too do the computational and memory demands. One approach to solve this is to train a model across multiple computers in a distributed and parallel fashion. By dividing up the required computations across a cluster of computers, the overall training time can be reduced. We implemented Parallel SGD on HPCC systems to achieve this. We combined the popular neural network training library TensorFlow with the powerful distributed cluster system, HPCC, to achieve a synchronous data parallel SGD method for training neural networks.

We examined how our proposed implementation reduces the required training time for a neural network with respect to dataset size and cluster size. We show the training time with respect to each additional node in a cluster is dependent on both its addition of computational power and its addition of communication overhead. Thus, the training time is highly dependent on both the data size and cluster size. We

presented the balance between cluster size and dataset size to minimize the training time using Medicare Part B data that clearly illustrated sufficiently larger datasets are required for increases in cluster size. The implementation is effective across a wide range of dataset sizes and is capable of scaling across several cluster sizes. However, since the implementation is highly dependent on the underlying cluster hardware, the size of the cluster must be carefully decided based upon the size of the training data to achieve optimal training time reductions.

The implementation presented in this paper provides the basis for future research and development of distributed parallel neural network training on HPCC systems. Our implementation is currently a synchronous, data parallel method. Additionally, our results are validated using one type of neural network architecture and data from one domain. One possible avenue of future work is to research, design and implement a Model Parallel method of distributed training on HPCC systems. Model Parallel methods are advantageous when the model architectures become huge, specifically, when it is larger than the memory resources of individual computational nodes. Second, a combination of Model and Data parallelism, a hybrid approach that uses different paradigms during specific training phases, would benefit HPCC systems by allowing it to train both very large models on very large datasets on commodity hardware. Lastly, implementation of complementary Data Parallel optimizers on HPCC systems would be worthwhile, specifically, implementation of various asynchronous optimizers. Though these methods present new complexities, such as stale parameters and an increased communication overhead, implementations of asynchronous and synchronous optimizers on HPCC systems would provide larger opportunities for future research in Deep Learning.

Additional file

Additional file 1. Additional figures and tables.

Abbreviations

API: Application programming interface; BLOB: Binary large object; CPU: central processing unit; ECL: Enterprise Control Language; ETL: extract, transform, load; GPU: graphics processing unit; HPCC: high-performance computing cluster; IDE: integrated development environment; MLP: multi-layer perceptron; RAM: random-access memory; SGD: stochastic gradient descent.

Authors' contributions

RKLK carried out the conception and design of the research, performed the implementation, performed the evaluation and validation and drafted the manuscript. TMK, FV, and TH provided reviews on the manuscript. TH provided expert advice on ECL and deployment of HPCC systems. All authors read and approved the final manuscript.

Author details

[1] Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431, USA. [2] LexisNexis Business Information Solutions, 245 Peachtree Center Avenue, Atlanta, GA 30303, USA.

Acknowledgements

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Not applicable.

Funding
Not applicable.

Publisher's Note

Received: 3 December 2018 Accepted: 28 January 2019
Published online: 14 February 2019

References

1. Dean J, Corrado GS, Monga R, Chen K, Devin M, Le QV, Mao MZ, Ranzato M, Senior A, Tucker P, Yang K, Ng AY. Large scale distributed deep networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ, editors. Advances in neural information processing systems 25. Curran Associates, Inc.; 2012. p. 1223–31. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.
2. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition; 2014. arXiv preprint. arXiv:1409.1556.
3. Coates A, Huval B, Wang T, Wu D, Catanzaro B, Andrew N. Deep learning with cots hpc systems. In: International conference on machine learning; 2013. p. 1337–45.
4. Najafabadi MM, Khoshgoftaar TM, Villanustre F, Holt J. Large-scale distributed l-bfgs. J Big Data. 2017;4(1):22. https://doi.org/10.1186/s40537-017-0084-5.
5. Najafabadi MM, Villanustre F, Khoshgoftaar TM, Seliya N, Wald R, Muharemagic E. Deep learning applications and challenges in big data analytics. J Big Data. 2015;2(1):1. https://doi.org/10.1186/s40537-014-0007-7.
6. Middleton AM, Chala A. Hpcc systems: introduction to hpcc (high-performance computing cluster). White paper, LexisNexis Risk Solutions; 2011. http://cdn.hpccsystems.com/whitepapers/wp_introduction_HPCC.pdf.
7. Nordstrom T, Svensson B. Using and designing massively parallel computers for artificial neural networks. J Parallel Distrib Comput. 1992;14(3):260–85.
8. Pethick M, Liddle M, Werstein P, Huang Z. Parallelization of a backpropagation neural network on a cluster computer. In: International conference on parallel and distributed computing and systems (PDCS 2003); 2003.
9. Zhang X, Mckenna M, Mesirov JP, Waltz DL. An efficient implementation of the back-propagation algorithm on the connection machine cm-2. In: Advances in neural information processing systems; 1990. p. 801–9.
10. Le QV, Ngiam J, Coates A, Lahiri A, Prochnow B, Ng AY. On optimization methods for deep learning. In: Proceedings of the 28th international conference on international conference on machine learning. Madison: Omnipress; 2011. p. 265–72.
11. Saad D. Online algorithms and stochastic approximations. Online Learning; 1998. p. 5.
12. Zinkevich M, Weimer M, Li L, Smola AJ. Parallelized stochastic gradient descent. In: Advances in neural information processing systems; 2010. p. 2595–603.
13. Zhang S, Choromanska AE, LeCun Y. Deep learning with elastic averaging sgd. In: Advances in neural information processing systems; 2015. p. 685–93.
14. Serbedzija NB. Simulating artificial neural networks on parallel architectures. Computer. 1996;29(3):56–63.
15. Chen J, Pan X, Monga R, Bengio S, Jozefowicz R. Revisiting distributed synchronous sgd. arXiv preprint; 2016 arXiv:1604.00981.
16. Murata N, Yoshizawa S, Amari S-i. Network information criterion-determining the number of hidden units for an artificial neural network model. IEEE Trans Neural Netw. 1994;5(6):865–72.
17. Johnston WE. High-speed wide area, data intensive computing: a ten year retrospective. In: Proceedings of the seventh international symposium on high performance distributed computing (Cat. No.98TB100244), Chicago, IL, USA, 31 July 1998. IEEE; 1998. https://doi.org/10.1109/HPDC.1998.709982.
18. Gorton I, Greenfield P, Szalay A, Williams R. Data-intensive computing in the 21st century. Computer. 2008;41(4):30–2.
19. Bryant RE Data intensive scalable computing. Carnegie Mellon University; 2009. https://www.cs.cmu.edu/~bryant/pubdir/disc-overview09.pdf. Accessed 10 Aug 2008.
20. Microsoft. Truly consistent hybrid cloud with microsoft azure. 2017. https://azure.microsoft.com/mediahandler/files/resourcefiles/bf2fe090-ec7c-4463-92e7-92501d86dd28/Truly%20Consistent%20Hybrid%20Cloud%20with%20Microsoft%20Azure.pdf.
21. Vaquero LM, Rodero-Merino L, Caceres J, Lindner M. A break in the clouds: towards a cloud definition. ACM SIGCOMM Comput Commun Rev. 2008;39(1):50–5.
22. Varia J, Mathew S. Overview of amazon web services. Amazon Web Services; 2014.
23. Abadi M, Agarwal A, BarhamP, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M,Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y,Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R,Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B,Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V,Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y,Zheng X. TensorFlow: large-scale machine learning on heterogeneousSystems. Software available from tensorflow.org; 2015 https://www.tensorflow.org/.
24. Maddison CJ, Huang A, Sutskever I, Silver D. Move evaluation in go using deep convolutional neural networks. arXiv preprint; 2014. arXiv:1412.6564..
25. Le QV. Building high-level features using large scale unsupervised learning. In: 2013 IEEE international conference on acoustics, speech and signal processing (ICASSP). New York: IEEE; 2013. p. 8595–8.
26. Frome A, Corrado GS, Shlens J, Bengio S, Dean J, Mikolov T, et al. Devise: A deep visual-semantic embedding model. In: Advances in neural information processing systems; 2013. p. 2121–9.

27. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2015. p. 1–9.
28. Zeiler MD, Ranzato M, Monga R, Mao M, Yang K, Le QV, Nguyen P, Senior A, Vanhoucke V, Dean J, et al. On rectified linear units for speech processing. In: 2013 IEEE international conference on acoustics, speech and signal processing (ICASSP). New York: IEEE; 2013. p. 3517–21.
29. Heigold G, Vanhoucke V, Senior A, Nguyen P, Ranzato M, Devin M, Dean J. Multilingual acoustic models using distributed deep neural networks. In: 2013 IEEE international conference on acoustics, speech and signal processing (ICASSP). New York: IEEE; 2013. p. 8619–23.
30. Hinton G, Deng L, Yu D, Dahl GE, Mohamed A-R, Jaitly N, Senior A, Vanhoucke V, Nguyen P, Sainath TN, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Process Mag. 2012;29(6):82–97.
31. GitHub.com: State of the Octoverse. https://octoverse.github.com/ Accessed 10 July 2018.
32. Rosenblatt F. The perceptron: a probabilistic model for information storage and organization in the brain. Psychol Rev. 1958;65(6):386.
33. Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986;323(6088):533.
34. Yann L. Modeles connexionnistes de lapprentissage. Ph.D. thesis, Ph.D. thesis, These de Doctorat, Universite Paris 6; 1987.
35. LeCun Y. The mnist database of handwritten digits; 1998 http://yann.lecun.com/exdb/mnist/. Accessed 1 Aug 2018.
36. Group TH. Hierarchical Data Format, Version 5. http://www.hdfgroup.org/HDF5/. Accessed 15 July 2018.
37. U.S. Government, U.S.C.f.M., Services., M. The offical U.S. Governement Site for Medicare. https://www.medicare.gov. Accessed 29 Oct 2018.
38. CMS. Research, statistics, data, and systems https://www.cms.gov/research-statistics-data-and-systems/research-statistics-data-and-systems.html. Accessed 29 Oct 2018.
39. U.S. Government, U.S. Centers for Medicare and Medicaid Services. What's Medicare? https://www.medicare.gov/sign-up-change-plans/decide-how-to-get-medicare/whats-medicare/what-is-medicare.html. Accessed 29 Oct 2018.
40. CMS. Center for medicare and medicaid services. https://www.cms.gov/. Accessed 29 Oct 2018.
41. HHS. U.S. Department of Health and Human Services. http://www.hhs.gov/. Accessed 29 Oct 2018.
42. Bauder RA, Khoshgoftaar TM. A survey of medicare data processing and integration for fraud detection. In: 2018 IEEE 19th international conference on information reuse and integration (IRI). New York: IEEE; 2018. p. 9–14.
43. Bauder RA, Khoshgoftaar TM. A survey of medicare data processing and integration for fraud detection. In: 2018 IEEE international conference on information reuse and integration (IRI); 2018. p. 9–14. https://doi.org/10.1109/IRI.2018.00010.
44. Bauder RA, Khoshgoftaar TM. Medicare fraud detection using machine learning methods. In: 2017 16th IEEE international conference on machine learning and applications (ICMLA). New York: IEEE; 2017. p. 858–65.
45. Khoshgoftaar TM, Seiffert C, Van Hulse J, Napolitano A, Folleco A. Learning with limited minority class data. In: Sixth international conference on machine learning and applications, 2007. ICMLA 2007. New York: IEEE; 2007. p. 348–53.
46. Seiffert C, Khoshgoftaar TM, Van Hulse J, Napolitano A. Mining data with rare events: a case study. In: Ictai. IEEE; 2007. p. 132–9.
47. Van Hulse J, Khoshgoftaar TM, Napolitano A. Experimental perspectives on learning from imbalanced data. In: Proceedings of the 24th international conference on machine learning. New York: ACM; 2007. p. 935–42.
48. Herland M, Khoshgoftaar TM, Bauder RA. Big data fraud detection using multiple medicare data sources. J Big Data. 2018;5(1):29. https://doi.org/10.1186/s40537-018-0138-3.
49. Prusa J, Khoshgoftaar TM, Dittman DJ, Napolitano A. Using random undersampling to alleviate class imbalance on tweet sentiment data. In: 2015 IEEE international conference on information reuse and integration; 2015. p. 197–202. https://doi.org/10.1109/IRI.2015.39.
50. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference. New York: ACM; 1967. p. 483–485.
51. Berenson M, Levine D, Goldstein M. Intermediate statistical methods and applications: a computer package approach. Englewood Cliffs: Prentice-Hall; 1983.
52. Abdi H, Williams LJ. Tukey's honestly significant difference (hsd) test. Encyclopedia of research design. Thousand Oaks: Sage; 2010. p. 1–5.
53. Seliya N, Khoshgoftaar TM, Van Hulse J. A study on the relationships of classifier performance metrics. In: 21st international conference on tools with artificial intelligence, 2009. ICTAI'09. New York: IEEE; 2009. p. 59–66.