CHAPTER 1

Parallel Computing, Graphics Processing Unit (GPU) and New Hardware for Deep Learning in Computational Intelligence Research

M. MADIAJAGAN, MS, PHD • S. SRIDHAR RAJ, BTECH, MTECH

1.1 INTRODUCTION

Machine learning is the competency of software to perform a single or a series of tasks intelligently without being programmed for those activities, which is a part of artificial intelligence (AI). Normally, the software behaves based on the programmer's coded instructions, while machine learning is going one step further by making the software capable of accomplishing intended tasks by using statistical analysis and predictive analytics techniques. In simple words, machine learning helps the software learn by itself and act accordingly.

Let us consider an example, when we like or comment a friend's picture or video on a social media site, the related images and videos are posted earlier and stay displayed. Same with the "people you may know" suggestions in facebook, the system suggests us another user's profile to add as a friend who is somehow related to our existing friend's list. And you wonder how the system knows that? This is called machine learning. The software uses statistical analysis to identify the pattern that you, as a user, are performing, and using the predictive analytics, it populates the related news feed on your social media site.

1.1.1 Machine and Deep Learning

Machine learning algorithms are used to automatically understand and realize the day-to-day problems that people are facing. The number of hidden layers in an artificial neural network reflects in the type of learning. The intent is to gain knowledge by learning through datasets using customized methods. But, in case of big data where the data is huge and complicated, it is difficult to learn and analyze [1].

Deep learning plays a very vital role in resolving this issue of learning and analyzing big data. It learns complex data structures and representations acquired from raw datasets to derive meaningful information. Big data also supports the nature of deep learning algorithms, which requires large amount of training data. Training many parameters in deep learning networks increases the testing accuracy [2].

Some deep learning applications are in natural language processing, video processing, recommendation systems, disease prediction, drug discovery, speech recognition, web content filtering, etc. As the scope for learning algorithms evolves, the applications for deep learning grows drastically.

1.1.2 Graphics Processing Unit (GPU)

Graphics Processing Unit (GPU) is a specialized circuit which can perform high level manipulation and alter the memory. It can perform rendering of 2D and 3D graphics to acquire the final display. In the beginning, the need for a GPU was driven by the world of computer games, and slowly the researchers realized that it has many other applications like movement planning of a robot, image processing, video processing, etc. The general task of the GPUs was just expressing the algorithms in terms of pixels, graphical representations and vectors. NVIDIA and AMD, two giants in GPU manufacturing, changed the perspective of GPUs by introducing a dedicated pipeline for rendering the graphics using multicore systems. CPU uses vector registers to execute the instruction stream, whereas GPUs use hardware threads which execute a single instruction on different datasets [1].

GPUs, and now TPUs (tensor processing units), reduce the time required for training a machine learning (ML) model. For example, using a CPU approach may take a week to train, a GPU approach on the same problem would take a day, and a TPU approach takes only a few hours. Also, multiple GPUs and TPUs can be used. Multiple CPUs can be used, but network latency and other factors make this approach untenable. As others have noted, GPUs are designed to handle high-dimensional matrices, which is a feature of many ML models. TPUs are designed specifically for ML models and don't include the technology required for image display.

1.1.3 Computational Intelligence

Computational intelligence deals with the automatic adaptation and organizes accordingly with respect to the implementation environment. By possessing the attributes such as knowledge discovery, data abstraction, association and generalization, the system can learn and deal with new situations in the changing environments. Silicon-based computational intelligence comprises hybrids of paradigms such as artificial neural networks, fuzzy systems and evolutionary algorithms, augmented with knowledge elements, which are often designed to mimic one or more aspects of carbon-based biological intelligence [3].

1.1.4 GPU, Deep Learning and Computational Intelligence

GPU is basically based on parallel processing in nature, which helps in improving the execution time of the deep learning algorithms. By imparting the parallel deep learning using GPU, all the computational intelligence research applications which involves images, videos, etc., can be trained at a very fast rate and the entire execution time is reduced drastically.

The rest of the chapter is organized as follows. In Section 1.2, we discuss the role and types of parallelization in deep learning. Section 1.3 tells us the role of GPU in parallel deep learning. Section 1.4 presents the data flow of parallelization and a numerical example on how the parallelization works in deep learning with a real time application. Section 1.5 shows the implementation details and screenshots, while Section 1.6 summarizes the entire contents discussed in the above sections.

1.2 DEEP LEARNING AND PARALLELIZATION

In this section, we will discuss what is parallel processing and the algorithms which are suitable for deep learning through analysis. The analysis is based on the time and throughput of the algorithms.

1.2.1 Parallel Processing Concepts

Parallel processing concept arises to facilitate the analysis of huge data and acquire meaningful information from it. Speech processing, medical imaging, bioinformatics and many similar fields are facing the difficulty of analyzing huge amounts of complex data. There are some problems in which the run-time complexity cannot be improved even with many processors.

Parallel algorithms are called efficient when their run-time complexity divided by the number of processors is equal to the best run-time complexity in sequential processing. Not everything should be parallelized. User experience, for example, is a serial task. If one thread redraws the screen when some other thread is trying to click something that cannot be encouraged for parallel processing, it has to be sequential. Sometimes sequential processing is faster than parallel where the latter requires gathering all the data in one place, but the former does not have to gather data [4].

In single processor systems, a set of inputs are given to the processor and it returns the output after processing. The performance of the processor can be made faster by increasing the frequency limit. But, there is a certain limit beyond which the processor emits a huge amount of heat. The amount of heat emitted by the electrons moving through the processor is very high, hence there is a certain frequency limit beyond which the processor melts down.

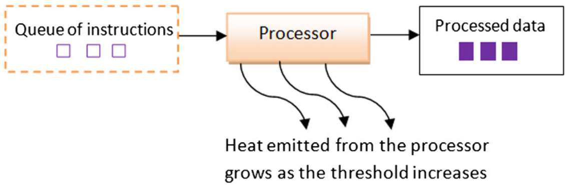

Queue of instructions
Processor
Processed data
Heat emitted from the processor grows as the threshold increases

FIG. 1.1 Single processor execution.

To rectify the issue shown in Fig. 1.1, we move to parallel processing where more than one processor is used to process the data. This way the workload is divided between multiple processors. See Fig. 1.2.

Parallel computing has its own disadvantage such as dependency between processors, i.e., one processor might wait for the results of the process running on another processor. In modern computing, we address the number of processors by using the term core. Dual-core, multi-core, i3, i5, i7, etc., all denote the number of processors.

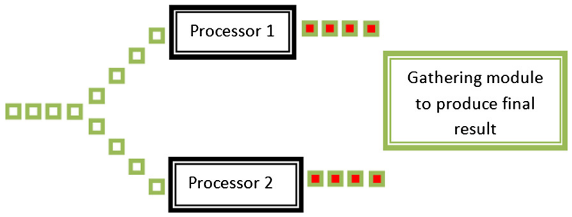


FIG. 1.2 Parallel processing execution.

1.2.2 Deep Learning Using Parallel Algorithms

Machine learning provides solutions on a very small scale even for sophisticated problems. If it is expanded into large scale problems, many new and surprising results can be explored. Unfortunately, the limited capacity of the sequential algorithms in terms of time and complexity has ruined its evolution. Parallel processing offered by modern data processing platforms like Hadoop and Spark can be leveraged for machine learning in multiple ways, especially for hyperparameter and ensemble learning [5].

1.2.2.1 Understanding the Needs and Benefits of Parallel Algorithms in Deep Learning

- Neural networks take a huge number of parameters from the datasets and learn to define the model. This learning of many parameters amounts to a very long computation time. The computation time is considered on order of days, and "q" denotes the number of cores in the processor. The VGGNet application takes about 10 hours for training even on an 8q machine. This is a computationally intensive process which takes a lot of time [6].
- In some cases, the datasets are very large for a single machine to store and process. Therefore we need parallel and distributed processing methodologies which reduce the training time.
- The very nature of deep learning is distributed across processing units or nodes. Using simulated parallelism is slow, but implementing deep learning in it's "natural form" would mean improvements in training time from months to weeks, or even days. Of importance here is the acceleration, nothing else, one can run deep learning solutions on a single processor or machine provided one can tolerate the sluggishness [5]. Hence, the sure way of speeding things up is to use hardware acceleration just like in computer graphics since both graphics and deep learning are inherently parallel in nature [7].

1.2.2.2 Challenges in Implementing Parallel Algorithms in Deep Learning

Applying machine learning algorithms to large scale datasets like web mining, social networks and other distributed environment data is challenging. Research works on making the normal sequential algorithms scalable has still not rectified its purpose. Sequential methods continue to take very long training time. The above mentioned problems face a tough challenge even though MapReduce frameworks which hide much of the complexity.

To overcome this challenge, a framework, which implements parallel machine learning algorithms on large distributed data such as social networks, is based on functional programming abstractions [8]. The algorithms can be implemented very easily by using the functional combinators, which yields the best composition of aggregation, distributed and sequential processes. This system also avoids inversion of control in a synchronous parallel model. Limited graphical processing unit memory is yet another challenge which has to be rectified.

1.2.3 Parallelization Methods to Distribute Computation Across Multiple Machines

The methodologies to perform parallelized or distributed computation on multi-core machines are given below.

1.2.3.1 Local Training

Local training means that the data is being trained on a single machine which has multi-core processors. The entire datasets are loaded onto the same machine, the cores inside take care of the processing task. Multi-core machines can be used in two ways:

- By loading multiple data in a single layer and processing them using the multi-core processor, which is a lengthy parallelization process;
- By using a batching system to separate the datasets into many small batches and sending each batch to a core for processing.

1.2.3.2 Distributed Training

When the datasets are so huge that they cannot be stored on a single system, distributed training resolves this problem. The data is stored across many machines in a distributed manner. Here, either the model or data can be distributed, which is discussed below.

- In data parallelism, data is distributed across multiple machines. When the data set is large or its faster processing is required, data parallelism can be used.

- In model parallelism, the model is typically too big to fit on a single system. When a model is placed into a single machine, one model demands the output of another model. This forward and backward propagation establishes communication between the models from different machines in a serial fashion [9].

1.2.4 Methods to Train Deep Neural Networks Using Parallelization

Deep neural networks or deep artificial neural networks follow the structure of the actual brain and its functions. They use multiple layers of artificial neurons for classification and pattern recognition.

Fig. 1.3 shows the structure of a non-deep neural network, having only one hidden layer, whereas Fig. 1.4 depicts a deep neural network with three hidden layers. Networks having between 3 and 10 hidden layers are called very deep neural networks. There are four ways to parallelize neural network training. They are discussed in what follows.

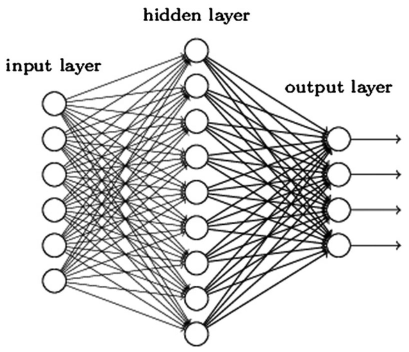


FIG. 1.3 Structure of non-deep neural networks.

1.2.4.1 Inter-Model Parallelism

Generally, when inter-model parallelism is used, there are different models, and each model can have different parameters such as equation function, layer types, number of neurons per layer, etc. All three different model cases are trained with the same dataset. See Fig. 1.5.

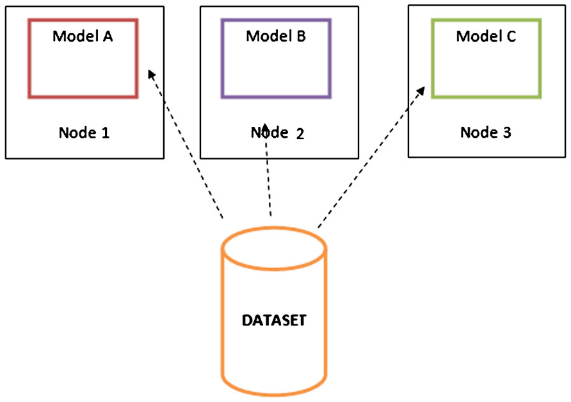


FIG. 1.5 Inter-model parallelism.

1.2.4.2 Data Parallelism

The idea of data parallelism was brought up by Jeff Dean style as parameter averaging. We have three copies of the same model. We deploy the same model A over three different nodes, and a subset of the data is fed over the three identical models. The values of the parameters are sent to the parameter server and, after collecting all the parameters, they are averaged. Using the parameter server, the omega is synchronized. The neural networks can be trained in parallel in two ways, i.e., synchronously (by waiting for one complete iteration and

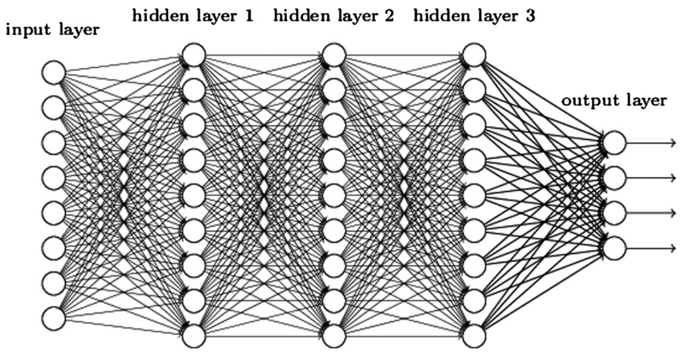


FIG. 1.4 Structure of deep neural networks.

updating the value for omega) and asynchronously (by sending outdated parameters out of the network). But the amount of time taken for both methods is same, and the method choice is not a big issue here. See Fig. 1.6.

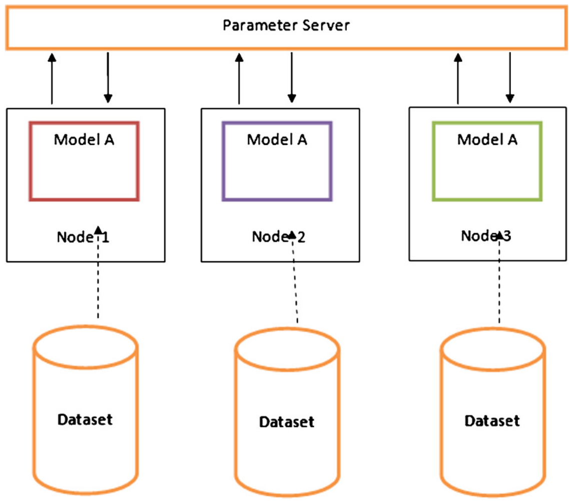


FIG. 1.6 Data parallelism.

To overcome the problems in data parallelism, task level parallelism has been introduced. Independent computation tasks are processed in parallel by using the conditional statements in GPUs. Task level parallelism can act without the help of data parallelism only to a certain extent, beyond which the GPU needs data parallelism for better efficiency. But the task level parallelism gives more flexibility and computation acceleration to the GPUs.

1.2.4.3 *Intra-Model Parallelism*

Machine learning can leverage modern parallel data processing platforms like Hadoop and Spark in several ways. In this section, we will discuss how to scale machine learning with Hadoop or Spark. Three different ways of parallel processing can benefit machine learning. When thinking about parallel processing in the context of machine learning, what immediately jumps to our mind is data partitioning along with divide-and-conquer learning algorithms. However, as we will find out that data partitioning is not necessarily, the best way is to exploit parallel processing. There are other more fruitful areas [10]. See Fig. 1.7.

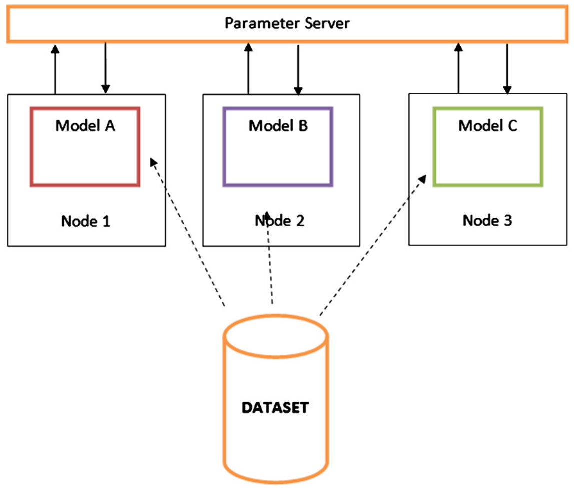


FIG. 1.7 Intra-model parallelism.

1.2.5 Parallelization Over Data, Function, Parameter and Prediction Scale

The focus is on classical machine learning algorithms; moreover, we are only considering Hadoop and Spark for a parallel processing platform.

1.2.5.1 *Data Partitioning*

With data partitioning and parallelism, the learning algorithm operates on each partition in parallel, and finally the results for each partition are stitched together. This is essentially the divide-and-conquer pattern. The critical assumption is that learning algorithms can be recast to execute in parallel. However, not all learning algorithms are amenable to parallel processing.

At first, this may sound very attractive. However, careful consideration of the following observations will lead us to a different conclusion:

- The amount of training data required has a complex relationship with model complexity, expressed as a VC dimension. Higher model complexity demands more training data.
- Most machine learning algorithms are not capable of handling very complex models because of high generalization error. Deep learning is an exception to this.
- Complexity is chosen based on a tradeoff between error due to bias and error due to variance.

Based on these observations, we can conclude that machine learning algorithms, when applied to real world problems, do not require very large training data set and hence do not require parallel processing capabilities of Hadoop or Spark. More on the relationship between model complexity and training data size can be found from our earlier work. We could still use Hadoop or Spark. We can use a sequential learning algorithm that will operate on the whole data set without any partitioning [11].

1.2.5.2 Function Partitioning

This is the flip side of data partitioning. A function is decomposed into several independent functions. Each function operates in parallel on the whole data set. Results are consolidated when all the functions have been computed. There is no learning algorithm that is amenable to functional decomposition. Moreover, Hadoop and Spark provide parallel processing capabilities only through data partitioning.

1.2.5.3 Hyperparameter Learning

This is an area where we can exploit parallel processing in Hadoop and Spark very effectively. Generally, any learning algorithm has many parameters that influence the final result, i.e., test or generalization error. Our goal is to select a set of parameter values that will give us best performance, i.e., minimum error.

This is essentially an optimization problem where we want to minimize error on a multi-dimensional parameter space. However, the error cannot be expressed as a function of the parameters in closed form, and hence many classical optimization techniques cannot be used.

Here are some of the optimization techniques that can be used for finding the optimal set of parameter values. The number of optimization techniques available is by no means limited to this list. For some parameter value sets we build a predictive model and test it to find the test or generalization error [12]. In ensemble learning, multiple predictive models are built. Random forest is a good example where an ensemble of decision trees is used. The ensemble of models is used for prediction, e.g., by taking a majority vote. With ensemble learning, error due to variance can be reduced.

The models in the ensemble are generally built by using a subset of training data and a subset of features. There are other generic ways to create ensembles, e.g., by bagging, boosting and stacking. There are also specific ensemble techniques for learning algorithms. Since the models in the ensemble are trained independently, they can be trained in parallel.

1.2.5.4 Prediction at Scale

Having built a predictive model, sometimes the model needs to be deployed to predict on a massive amount of data and with low latency. Additionally, the prediction may have to be made in close to real time. Here is an example where predictions are to be made in near real time and a large amount of data is involved. Consider a model that predicts the probability of a customer buying something during the current visit to an e-commerce site, based on real time click stream data [5]. This could be done with Spark Streaming with click stream data arriving through a Kafka topic. To maximize throughput, the data could be processed with multiple Spark partitions. Each Spark task processing a partition will load the predictive model [13]. The output of the prediction could be written back to another Kafka topic. The website could personalize content based on prediction from the model [6].

1.2.6 Types of Speed-Up and Scaling

Speeding up and scaling the capacity of the processors leads to reducing the execution time of the processor. The types of scaling the models are discussed below.

The number of resources added for scaling the processor is nearly propositional to the performance of the processor. The resources denote the processors, memory size and bandwidth offered in case of distributed environment. Adding "y" times more resources yields "y" times speed-up [3]. The idea is to scale the number of processors and check the efficiency of the machine.

There are two scalability models:

- Problem constrained;
- Time constrained.

1.2.6.1 Problem Constrained (PC) Scaling

The size of the problem here is fixed, and the reduction of the execution time is the aim. Therefore, without increasing the size of the problem, the number of processors and memory size are increased. The speed-up is computed by the equation below:

$$\text{SPC} = \text{Time (1 processor)} / \text{Time (“}p\text{” processors)}.$$

The ratio of time taken by one processor and time taken by the total number of processors used yields the speed-up value.

1.2.6.2 Time Constrained (TC) Scaling

Unlike the problem constrained case, here in the time constrained situation, the execution time is fixed to the maximum limit. Increasing the problem size is the objective here. Speed-up is defined in terms of the work, and the time is kept constant. "Speed-up" is then defined as

$$\text{Src} = \text{Work (“}p\text{” processors)} / \text{Work (1 processor)},$$

the work or problem size executed by all processors over that accomplished by a single processor [7].

1.3 ROLE OF GRAPHICS PROCESSING UNIT IN PARALLEL DEEP LEARNING

1.3.1 Hardware Architecture of CPU and GPU

A discussion about the architecture of the CPU and GPU is given below with diagrammatic representations.

1.3.1.1 Conventional CPU Architecture

The hardware architecture of a conventional CPU is given below in Fig. 1.8.

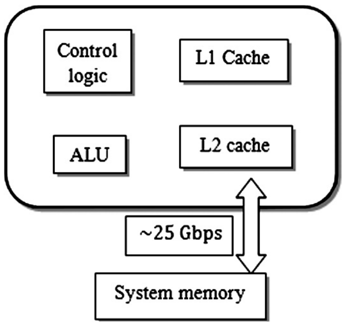


FIG. 1.8 Conventional CPU architecture.

The architecture consists of control logic, cache memory, system memory and arithmetic logic unit (ALU). In CPUs, more space is allocated to control logic than ALU. The latency of a single thread is optimized and hidden for multi-level caches. CPUs can effectively regulate the flow control even under heavy workloads. A typical multi-core CPU has the number of ALUs ranging between 1 to 32, and the cache memory is shared across all the cores.

1.3.1.2 Modern GPU Architecture

In the modern GPU architecture (Fig. 1.9), very less space is allocated to the control logic and cache memory. Since it has multiple threads across the core, large register files are required to adapt them. Unlike CPU, here many ALUs are placed in each core. Each core has a small cache memory, which is user manageable. A bandwidth of about 150 GB/s manages to service many ALUs at the same time.

The main specialization of GPU is the in-depth and data parallel computation. Therefore, more transistors are allocated for processing the data instead of flow control and caching mechanism [14].

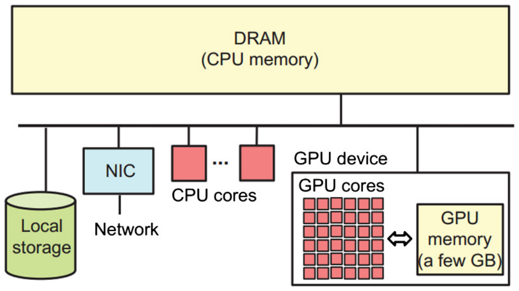


FIG. 1.9 Modern GPU architecture.

1.3.2 Suitability of GPU to Parallel Deep Learning

The suitability of GPU for deep learning is discussed by an example. Consider a PC game where a low-end machine is chosen with just a CPU or a high-end machine with a CPU and GPU. It is possible to play some games on low-end machines but the frame rate is quite low compared to the frame rate obtained on a high-end machine. The GPU speeds up or accelerates graphical computations very well; both CPU and GPU can handle graphical operations, but the latter performs faster because of the distributed setup. The parallel architecture in a GPU can perform matrix and vector operations effectively. In 3D computer graphics there are a lot of such operations, like computation of lighting effects from normal maps and 3D effects. GPUs were designed to handle such vector and matrix operations in parallel unlike a single core CPU that would handle matrix operations in serial form processing one element at a time. This makes it possible to play games at 60 fps with impressive real-time visuals. Now coming back to deep learning, there are a lot of vector and matrix operations [10].

A lot of data can be stored in L1 caches and register files on GPUs to reuse convolutional and matrix multiplication tiles. For example, the best matrix multiplication algorithms use 2 tiles of 64×32 to 96×64 numbers for 2 matrices in L1 cache, and a 16×16 to 32×32 number register tile for the output sums per thread block (1 thread block = up to 1024 threads; one has 8 thread blocks per stream processor, there are 60 stream processors in total for the entire GPU). If you have a 100 MB matrix, it can be split into smaller matrices that fit into your cache and registers, and then you can do matrix multiplication with three matrix tiles at speeds of 10–80 TB/s. This is one more reason why GPUs are so much faster than CPUs, and why they are so well suited for deep learning [15].

1.3.3 CPU vs. GPU

- CPUs are designed for more general computing workloads. GPUs in contrast are less flexible; however, GPUs are designed to compute the same instructions in parallel. See Fig. 1.10.

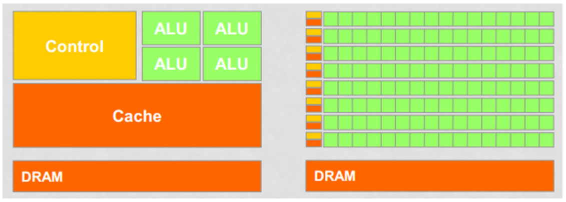


FIG. 1.10 Architecture difference between CPU and GPU.

- In image processing applications, GPU's graphics-specific capabilities can be exploited to speed up the calculations further.
- The primary weakness of GPUs as compared to CPUs is memory capacity on GPUs which is lower than on CPUs. The best known GPU contains 24 GB of RAM; in contrast, CPUs can reach 1 TB of RAM. A secondary weakness is that a CPU is required to transfer data into the GPU card. This takes place through the PCI-E connector which is much slower than CPU or GPU memory. The final weakness is that GPUs' clock speeds are one-third that of high-end CPUs, so on sequential tasks a GPU is not expected to perform comparatively well.
- GPUs are so fast because they are so efficient in matrix multiplication and convolution, and the reason for this is memory bandwidth and not necessarily parallelism. In short and in order of importance, high bandwidth main memory, hiding memory access latency under thread parallelism and large and fast register and L1 memory, which is easily programmable, are the components which make GPUs so well suited for deep learning.
- CPUs are latency optimized while GPUs are bandwidth optimized.
- The CPU L1 cache only operates at about 5 TB/s, which is quite slow, and has the size of roughly 1 MB; CPU registers usually have sizes of around 64–128 KB and operate at 10–20 TB/s. Of course, this comparison of numbers is a bit flawed because registers operate a bit differently than GPU registers (a bit like comparing apples and oranges), but the difference in size here is more crucial than the difference in speed, and it does make a difference.
- It is easy to tweak the GPU code to make use of the right amount of registers and L1 cache for fast performance. This gives GPUs an advantage over other architectures like Xeon Phis where this utilization is difficult to achieve and difficult to debug, which in the end makes it difficult to maximize performance on a Xeon Phis. [5].

1.3.4 Advantages of Using GPU in Parallel Deep Learning

- The advantage of the GPU here is that it can have a small pack of registers for every processing unit (steam processor, or SM), of which it has many. Thus we can have a lot of register memory in total, which is very small and thus very fast. This leads to the aggregate GPU registers' size being more than 30 times larger compared to CPUs and still twice as fast, which translates into up to 14 MB register memory that operates at a whooping 80 TB/s.
- A neural network involves lots of matrix manipulations, such as multiplication, addition and element-wise calculations. These manipulations can be significantly sped up because they are highly parallelizable.
- GPUs are massively parallel calculators that allow performing many mathematical operations very quickly and at once. Using GPUs cuts down the training time.
- GPU programming must be vectorized to be effective. This is because GPU processors are built to do computations on images which come in a form of matrices, so vectorized operations are natural in this domain.
- Deep neural networks and most AI stuff in machine learning (ML) can thus be cast as parallel problems, which means parallel computing solutions like GPUs can speed up 90% or so of the algorithms in AI, only few algorithms such as tree traversing or recursive algorithms are not parallelizable, so those can be handled on a CPU more efficiently.
- GPUs are best for speeding up distributed algorithms whereby each unit in the distributed system works independently of the other units. Thus, an ensemble of processing nodes in a neural network, like most AI algorithms, fall into this category.

1.3.5 Disadvantages of Using GPU in Parallel Deep Learning

- Full register utilization in GPUs seems to be difficult to achieve at first because it is the smallest unit of computation which needs to be fine-tuned by hand for good performance. But NVIDIA has developed good compiler tools here which exactly indicate when you are using too many or too few registers per stream processor.

- One example algorithm that is hard to get sped up from GPUs is the Fibonacci sequence calculation, which is sequential. By speeding up the calculations, neural networks can be optimized using more data and more parameters, thanks to progress in deep learning.
- CPUs are better suited to perform a wider range of operations, at the cost of slower performance for some of the rendering-specific operations.
- A CPU usually has less cores, current CPUs commonly have between 4 and 16, while newer high-end GPUs have more than a 1000. Each of these cores is essentially a computer in itself. So, why isn't a GPU always better than a CPU if it has way more cores? One reason is that the clock speed is typically much higher on a CPU, meaning that each of the individual cores can perform more calculations per second than the individual cores of the GPU. This makes CPUs faster on sequential tasks.
- There are other things to consider when calculating the benefit of using a GPU instead of a CPU, such as memory transfer. If you want to multiply 1000 numbers on the GPU, you need to tell it first what those 1000 numbers are, so the GPU tends to be more useful in cases where you need to do a lot of things with little change in input and a small volume of output.

1.3.6 Famous GPUs on the Market

NVIDIA and AMD are the two leading manufacturers, the GPUs of which are widely used in the technology world. The discussion about both the GPUs are carried out below.

1.3.6.1 NVIDIA

All the NVIDIA products are under the standard compute unified device architecture (CUDA). CUDA is defined as an architectural register, where the binary files required for execution of one CUDA do not necessarily work for another CUDA based GPU. The CUDA GPU consists of multiprocessors which execute multiple threads in blocks. Multiple blocks can be executed by a multiprocessor simultaneously. Each multiprocessor contains 8 CUDA cores with compute capability of 1x. As the capability increases to 2, 3, etc., the number of CUDA cores also increases.

To hide the memory access and arithmetic latencies, multiple threads have to be concurrently executed. NVIDIA runs 192 to 256 threads per multiprocessor for the GPUs having compute capability 1x. It is better to run more threads in case of data parallelism to free up the registers. High performance can be achieved only by knowing the optimal number of threads that can be concurrently executed in a multiprocessor. The unified virtual address is useful in establishing the connection between two GPUs.

1.3.6.2 AMD

AMD accelerated parallel processing (APP) or ATI Stream is the technology which is used to execute general computations. Each APP device consists of multiple compute units, each compute unit contains multiple stream cores, and each core contains multiple processing elements. The instances of GPU program are concurrently executed, each instance is named as a work item. In lockstep, multiple work items are executed in parallel by all the cores of a compute unit. The total work items are decided based on the hardware and requirement of the programmer for a particular work group [3].

1.4 GPU BASED PARALLEL DEEP LEARNING ON COMPUTATIONAL INTELLIGENCE APPLICATIONS WITH CASE STUDY

In this section, we discuss the computational intelligence applications and how GPU based parallel deep learning is applied over those applications by considering examples.

1.4.1 Dataflow of the Deep Parallelized Training and Testing of Computational Intelligence Applications

Applying the parallel deep learning methodology over the computational intelligence research applications brings a greater challenge in implementation. The overheads, applicability problems and related issues are addressed in this section.

A general model has been designed in order to execute the computational intelligence applications data in parallel with deep learning algorithms. Fig. 1.8 depicts the data flow of the parallel execution using deep learning. The data flow comprises of the following steps:

1. The required data is collected by means of a sensor or similar devices from the subject.
2. Once the data is ready for training, the entire data is separated into training and test data.
3. The training data is fed into the model.
4. In order to perform parallel processing, the dataset has to be separated into halves for parallel processing. Certain scheduling algorithms can be used to schedule the processes based on the number of cores available in the process.
5. The dataset gets trained in the separate cores.

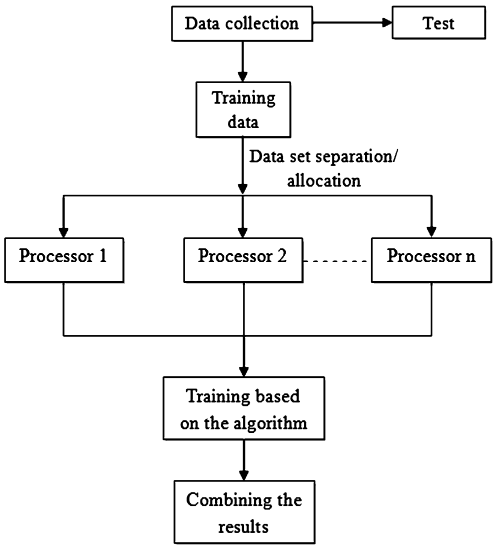


FIG. 1.11 Dataflow of parallel dataset training.

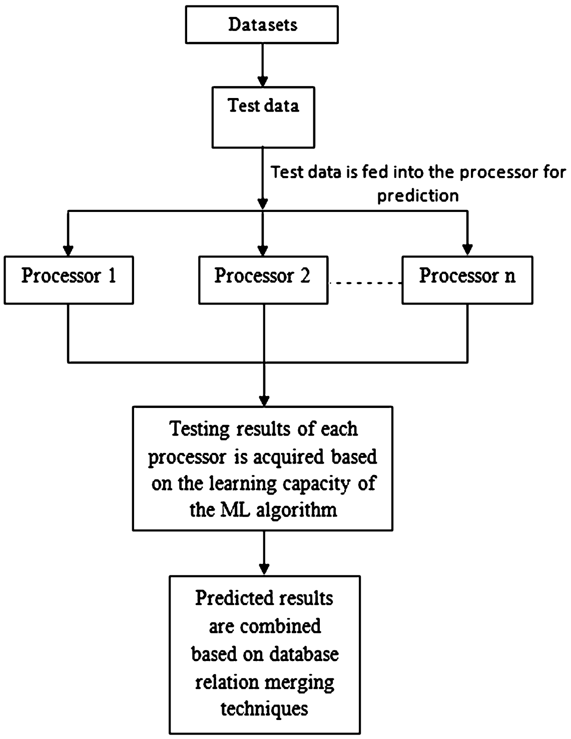

Datasets
Test data
Test data is fed into the processor for prediction
Processor 1
Processor 2
Processor n
Testing results of each processor is acquired based on the learning capacity of the ML algorithm
Predicted results are combined based on database relation merging techniques

FIG. 1.12 Dataflow of parallel testing process.

6. After the training is over, the results of the trained data have to be combined into a single module.
7. By having the trained results in a single module makes the testing process smoother. See Figs. 1.11 and 1.12.

1.4.2 Numerical Example for a Generic Computational Intelligence Application

Let us consider some real time numerical values regarding the time overhead led by imparting parallel processing in deep learning applications. Let us assume that

- The total number of records in the dataset is 1100;
- A quadcore processor is given for parallel execution.

Training part parameters:

- The total number of processors given equals 4;
- The number of training records in the dataset is 1000;
- The number of records given for each processor for parallel execution is 250;
- The time taken for one data allocation to the processor is 1 s ($4 * 1 = 4$ s);
- The training time of one record is 1 s;
- The table merging time is 2 s per merging;
- The overall computation of the quadcore processor takes 10 s.

Testing part parameters:

- The total number of processors given is 4;
- The number of testing records in the dataset is 100;
- The number of records given for each processor for parallel execution is 25;
- The time taken for one data allocation (separation) to the processor equals 1 s ($4 * 1 = 4$ s);
- The training time of one record is 1 s;
- The table merging time is 2 s per merging;
- The overall computation of the quadcore processor takes 10 s;
- The prediction time is 2 s;
- Writing the predicted result to the dataset takes 1 s per record;

- Recording the datasets is done as required by the application;
- Once the dataset is ready, the data schema has to be framed in such a way that it is lossless and dependency preserving;
- Now, the first level of dataset separation involves splitting the data into training and test data. This level of data separation need not be checked for lossless and dependency preserving properties, since we are not going to combine it again;
- The second level involves dividing or scheduling the dataset based on the availability and the number of processors given for execution.

FIG. 1.13 Training image.

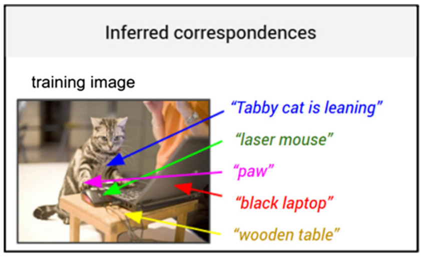


FIG. 1.14 Inferred correspondences.

1.4.3 Dealing With Limited GPU Memory

A major problem with parallelized deep learning is limited memory. All the intermediate states and the input mini-batches have to be fit into the limited GPU memory. GeePS architecture is introduced to rectify this problem by buffering the data to the GPU memory from the large CPU memory.

The efficiency of the GeePS is demonstrated using only a fraction of the GPU memory for the largest case and by experimenting with a much larger synthetic model. GeePS's memory management support allows us to do video classification on longer videos. If we further consider the memory used for testing, the supported maximum video length will be shorter. The parallel approach to split the model across four machines incurs extra network communication overhead. By contrast, with the memory management support of GeePS, we are able to train videos with up to 192 frames, using solely data parallelism.

1.4.4 Computational Intelligence Applications

The different application areas of computational intelligence possess a hybrid of neural networks, fuzzy systems and evolutionary algorithms. The categorization of applications is as follows:

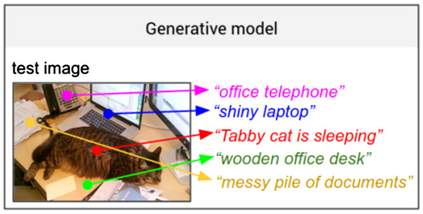


FIG. 1.15 Generative model.

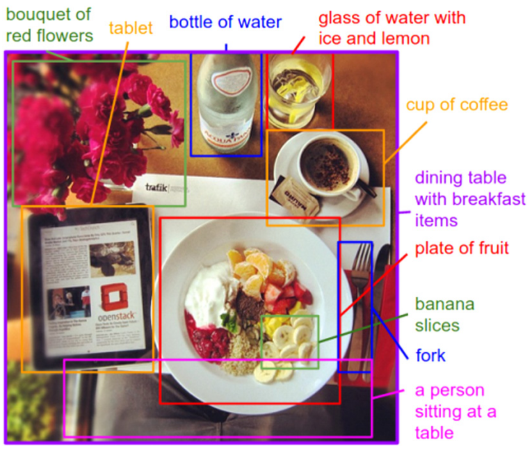

bouquet of red flowers
tablet
bottle of water
glass of water with ice and lemon
cup of coffee
dining table with breakfast items
plate of fruit
banana slices
fork
a person sitting at a table

FIG. 1.16 Detection of objects in an image.

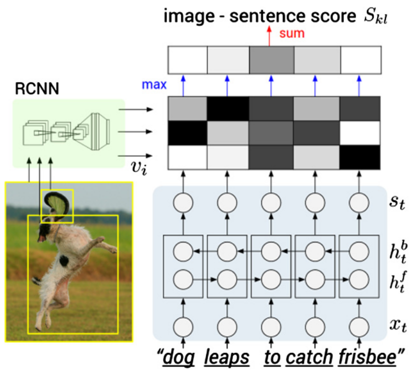
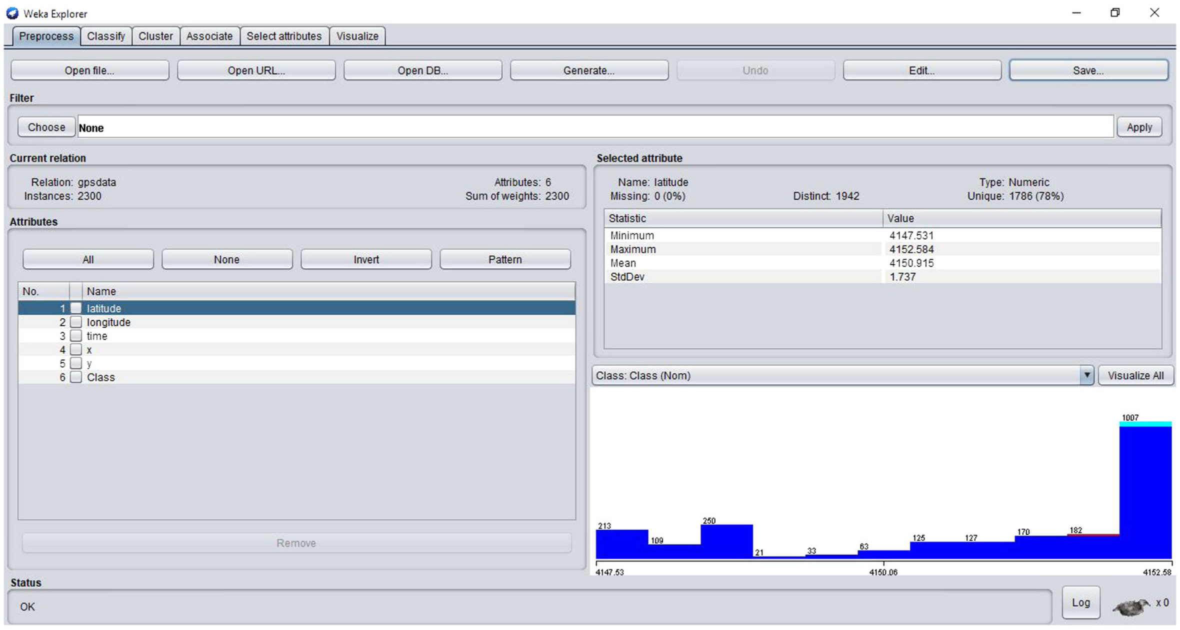


FIG. 1.17 Neural network training of the segments of the image.

1. Neural networks are used on problems of category clustering, classification, prediction, composition and control systems.
2. Evolutionary computation involves route or path optimization, scheduling problem and medical diagnosis of diseases.
3. Fuzzy logic deals with vehicle monitoring, sensor data in home appliances and control systems.



Filter: None

Current relation
Relation: gpsdata
Instances: 2300
Attributes: 6
Sum of weights: 2300

Attributes

No.	Name
1	latitude
2	longitude
3	time
4	x
5	y
6	Class

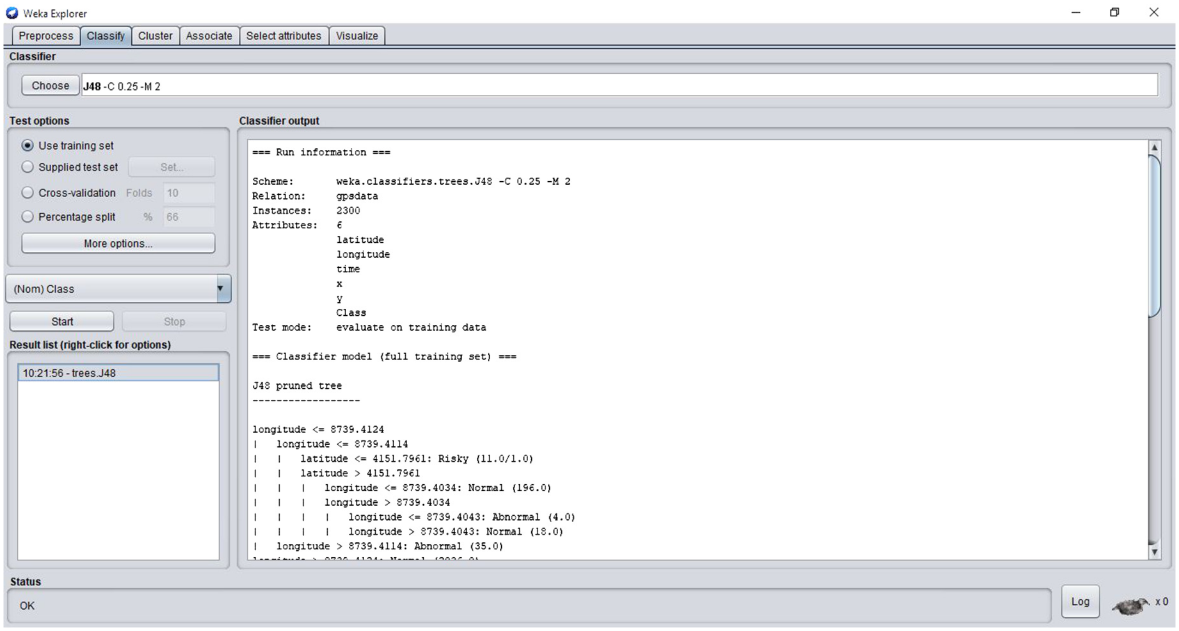
Selected attribute
Name: latitude
Missing: 0 (0%)
Distinct: 1942
Type: Numeric
Unique: 1786 (78%)

Statistic	Value
Minimum	4147.531
Maximum	4152.584
Mean	4150.915
StdDev	1.737

Class: Class (Nom)

Status
OK

FIG. 1.18 Type of pre loaded data selection.



Classifier: J48 -C 0.25 -M 2

Test options
[✓ Use training set]
Supplied test set
Cross-validation Folds 10
Percentage split % 66

(Nom) Class

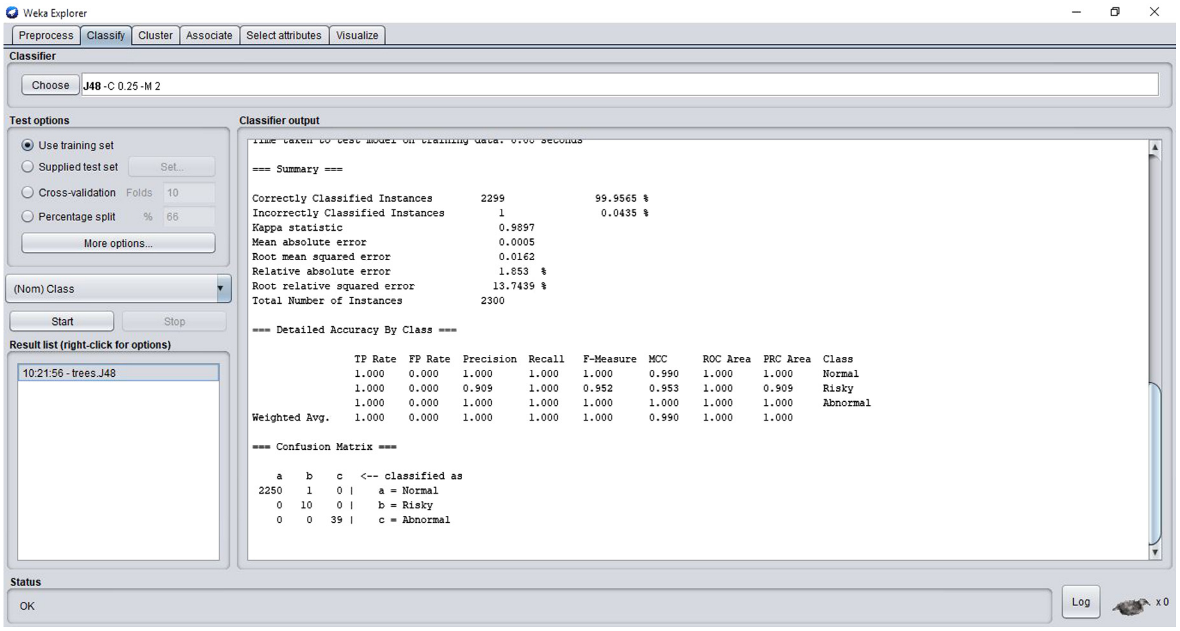
Result list (right-click for options)
10:21:56 - trees.J48

Classifier output

```
=== Run information ===

Scheme:       weka.classifiers.trees.J48 -C 0.25 -M 2
Relation:     gpsdata
Instances:    2300
Attributes:   6
              latitude
              longitude
              time
              x
              y
              Class
Test mode:    evaluate on training data

=== Classifier model (full training set) ===

J48 pruned tree
------------------

longitude <= 8739.4124
|   longitude <= 8739.4114
|   |   latitude <= 4151.7961: Risky (11.0/1.0)
|   |   latitude > 4151.7961
|   |   |   longitude <= 8739.4034: Normal (196.0)
|   |   |   longitude > 8739.4034
|   |   |   |   longitude <= 8739.4043: Abnormal (4.0)
|   |   |   |   longitude > 8739.4043: Normal (18.0)
|   longitude > 8739.4114: Abnormal (35.0)
```

Status
OK

FIG. 1.19 Preprocessing phase.



Classifier: J48 -C 0.25 -M 2

Test options: (•) Use training set; ( ) Supplied test set; ( ) Cross-validation Folds 10; ( ) Percentage split % 66

(Nom) Class

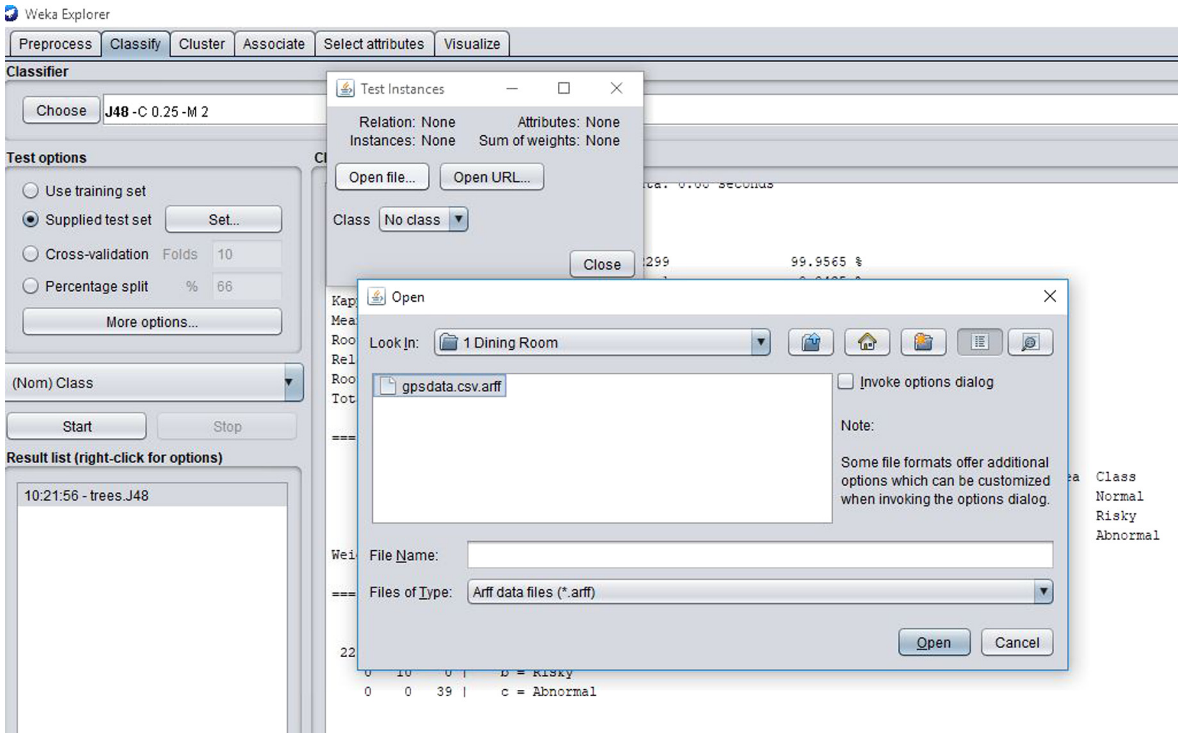
Result list (right-click for options): 10:21:56 - trees.J48

Status: OK

Classifier output

```
Time taken to test model on training data: 0.08 seconds

=== Summary ===

Correctly Classified Instances        2299               99.9565 %
Incorrectly Classified Instances         1                0.0435 %
Kappa statistic                          0.9897
Mean absolute error                      0.0005
Root mean squared error                  0.0162
Relative absolute error                  1.853  %
Root relative squared error             13.7439 %
Total Number of Instances             2300

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 1.000    0.000    1.000      1.000    1.000      0.990    1.000     1.000     Normal
                 1.000    0.000    0.909      1.000    0.952      0.953    1.000     0.909     Risky
                 1.000    0.000    1.000      1.000    1.000      1.000    1.000     1.000     Abnormal
Weighted Avg.    1.000    0.000    1.000      1.000    1.000      0.990    1.000     1.000

=== Confusion Matrix ===

    a    b    c   <-- classified as
 2250    1    0 |    a = Normal
    0   10    0 |    b = Risky
    0    0   39 |    c = Abnormal
```

FIG. 1.20 Setting the cross-validation for one instance of a processor.



Classifier: J48 -C 0.25 -M 2

Test options: ( ) Use training set; (•) Supplied test set; ( ) Cross-validation Folds 10; ( ) Percentage split % 66

(Nom) Class

Result list (right-click for options): 10:21:56 - trees.J48

Test Instances

Relation: None Attributes: None
Instances: None Sum of weights: None

Class: No class

Open

Look In: 1 Dining Room

gpsdata.csv.arff

☐ Invoke options dialog

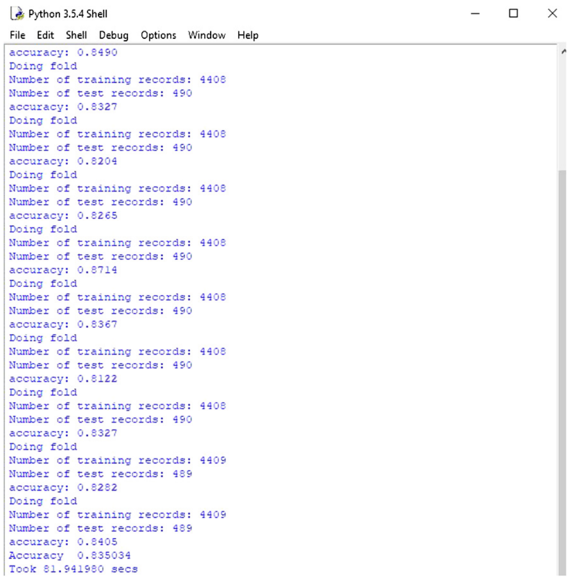
Note:

Some file formats offer additional options which can be customized when invoking the options dialog.

File Name:

Files of Type: Arff data files (*.arff)

FIG. 1.21 Classification outcome for a general application.



```
accuracy: 0.8490
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8327
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8204
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8265
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8714
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8367
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8122
Doing fold
Number of training records: 4408
Number of test records: 490
accuracy: 0.8327
Doing fold
Number of training records: 4409
Number of test records: 489
accuracy: 0.8282
Doing fold
Number of training records: 4409
Number of test records: 489
accuracy: 0.8405
Accuracy  0.835034
Took 81.941980 secs
```

FIG. 1.22 Snippet while the dataset is getting trained in Python shell.

4. Expert systems have financial applications, robot production, diagnostics and various industry based operations.

1.5 IMPLEMENTATION SCREENSHOTS TO VISUALIZE THE TRAINING PROCESS

Parallel execution of GPU involves data parallelism, where the training images are divided into separate batches. After the computation of the GPU batches separately, the average of the entire batch is calculated. Since the batching results are synchronous, the results will not vary from a single GPU execution. This batching process has been observed to produce 3.75 times better speed than a single GPU.

Fig. 1.13 presents a training image, which shows a tabby cat leaning on a wooden table, with one paw on a laser mouse and the other on a black laptop.

Fig. 1.14 shows the objects or the information inferred from the training. Fig. 1.15 is the test image which is different from the original image. The deep learning algorithms perform identification of the information by learning from the trained data.

Fig. 1.16 gives another example of a sample image (some items placed in a table) to illustrate the detection of the pixels and corresponding objects in the image.

Fig. 1.17 shows the training mechanism of the image using deep convolutional neural networks. Also see Figs. 1.18–1.22.

1.6 SUMMARY

GPUs work well on parallel deep neural network computations because:

- GPUs have many more resources and faster bandwidth to memory;
- Deep neural networks' computations fit well with GPU architecture. Computational speed is extremely important because training of deep neural networks can take from days to weeks. In fact, many of the successes of deep learning may have not been discovered if it were not for the availability of GPUs.
- Deep learning involves huge amounts of matrix multiplications and other operations, which can be massively parallelized and thus sped up on GPUs.

In this chapter, the basic concepts of parallel processing are explained with examples in order to make a clear way to parallel deep learning. Various parallelization techniques are discussed with diagrammatic explanation, and ways in which they can be internally classified are focused on. The relation between computational intelligence and parallel deep learning, the challenges in combining them together and benefits are discussed. The applicability of the parallel deep learning algorithms to the real time datasets are explained with simple numerical examples.

REFERENCES

1. Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436.
2. J. Schmidhuber, Deep learning in neural networks: an overview, Neural Networks 61 (2015) 85–117.
3. Q.V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, A.Y. Ng, On optimization methods for deep learning, in: Proceedings of the 28th International Conference on International Conference on Machine Learning, Omnipress, 2011, pp. 265–272.
4. A. Karpathy, L. Fei-Fei, Deep visual-semantic alignments for generating image descriptions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3128–3137.
5. S. Salza, M. Renzetti, Performance modeling of parallel database systems, Informatica-Ljubljana 22 (1998) 127–140.
6. G. Hinton, L. Deng, D. Yu, G.E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T.N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, IEEE Signal Processing Magazine 29 (6) (2012) 82–97.
7. V. Hegde, S. Usmani, Parallel and Distributed Deep Learning, Tech. report, Stanford University, June 2016, https://stanford.edu/~rezab/dao/projects_reports/hedge_usmani.pdf.

8. K.R. Foster, R. Koprowski, J.D. Skufca, Machine learning, medical diagnosis, and biomedical engineering research-commentary, Biomedical Engineering Online 13 (1) (2014) 94.
9. Y.B. Kim, N. Park, Q. Zhang, J.G. Kim, S.J. Kang, C.H. Kim, Predicting virtual world user population fluctuations with deep learning, PLoS ONE 11 (12) (2016) e0167153.
10. I. Goodfellow, Y. Bengio, A. Courville, Y. Bengio, Deep Learning, vol. 1, MIT Press, Cambridge, 2016, p. 2016.
11. A. Ike, T. Ishihara, Y. Tomita, T. Tabaru, Technologies for practical application of deep learning, Fujitsu Scientific and Technical Journal 53 (5) (2017) 14–19.
12. N. Friedman, M. Linial, I. Nachman, D. Pe'er, Using Bayesian networks to analyze expression data, Journal of Computational Biology 7 (3–4) (2000) 601–620.
13. S.S. Raj, M. Nandhini, Ensemble human movement sequence prediction model with a priori based probability tree classifier (APTC) and bagged j48 on machine learning, Journal of King Saud University: Computer and Information Sciences (2018).
14. H. Greenspan, B. Van Ginneken, R.M. Summers, Guest editorial deep learning in medical imaging: overview and future promise of an exciting new technique, IEEE Transactions on Medical Imaging 35 (5) (2016) 1153–1159.
15. I.-H. Chung, T.N. Sainath, B. Ramabhadran, M. Picheny, J. Gunnels, V. Austel, U. Chauhari, B. Kingsbury, Parallel deep neural network training for big data on Blue Gene/Q, IEEE Transactions on Parallel and Distributed Systems 28 (6) (2017) 1703–1714.