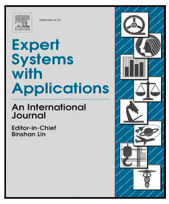
Expert Systems With Applications 279 (2025) 127386

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



A parallel approach to accelerate neural network hyperparameter selection for energy forecasting

D. Criado-Ramón [a,*], L.G.B. Ruiz [b], M.C. Pegalajar [a]

[a] Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain
[b] Department of Software Engineering, University of Granada, Granada, Spain

ARTICLE INFO

Keywords:
Neural networks
Energy
Time series forecasting
Parallel computing
Hyperparameter selection

ABSTRACT

Finding the optimal hyperparameters of a neural network is a challenging task, usually done through a trial-and-error approach. Given the complexity of just training one neural network, particularly those with complex architectures and large input sizes, many implementations accelerated with GPU (Graphics Processing Unit) and distributed and parallel technologies have come to light over the past decade. However, whenever the complexity of the neural network used is simple and the number of features per sample is small, these implementations become lackluster and provide almost no benefit from just using the CPU (Central Processing Unit). As such, in this paper, we propose a novel parallelized approach that leverages GPU resources to simultaneously train multiple neural networks with different hyperparameters, maximizing resource utilization for smaller networks. The proposed method is evaluated on energy demand datasets from Spain and Uruguay, demonstrating consistent speedups of up to 1164x over TensorFlow and 410x over PyTorch.

1. Introduction

In the last decade, neural networks have become one of the most relevant Artificial Intelligence (AI) models of our time, being used with astonishing results in a wide range of applications such as computer vision (Voulodimos, Doulamis, Doulamis, Protopapadakis, et al., 2018), time series forecasting (Hewamalage, Bergmeir, & Bandara, 2021), speech recognition (Nassif, Shahin, Attili, Azzeh, & Shaalan, 2019) or natural language processing (Alshemali & Kalita, 2020). In the energy sector, many neural network architectures have been used to forecast energy consumption in households, public buildings and entire markets, among others. However, the prevailing trend in recent years has been the use of Deep Neural Networks, usually incorporating the Long-Short Term Memory (LSTM) architecture in at least one of the hidden layers. In fact, this architecture is featured in almost 50% of the publications that used a Recurrent Neural Network (RNN) to predict energy consumption in buildings (Lu, Li, & Lu, 2022).

Several recent works show that the use of LSTM neural networks or hybrid models comprised of at least one LSTM layer usually outperforms other machine learning approaches to forecast energy consumption. Kim, Choi, Jeon, and Liu (2019) proposed a hybrid model with LSTM and Convolutional Neural Network (CNN) that showed better results than ARIMA and a combination of LSTM and Seq2Seq in energy demand data from Korea's electric grid. Another hybrid model was proposed by Yan, Li, Ji, Qi, and Du (2019) to forecast energy consumption in individual households. This hybrid model featured LSTM neural networks with a Stationary Wavelet Transform and achieved more accurate forecasts than the standalone LSTM, hybrid models combining LSTM and CNN, and Support Vector Regression. Torres, Martínez-Álvarez, and Troncoso (2022) presented a deep LSTM architecture to forecast energy demand on the Spanish electric grid. The results showed that the deep LSTM neural network outperformed other deep neural network architectures and other Machine Learning models. Jin et al. (2022) used a hybrid model of Singular Spectrum Analysis and parallel LSTMs to forecast energy consumption of multiple UK households at different sampling rates. Rick and Berton (2022) presented a different hybrid model comprised of CNNs, LSTMs and autoencoders, during the same year, to study energy consumption in the grid of a Brazilian energy distributor.

In closely related fields, such as power generation forecasting, hybrid models featuring the LSTM architecture have also become the state of the art. Zhou et al. (2019) evaluated the enhancement provided by the inclusion of an attention mechanism in the LSTM architecture to forecast photovoltaic power generation. Wan, Chang, AL-Bukhaiti, and He (2023) applied a similar idea to simultaneously forecast power and heat with a hybrid model that combines CNN, LSTM and attention, outperforming other deep learning models.

* Corresponding author at: Department of Software Engineering, University of Granada, Granada, Spain.
E-mail addresses: dcriado@ugr.es (D. Criado-Ramón), bacaruiz@ugr.es (L.G.B. Ruiz), mcarmen@decsai.ugr.es (M.C. Pegalajar).

https://doi.org/10.1016/j.eswa.2025.127386
Received 25 October 2024; Received in revised form 19 March 2025; Accepted 21 March 2025
Available online 31 March 2025
0957-4174/© 2025 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

Nonetheless, there are also other scenarios where simpler neural network architectures are a better fit for the problem, particularly in cases where the amount of data available to train the model is limited. Manno, Martelli, and Amaldi (2022) showed how a simpler feed-forward neural network with one hidden layer could provide better hourly forecasts than LSTM and other machine learning models in three energy datasets, and Maragkos, Tzelepi, Passalis, Adamakos, and Tefas (2022) showed how a simple multilayer perceptron (MLP) with two hidden layers could outperform the deep pre-trained model ResNetPlus to forecast energy consumption in the Greek market.

Given the large variety of ML models that can be applied to produce accurate forecasts and the large search space of hyperparameters, finding the optimal model for a specific task can be challenging and time-consuming, as they are usually evaluated with a trial-and-error approach. This can be done either exhaustively over a selected range of hyperparameters ("grid search") or guided by some optimization algorithm (Luo & Oyedele, 2021). This large search space for hyperparameters in conjunction with the slow training time of some of the most complex models has led to the development of specialized implementations that leverage specific hardware to accelerate the training process.

A noteworthy example of this trend is the prevalent use of Graphics Processing Units (GPUs) for training machine learning models. Nowadays, most machine learning models, particularly Artificial Neural Networks, leverage GPUs for efficient training. Initially, parallel implementations were introduced for specific neural network architectures (Jang, Park, & Jung, 2008; Uetz & Behnke, 2009). However, with the advent of user-friendly neural network frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019), efficient implementations of the majority of neural network architectures have become easily accessible. Furthermore, the significance of AI in the GPU landscape has prompted manufacturers to offer libraries with tailored primitives for deep learning (Appleyard, Kociský, & Blunsom, 2016; Chetlur et al., 2014), which these frameworks utilize to optimize the training process. This inclination toward GPU utilization extends beyond neural networks to include other traditional machine learning models. For example, the cuML library (Raschka, Patterson, & Nolet, 2020), also developed by a GPU manufacturer, facilitates seamless GPU-accelerated usage of various classic machine learning models, and many recently proposed machine learning models have been released with a GPU implementation available (Chen & Guestrin, 2016; Ke et al., 2017).

However, despite the widespread availability of GPU implementations for many machine learning models, certain specialized use cases still lack efficient implementations. For example, CUDA implementations of metaheuristic algorithms are not generally available and are frequently studied in the AI literature for different purposes (Iruela, Ruiz, Pegalajar, & Capel, 2020; Kintsakis, Chrysopoulos, & Mitkas, 2015; Ting, Ma, Kim, & Huang, 2016; Wang, Zhang, Huang, & Tsui, 2018; Zhuo, Zhang, Du, & Liu, 2023). In the context of Artificial Neural Networks (ANNs), publicly accessible implementations are generally tailored to enhance training speed with a large number of features or neurons. These implementations often depend on the use of efficient parallelized General Matrix Multiplications (GEMM), typically facilitated by linear algebra libraries provided by the hardware manufacturer, such as cuBLAS (NVIDIA, 2023). Consequently, employing these approaches for training small neural networks on GPUs might result slower than using the CPU. One potential remedy for this challenge could involve increasing the batch size during training to operate on larger matrices, assuming sufficient data is available. However, it is widely recognized that an excessively large batch size can compromise accuracy, leading to poorer generalization (Keskar, Mudigere, Nocedal, Smelyanskiy, & Tang, 2017).

Another prospective solution could involve leveraging GPU resources to simultaneously train, in parallel, multiple neural networks with different hyperparameters, which would provide a better GPU utilization than using the entire computational power of a GPU to train a single small neural network. Notably, the application of GPUs under these circumstances has not been thoroughly examined in the current literature and the implementations provided by the mainstream neural network frameworks are limited for this use case, as they are optimized to train one large model. Therefore, in this study, we aim to address this gap by developing and evaluating an efficient GPU implementation capable of training multiple shallow neural networks simultaneously, each with different hyperparameters. Our evaluation will focus on energy forecasting data, where the number of input features is typically relatively low, involving only the previous number of time steps used for the forecast and a few exogenous variables like temperature. Thus, this paper strives to contribute to the existing body of knowledge and addresses the following research questions.

- Is it faster to train simultaneously multiple neural networks in the GPU or use the optimized implementation from libraries such as TensorFlow or PyTorch to accelerate the training of each neural network?
- How do the batch size and complexity of each neural network affect the results?

These questions will be solved with a real-case study with energy demand data from Spain and Uruguay, comparing the time required to find the optimal architecture using our approach with TensorFlow and PyTorch.

The remainder of this paper is structured as follows. Section 2 presents a brief introduction to the CUDA architecture, the neural networks used, and an explanation of our approach. Section 3 presents and discusses the results obtained. Lastly, Section 4 draws the main conclusions from our work.

2. Methodology

2.1. The CUDA architecture

Although Graphics Processing Units (GPUs) were initially created to accelerate graphical computation, their massively parallel GPU architecture greatly benefited many other general-purpose applications. Compute Unified Device Architecture (CUDA) was the first proposal of a language for General-Purpose GPU (GPGPU) made by NVIDIA for their graphics cards. The creation of this GPGPU language facilitated substantially the development of GPGPU applications as previously they had to be written through assembly or graphical APIs.

The CUDA language is an extension of the C/C++ language that adds additional syntax to indicate the operations the threads of the GPU should do. This is mainly done through special functions called "kernels" executed simultaneously by all threads used. Whenever a kernel is launched, the programmer must specify the number of blocks and the number of threads on each block that should execute the kernel. At a high level of granularity, all threads within the same block have additional advantages as they are executed on the same streaming multiprocessor. Each streaming multiprocessor has a unique set of cores, registers, cache memory, and a scheduler. This allows all threads within the same block to cooperate faster through the use of a programmed-managed part of the L1 cache memory called "shared memory" and a synchronization operation available for all threads of the block. If there is not enough work to use an entire streaming multiprocessor, the scheduler may run multiple blocks concurrently in the same streaming multiprocessor, even from different kernels. On the other hand, at the smallest level of granularity, the CUDA cores use a Single Instruction Multiple Threads (SIMT) architecture where a "warp" (32 contiguous threads) will always execute the same instruction. This means that in any instance in which the kernel code branches (e.g., if-else statements), the performance may be worse as it may require the entire warp to execute all branches before continuing with

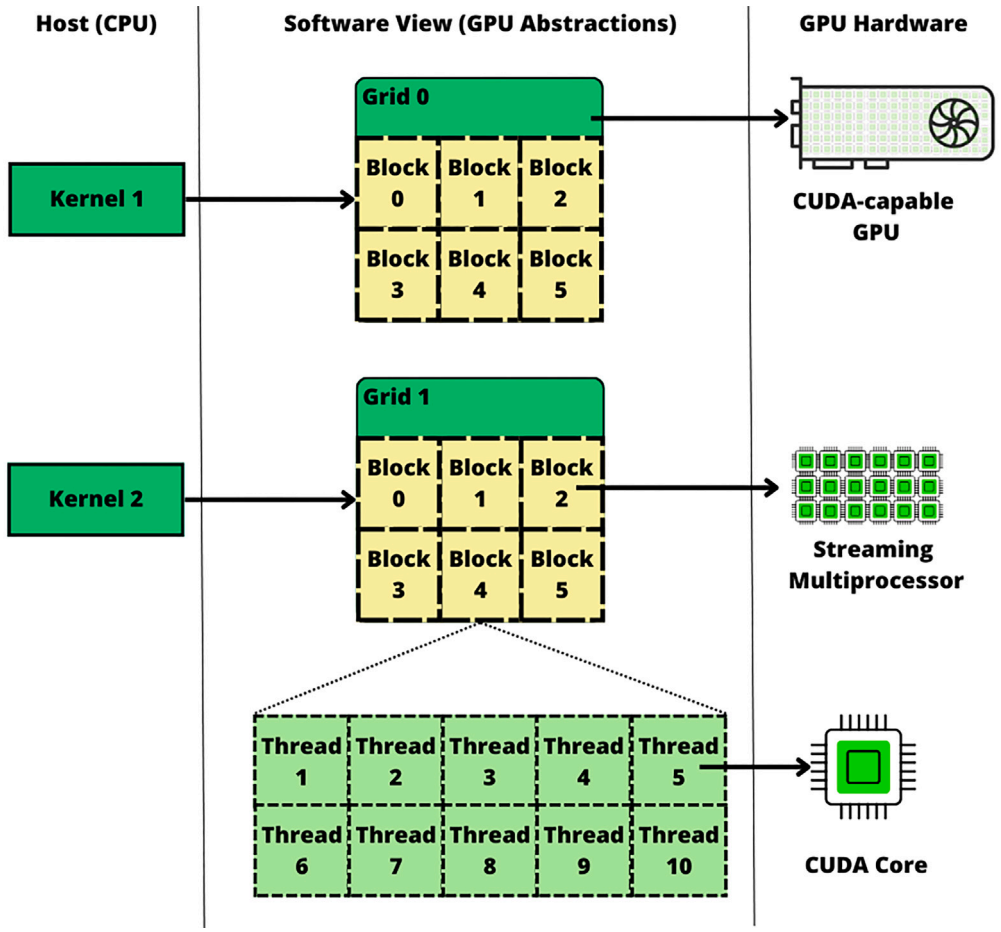


Fig. 1. Relationship between the CUDA software-level abstractions and the GPU hardware.

the following instruction and, as such, they should be avoided as long as possible. Fig. 1 illustrates the relationship between the abstractions utilized by the programmer and the corresponding hardware. When launching the kernel, the programmer specifies the grid (number of blocks and threads per block), and that grid is executed by an entire CUDA-capable GPU device. At a lower granularity level, each of the blocks that compose that grid will be executed in one of the Streaming Multiprocessors available in that GPU device. At the lowest level of granularity, each of the threads within the block will be executed on one of the CUDA cores available on the Streaming Multiprocessor assigned to its block.

One of the most common bottlenecks in GPU-based applications are slow memory accesses. Thus, understanding the GPU memory hierarchy is extremely important to ensure peak performance. Fig. 2 presents the memory hierarchy of the CUDA-capable GPU device employed in our experimentation. The figure is organized to showcase memory locations with the slowest access at the top, gradually progressing to those with the fastest access as we move downward. The slowest access occurs with data stored in the CPU/motherboard RAM, as it necessitates traversing the PCIe connection and traversing all memory locations within the GPU. Consequently, transfers of data between the CPU and GPU are minimized as much as possible. In fact, they are done only twice in many applications. The initial transfer occurs from the CPU/Motherboard to the GPU, facilitating the loading of all data necessary for computations, such as a dataset. The second transfer takes place after completing all computations, ensuring that the end user receives the computation results. This is essential since the output needs to reside in the CPU/Motherboard for the end user to view or store the output. The main on-chip memory on the GPU is the "global memory", serving as the principal storage location for data within the GPU. As such, it has the largest store capacity, but it is the slowest location inside the GPU. Data that needs to be exchanged between the CPU and the GPU or between multiple Streaming Multiprocessors (or blocks), must be stored into global memory. The next level in the memory hierarchy is the cache memory. There are two levels of cache memory (L1 and L2). The L2 cache is a slightly slower type of cache memory that has a larger storage capacity and it is shared across all streaming multiprocessors. The L1 cache is the fastest memory location besides registers, as it is local to each streaming multiprocessor. Thus, data in the L1 cache can only be accessed by threads within the same block, making it an ideal location in workflows that require shared memory access from multiple threads within the same block. In fact, programmers may specify within the kernel the amount of shared memory required and directly manage access to this memory without having to rely on compiler optimizations. Lastly, the use of registers is usually limited to the data that is required for the current computation. Nonetheless, the optimizer may select specific variables and small arrays local to a thread to store them in registers if sufficient space is available, removing the need of memory accesses until all computations with that data have finished.

2.2. Artificial neural networks (ANNs)

ANNs are computational models inspired by the human brain. They contain many computational nodes denominated "neurons" structured in layers. These neurons are interconnected with other neurons and each connection is associated with a weight. Each neuron computes the

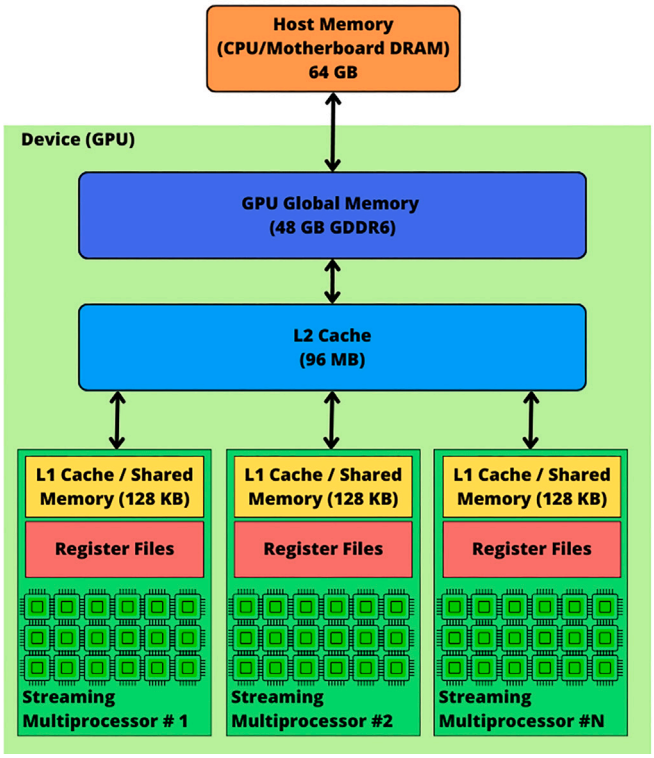


Fig. 2. A simplified representation of the CUDA memory hierarchy for the RTX 6000 Ada.

weighted sum of the outputs of the previous layers with the weights of the connections. Furthermore, a non-linear function is usually applied to the output of each neuron, allowing the neural network to learn non-linear relationships. During training, the connection weights are optimized to minimize a loss function between the outputs of the last layer and the desired values.

The Multi-Layer Perceptron (MLP) (Almeida, 1997) is a simple and widely used neural network. This architecture has one input layer, one or more hidden layers, and one output layer. In this architecture, each neuron j from a layer performs the weighted sum of the output x of all neurons from the previous layer i weighted by the weight of the connection $w_{(i,j)}$. After that, to obtain the final output, the bias of that neuron b_j is added and the activation function f is applied.

$$h_j = f(\sum_i w_{i,j} x_i + b_j). \tag{1}$$

The Elman neural network (Elman, 1990) is a Recurrent Neural Network (RNN) that includes a new kind of layer: the context layer. RNNs are capable of processing sequences of variable length through the use of recurrent connections between the neurons. In the Elman Neural Network, there will be as many context layers as hidden layers. Each context neuron copies the output of each hidden neuron, which will be used as additional input to the hidden layer along the context weights for the next element of the sequence. Mathematically, this can be expressed as follows:

$$h_j(t) = f(w_j x(t) + u_j h(t-1) + b_j). \tag{2}$$

where $h_j(t)$ is the output of the hidden neuron j for the element in position t of the sequence, w_j are the weight between the hidden neuron and all neurons of the previous layer, $x(t)$ are the output of this previous layer for the element t of the sequence, u_j are the recurrent weights between the context neurons and the neuron j, $h(t-1)$ are the hidden outputs for the previous element of the sequence and b_j is the bias of the hidden neuron.

LSTM neural networks (Hochreiter & Schmidhuber, 1997) are another RNN type that uses special neurons, denominated "LSTM cells" instead of hidden neurons. This type of neural network was created to solve the vanishing gradient problem in RNN, an issue that arises while training the neural network with backpropagation. The vanishing gradient occurs because the gradient must be passed through all time steps t of the sequence and the activation functions will squash the outputs to a limited range, usually between 0 and 1 (sigmoid), or between -1 and 1 (tanh). Therefore, for a long sequence, the repeated multiplication of a value below 1 will lead to a value closer and closer to 0, thus vanishing the gradient and making the neural network receive minimal to no updates in those scenarios. To solve this issue, LSTM neural networks incorporate two recurrent states: the hidden state h_t (for short-term memory) and the cell state $C(t)$ (for long-term memory). Alongside both states, the LSTM neural network also incorporates three gating mechanisms to regulate the information flow in the cell. The input gate $i(t)$, determines how much information from the current step in the sequence can be used to update the states. The forget gate $f(t)$ decides how much information from the previous cell state should be forgotten. Lastly, the output gate $o(t)$ decides how much of the current cell state is used to produce the hidden states. All of these gates have a set of weights $W_{i|f|o}$ to be learned and use a sigmoid activation function, limiting the range of each value of the gate from 0 (blocking information) to 1 (allowing all information through). Mathematically, an LSTM cell works as follows:

$$i_t|f_t|o_t = \sigma(W_{i|f|o} \cdot [h(t-1), x(t)] + b_{i|f|o}). \tag{3}$$

$$\widetilde{C}(t) = tanh(W_c \cdot [h(t-1), x(t)] + b_c). \tag{4}$$

$$C(t) = f_t \cdot C(t-1) + i_t \cdot \widetilde{C}(t). \tag{5}$$

$$h(t) = o(t) \cdot tanh(C(t)). \tag{6}$$

2.3. The proposed method

Fig. 3 shows the general idea of how the proposed method will run inside a GPU. Since we want to find the optimal hyperparameters, we developed one kernel that will train simultaneously multiple neural networks at once. The selection of only using one kernel was made to avoid the overhead of launching multiple kernels and the limitation provided by the fact that the number of concurrent kernels in execution may be lower than the number of streaming multiprocessors available. Thus, if we were to launch one kernel per neural network, 14 streaming multiprocessors would have remained completely idle during the entire training process with the GPU we used. In the kernel proposed in our method, each block will train a specific neural network, overcoming these limitations. Each thread will, for the most part, perform the computations related to one hidden neuron. If the number of threads per block is smaller than the number of hidden units, the kernel will do as many iterations as required to compute all the results from the hidden neurons.

Before running the kernel, it is essential to initialize and allocate most of the data structures in memory. Since many of these structures are accessed by the CPU and undergo storage and retrieval only once, they are allocated in global memory. This encompasses weights, biases, intermediate outputs, and non-recurrent gradients. Recurrent gradients are allocated in local arrays for each hidden neuron or thread, facilitating the utilization of registers when the dimension is sufficiently low. All these data structures are organized in row-major order and have dimensions in the following order: neural network size, batch size (for intermediate results data structures only), lags (for recurrent neural networks only) and hidden size. This is done to ensure that all threads access contiguous positions in memory, since every thread within the same block will need to access the same data structure in the position corresponding to its hidden neuron. Thus, this configuration

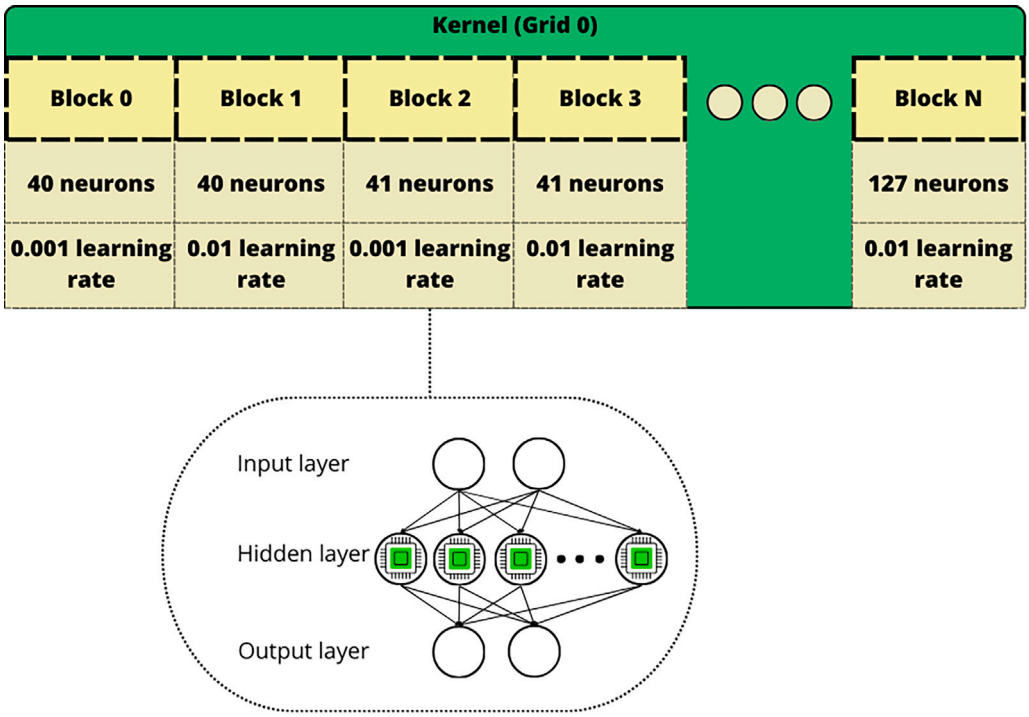


Fig. 3. Distribution in blocks and threads of the proposed method.

minimizes the number of memory accesses required to load data from other memory locations to the bare minimum.

After allocating these data structures, non-recurrent weights undergo initialization using the Xavier-Glorot method (Glorot & Bengio, 2010), where each thread handles one element of the data structure at the start of the kernel. Recurrent weights, on the other hand, are initialized through the orthogonal method using CuPy's (Nishino & Loomis, 2017) implementation of Singular Value Decomposition. Meanwhile, biases are initialized to 0, except for those associated with the forget gate of LSTMs, which are initialized to 1, following common practices. These initialization techniques are the default for TensorFlow and were used in all implementations to make the comparison fair.

Afterward, several arrays containing hyperparameters for each neural network are transmitted from the host to the GPU. In our implementation, there is an array for hidden sizes, another for learning rates, and the last one for activation functions. Each of these arrays will have as many values as neural networks need to be trained. Therefore, the position i in each array will indicate the value of its hyperparameter in the ith neural network. Additionally, an array containing a permutation per training epoch of the samples indexes is initialized using CuPy. This array is used to avoid having to shuffle the array in memory, thus allowing different neural networks to progress at a different pace. Once this initialization progress is finished, the kernel will iterate over each epoch and each sample of a batch.

In Fig. 4, the workflow of all threads within the same block is illustrated, showing the tasks undertaken to process an entire batch. First, each thread will do all the computations to compute the hidden output of a neuron, storing the results in the corresponding array for intermediate values. These computations are the weighted sum of the output of the previous layer and the activation function. A synchronization barrier is placed afterward to ensure that all hidden outputs have been computed before proceeding to the next step. In the case of RNNs, this first step is repeated until all lags from the input sequence have been computed. Then, after the last synchronization is done, the Harris' (Harris et al., 2007) parallel reduction is used with the final hidden states and the weights of the output layer to compute each output neuron's output efficiently. At this step, we start computing the loss for this sample with one thread per time step (output neuron) forecast, storing the loss for that sample in an array. After a synchronization, each thread computes the loss at the hidden layer by backpropagating the loss according to the chain rule with the loss of the output layer, the output weights and the activation. Finally, in RNNs, this process is repeated for all time steps and the next sample inside the batch is processed.

After processing all samples within a batch, the weights and biases are updated using the ADAM algorithm (Kingma & Ba, 2015). This update is performed with the desired learning rate, using the previously computed backpropagated loss and any other required intermediate values and weights. During this process, each thread is responsible for updating one neuron, and no synchronization is needed since all necessary computations have been previously executed and stored, mitigating any potential race conditions.

In the case of RNNs, additional local arrays are employed to store recurrent gradients. These gradients pertain to the connections between a hidden neuron and the context layer in the Elman network and between a hidden LSTM unit and all recurrent connections through the gates in the LSTM network. This design allows these recurrent gradients to potentially be stored in registers if the number of hidden neurons in the neural network is small enough.

Upon completing the weight and bias updates, the CUDA block's neural network can proceed to process the next batch without waiting for all other neural networks to complete processing the same batch since our approach does not require shuffling the samples in memory thanks to the use of the data structure with the permuted indexes. The source code used for this project can be found at https://github.com/xkuzz/MultiNNCuda.

3. Results

3.1. Experimental setup

To assess the efficacy of our approach, we conducted a comparative analysis against TensorFlow's and PyTorch's implementations on two energy demand datasets, as outlined in the following subsection.

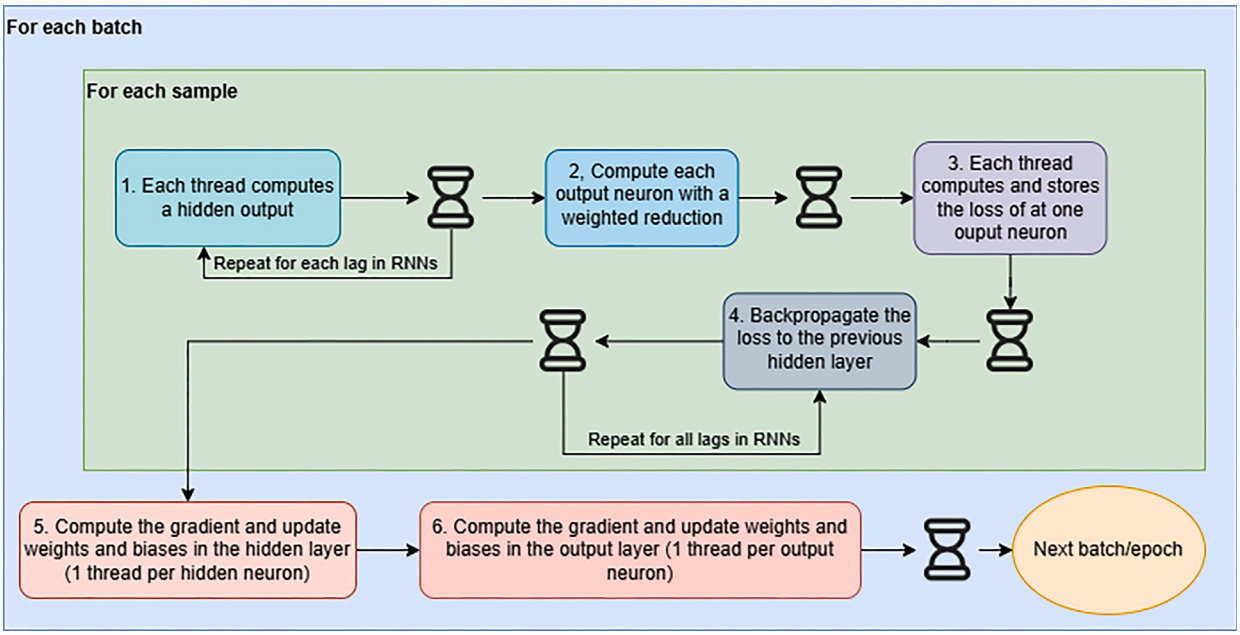


Fig. 4. A visual representation of the work done by the threads inside a block to train a neural network.

Table 1
List of hyperparameters considered per ANN architecture in the grid search procedure.

Hyperparameter	Range
Hidden Neurons/Units	$\{40, 41, 42, \ldots, 127\}$
Learning Rate	$\{0.05, 0.01, 0.001, 0.0001\}$

It is worth emphasizing that these frameworks use distinct implementations based on certain constraints and the ANN architecture employed. Consequently, we divided our experiments into three distinct sets, each corresponding to a unique architecture with a specific activation function (ReLU for MLP, tanh for Elman and LSTM RNNs). These activation functions were selected in a preliminary study done with TensorFlow, where we studied which activation function provided the best results for each ANN architecture.

In each experiment, 348 ANN configurations were trained during 10 epochs using a grid search approach, with the hyperparameters specified in Table 1 in all implementations. These numbers were selected to have a reasonable degree of variety in learning rate and number of hidden neurons that are still within a reasonable boundary to work well. These boundaries were selected to be neither too small nor too big for the size of the problem studied based on some preliminary studies with TensorFlow. Each experiment with the same 348 ANN configuration was repeated per batch size studied (1, 2, 4, 8, 16, 32 and 64), ANN architecture, implementation and dataset.

The experimental setup used was comprised of a private cloud server with two GPU nodes, two Xeon 4310 CPUs and 64 GB of RAM. Each GPU node had an NVIDIA RTX A6000 ADA with 48 GB GDDR6 global memory and 18176 CUDA cores. In the case of our implementation, a kernel with half of the hyperparameters was sent to each GPU and in the case of TensorFlow, we evaluated two different approaches. In the first one (from now on, denominated "TF-A"), TensorFlow was allowed to use the full potential of a GPU to train a neural network as fast as possible. Thus, two neural networks were being trained as fast as possible at once (one on each GPU). However, since this was not fully using all the resources of the GPU, we also evaluated another approach (from now on denominated "TF-B"), in which we tried to train as many neural networks as possible simultaneously. We did this by training 14 neural networks simultaneously between both GPUs (7 on each) in a multi-process approach, as adding more would create a bottleneck in RAM memory, significantly slowing the training process. This last approach was also used to evaluate the other mainstream ANN framework, PyTorch, as it was substantially faster than the other approach for the cases studied.

3.2. Datasets description

Two datasets were used to evaluate the proposed method: one containing energy consumption data from Spain and another with energy demand data from Uruguay.

The Spanish dataset includes energy consumption records from January 1, 2007, to the present. This data was scraped from the Spanish energy operator, initially with a 10-minute granularity, later updated to a 5-minute granularity. While the dataset also provides additional information on market prices, emissions, and energy generation, only the energy consumption data was utilized, adhering to the preprocessing pipeline outlined in Torres et al. (2022). This pipeline considers data up to June 2016, with input sequences comprising 168 lags and a forecast horizon of 24 observations.

The Uruguayan dataset provides hourly energy consumption data aggregated from several smart meters from January 1, 2007, to December 31, 2014. In addition to energy consumption, it includes information on temperature and holidays. Similar to the Spanish dataset, this study focuses exclusively on energy consumption, treating it as a univariate time series, as done in Pérez-Chacón, Asencio-Cortés, Martínez-Álvarez, and Troncoso (2020).

For both datasets, the data was normalized to the range $[0, 1]$ using min–max normalization. The datasets were split into three partitions while preserving chronological order. The first 70% of the data was allocated for training and validation, with the remaining 30% reserved for testing, which was used to generate the results presented in Section 3.5. The validation set constituted the last 30% of the training partition and was employed to select the best-performing architecture.

3.3. Metrics used

To measure the execution performance of each approach, we measure the total execution time of each algorithm per experiment (one neural network architecture with a specific batch size). Additionally,

Table 2
Execution times (in seconds) and speedups between the studied approaches for the Spanish dataset.

Architecture	Batch size	Proposed approach	TF-A	TF-B	PyTorch	Speedup vs TF-A	Speedup vs TF-B	Speedup vs PyTorch
MLP	64	2.36	1806.79	585.39	188.10	765.70	248.08	79.71
	32	2.40	3075.54	799.10	275.98	1281.08	332.86	114.96
	16	2.76	5895.01	1212.89	461.40	2139.24	440.15	167.43
	8	3.75	11 435.87	2065.09	821.46	3051.88	551.11	219.22
	4	5.96	22 626.11	3751.20	1517.55	3794.43	629.08	254.49
	2	11.19	38 171.32	6153.01	2580.76	3411.81	549.96	230.67
	1	21.39	74 304.23	11 664.54	4452.37	3474.47	545.43	208.19
Elman	64	147.21	25 969.14	4049.37	301.89	176.41	27.51	2.05
	32	146.36	51 806.80	7676.64	493.05	353.98	52.45	3.37
	16	142.92	103 078.45	14 977.93	905.34	721.25	104.80	6.33
	8	139.46	205 481.84	29 390.57	1686.71	1473.38	210.74	12.09
	4	127.27	223 986.12	58 379.09	3205.68	1759.95	458.71	25.19
	2	117.05	416 869.05	69 425.95	6192.48	3561.43	593.13	52.90
	1	121.27	831 297.82	138 391.29	11 718.23	6854.91	1141.18	96.63
LSTM	64	1641.00	5185.25	1666.22	410.44	3.16	1.02	0.25
	32	1628.41	9809.22	2762.19	680.05	6.02	1.70	0.42
	16	1629.18	18 753.44	4867.63	1214.19	11.51	2.99	0.75
	8	1629.51	37 053.42	8978.58	2260.94	22.74	5.51	1.39
	4	1638.66	72 995.02	17 236.88	4332.37	44.55	10.52	2.64
	2	1646.03	134 860.80	31 196.69	8379.62	81.93	18.95	5.09
	1	1657.07	244 583.40	44 268.82	16 130.56	147.60	26.72	9.73

we measured the speedup obtained between our implementation and TensorFlow approaches. Regarding the accuracy metrics, the following metrics were used as they are utilized frequently for time series forecasting.

The Mean Absolute Error (MAE) measures the average absolute difference between the predicted and expected values.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|. \tag{7}$$

The Mean Absolute Percentage Error (MAPE) is a measure that represents the MAE as a percentage according to the following formula.

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{y_i - \hat{y}_i}{y_i}. \tag{8}$$

Lastly, the Root Mean Squared Error (RMSE) is a metric that gives more weight to large errors, punishing harder forecast values far away from the expected values, but also making it heavily influenced by outliers.

$$RMSE = \frac{1}{N}\sqrt{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}. \tag{9}$$

For all these metrics, N represents the total number of observations of all samples, $\hat{y}$ represents the predicted value, $\bar{y}$ is the average of the observations and y represents the expected value. For all metrics except R^2, a lower value indicates a better forecast.

3.4. Speedup analysis

Table 2 presents the time (in seconds) required to train all 348 ANN configurations across different architectures, batch sizes, and implementations. As expected, our approach demonstrated a greater speedup with smaller batch sizes.

For the MLP architecture, our approach consistently outperformed mainstream frameworks, making it one of the ideal scenarios for its use. Due to the simplicity of this ANN architecture, it is difficult to fully utilize the GPU's resources unless an extremely large batch size is used, or a large number of neural networks are trained simultaneously. Our approach delivered the fastest training times among all implementations, taking only 2.36 s for a batch size of 64 and 21.39 s for a batch size of 1. Depending on the batch size, the speedup ranged from 248 to 545 times compared to TF-B, and from 79 to 254 times compared to PyTorch.

As it could be expected, the use of TF-A, representing the classic use of Tensorflow, where each neural network is trained using the full potential of 1 GPU, led to extremely slow training regardless of the ANN architecture, with a worse performance the lower the batch size, as the number of operations that could be done in parallel would be even smaller. Additionally, the PyTorch equivalent of TF-B consistently outperformed its TensorFlow counterpart, reliably training multiple models substantially faster regardless of the ANN architecture.

For the Elman architecture, due to the complex nature of the recurrent connection, the time required to train the model with all implementations was slower than the ones required to train MLPs, as the inclusion of time-step dependencies involves a higher number of operations and a mandatory synchronization before processing the next time step. In particular, our approach offers a relatively fast training time, between 117 and 147 s, while the best approach using TensorFlow is between 27 and 1141 times slower, depending on the batch size used, and the best approach using PyTorch is between 2 and 96 times slower.

For the LSTM architecture, PyTorch's and ours implementation are slower than for the Elman architecture, due to its more complex nature. However, this is not the case for the TensorFlow implementation, where LSTMs are faster than Elman networks. This performance difference is due to TensorFlow's use of a highly optimized CUDNN implementation for LSTM networks (provided by GPU manufacturers) as long as certain conditions are met, such as using the hyperbolic tangent activation function. This same implementation is also used by the PyTorch library for the computations of the LSTM layer.

Although one might expect both implementations to have similar training times, the differences in how the frameworks operate, such as PyTorch's native support for asynchronous operations, result in PyTorch being faster than TensorFlow. It should also be noted that, while the optimizations presented in Appleyard et al. (2016) are expected to be utilized, the full implementation details are not publicly available. Therefore, for this architecture, we observe results that are closer to our approach, with some cases involving larger batch sizes where our implementation is slower than PyTorch. For this architecture, our approach was up to 26.72 time faster than TensorFlow and up to 9.73 times faster than PyTorch, although it was slower than PyTorch if the batch size was 16 or higher.

Table 3
Detailed breakdown of execution time for the REE dataset.

Architecture	Batch size	Data initialization/Transfer (s)	Training (s)
MLP	64	0.035	2.31
	32	0.035	2.35
	16	0.035	2.76
	8	0.034	3.71
	4	0.033	5.90
	2	0.033	11.13
	1	0.037	21.46
Elman	64	0.927	144.92
	32	0.964	144.48
	16	0.932	141.07
	8	0.862	134.80
	4	0.890	124.22
	2	0.811	114.57
	1	0.988	119.04
LSTM	64	3.15	1903.00
	32	3.14	1910.45
	16	3.18	1910.76
	8	3.07	1906.01
	4	3.16	1898.40
	2	3.39	1918.22
	1	3.04	1967.17

In general, as the batch size increases, it is expected that our approach will provide lower speedup. This is because, in our approach, each element of the batch is processed sequentially, and larger data structures are required, leading to more cache misses. Therefore, there will be a point where the performance of our approach falls behind that of the TensorFlow and PyTorch implementations. This breakpoint will be reached as complexity increases, either due to a larger batch size or a more complex ANN topology, at which point our approach may become slower. Therefore, the GPU specification, datasets, number of configurations to evaluate and batch size are crucial factors to determine whether our approach or using PyTorch would be more effective.

Table 3 provides a detailed breakdown of the time spent on data transfer, initialization of data structures, and model training in our approach. As expected, the majority of time is devoted to model training across all architectures, as it is significantly more computationally expensive than data initialization and transfer. Notably, the time required for initialization and data transfer to the GPU remains relatively consistent across different batch sizes. However, a substantial difference emerges between architectures, ranging from 35 ms for the MLP architecture to approximately 3.15 s for the LSTM architecture. This increase is attributed to the LSTM's larger and more complex data structures, as well as the use of orthogonal initialization. The data in this table was collected using the *nsys* profiling tool, with two *nvtx* ranges to separate the initialization and data transfer phases from the model training kernel covering the second time the method is run (the first is a warm-up round to make sure that the CuPy compilation of the kernel does not influence the result). Full profiling dumps are available at https://osf.io/r7djh/.

To complement the study presented in the previous tables, Table 4 shows the same set of experiments conducted on the Uruguayan dataset, with the exception of TF-A, which was excluded as it is consistently slower than TF-B. Although this dataset comprises more years, it has only hourly granularity, resulting in fewer samples compared to the previous dataset. Consequently, training times are shorter and our approach works slightly better, leading to bigger speedups. Nonetheless, all the major conclusions drawn from the Spanish dataset still hold true: our approach is consistently faster for MLP and Elman architectures, PyTorch is consistently faster than TensorFlow, and for the LSTM architecture, PyTorch performs slightly better at the largest batch sizes studied, while our approach achieves up to a 10x speedup at the lowest batch size.

3.5. Analysis of batch size impact in forecast accuracy

At last, we compare the results in terms of the accuracy of each model. A first point of interest is to evaluate the impact the batch size has had on each of these architectures, as its optimal size has a major impact on the usefulness of each implementation. Fig. 5 shows the evolution of all metrics used as the batch size grows on each architecture for the Spanish dataset. As it can be observed from this figure, regardless of the architecture, the use of a smaller batch size leads to the best results, with the best model always obtained through the use of a batch size of 1 or 2. The use of larger batch sizes (32 and 64) led to a less accurate model, particularly in the case of the LSTM, where higher batch sizes provided worse models than the simple MLP architecture with similar batch size. It should be remarked that, for all metrics, the LSTM architecture usually delivered the most accurate forecast, closely trailed by the MLP architecture.

Fig. 6 shows the evolution of metrics as batch size increases for the Uruguayan dataset. In this case, the results differ slightly. This is primarily because RNNs perform significantly worse on this dataset, likely due to the smaller number of available samples. However, the best model for the MLP architecture is still achieved with a relatively small batch size, although most results are fairly close with the exception of the largest batch size with the LSTM network, which performs much worse than the others.

3.6. Advantages and limitations of the proposed approach

One of the major advantages of using the proposed approach is how much faster we can find the optimal hyperparameters for a neural network, as it was shown in Section 3.4. This has many advantages, as it allows researchers and practitioners do to a more exhaustive search to find the optimal model in the lowest amount of time possible. Furthermore, depending on the application for which they are used, some other advantages may arise. For example, in our energy forecasting case study, the training could be done in a cloud service. Then, once trained, the models could be sent to edge devices or smart meters that can be used for inference purposes without the need to train an individual model per smart meter with limited resources and facilitating its use in any other advanced analytics provided to the customer at the edge (energy disaggregation, demand response, pricing, recommender systems, etc.).

However, even though the proposed approach should work greatly in a large number of applications, the major drawback of the proposed approach is how it scales as the complexity of the datasets and neural networks rises. This is mainly due to the fact that more complex datasets will usually require neural networks with a larger number of trainable parameters (i.e., computer vision problems) and there will be a breakpoint where the batch size and the number of trainable parameters per layer is large enough that GEMM-based approaches can use optimally all the GPU resources or we cannot fit in the GPU memory all of the data structures required for our approach. In those cases, the highly optimized GEMM-based approaches available in frameworks like TensorFlow or PyTorch should be preferred. Nevertheless, there will still be some instances in which depending on the data, the number of neural networks evaluated and their complexity, it may still be more beneficial to use our approach multiple times with a reduced number of neural networks trained simultaneously in order to fit them in memory.

4. Conclusion

This paper presented a novel approach to train simultaneously multiple neural networks with different hyperparameters in parallel with the GPU, allowing researchers and practitioners to quickly find the optimal topology for a neural network model. The proposed method was evaluated with three different neural network architectures (MLP, Elman and LSTM) using energy demand data from Spain and Uruguay.

Table 4
Execution times (in seconds) and speedups between the studied approaches for the Uruguayan dataset.

Architecture	Batch size	Proposed approach	TF-B	PyTorch	Speedup vs TF-B	Speedup vs PyTorch
MLP	64	0.64	343.40	133.48	536.56	208.56
	32	0.35	373.73	143.68	1067.8	410.51
	16	0.40	430.40	139.83	1076	349.58
	8	0.54	540.72	183.44	1001.33	339.70
	4	0.84	762.29	292.63	907.49	348.37
	2	1.58	1204.38	511.25	762.27	323.58
	1	2.99	1872.50	678.46	626.25	226.91
Elman	64	21.88	872.61	147.21	39.88	6.73
	32	20.80	1418.62	157.18	68.20	7.56
	16	20.43	2434.45	205.57	119.16	10.06
	8	19.74	4571.34	294.42	231.58	14.91
	4	18.26	8696.95	517.69	476.28	28.35
	2	17.03	17 317.70	953.76	1016.89	56.00
	1	17.35	20 204.14	1753.18	1164.50	101.04
LSTM	64	231.13	487.29	165.25	2.11	0.71
	32	230.64	631.06	197.33	2.74	0.85
	16	230.47	923.17	245.04	4.01	1.06
	8	230.86	1495.64	386.07	6.48	1.67
	4	231.45	2618.77	685.38	11.31	2.96
	2	231.61	4894.78	1274.49	21.13	5.50
	1	233.94	6483.59	2392.55	27.74	10

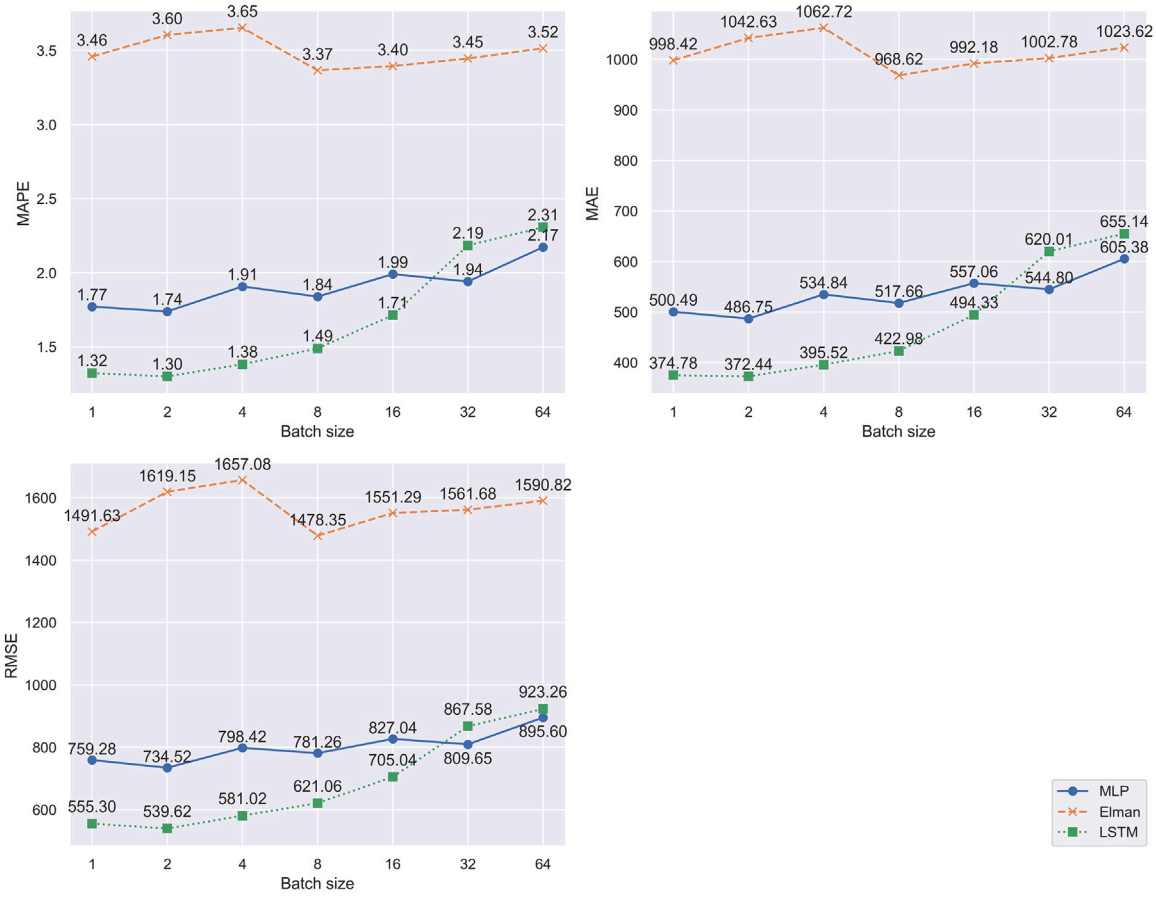


Fig. 5. Evolution of metrics with batch size for the Spanish dataset.

The developed implementation was compared against two mainstream ANN frameworks in terms of training time, TensorFlow and PyTorch. Furthermore, we evaluated each neural network architecture with different batch sizes, allowing us to study the impact of batch size selection in accuracy metrics and allowing us to see the evolution in speedup as the batch size increases. After evaluating the developed implementation we have learned that:

- It was faster to train multiple neural networks with our implementation than using other approaches until reaching a breakpoint in which a neural network may be so big that either all resources

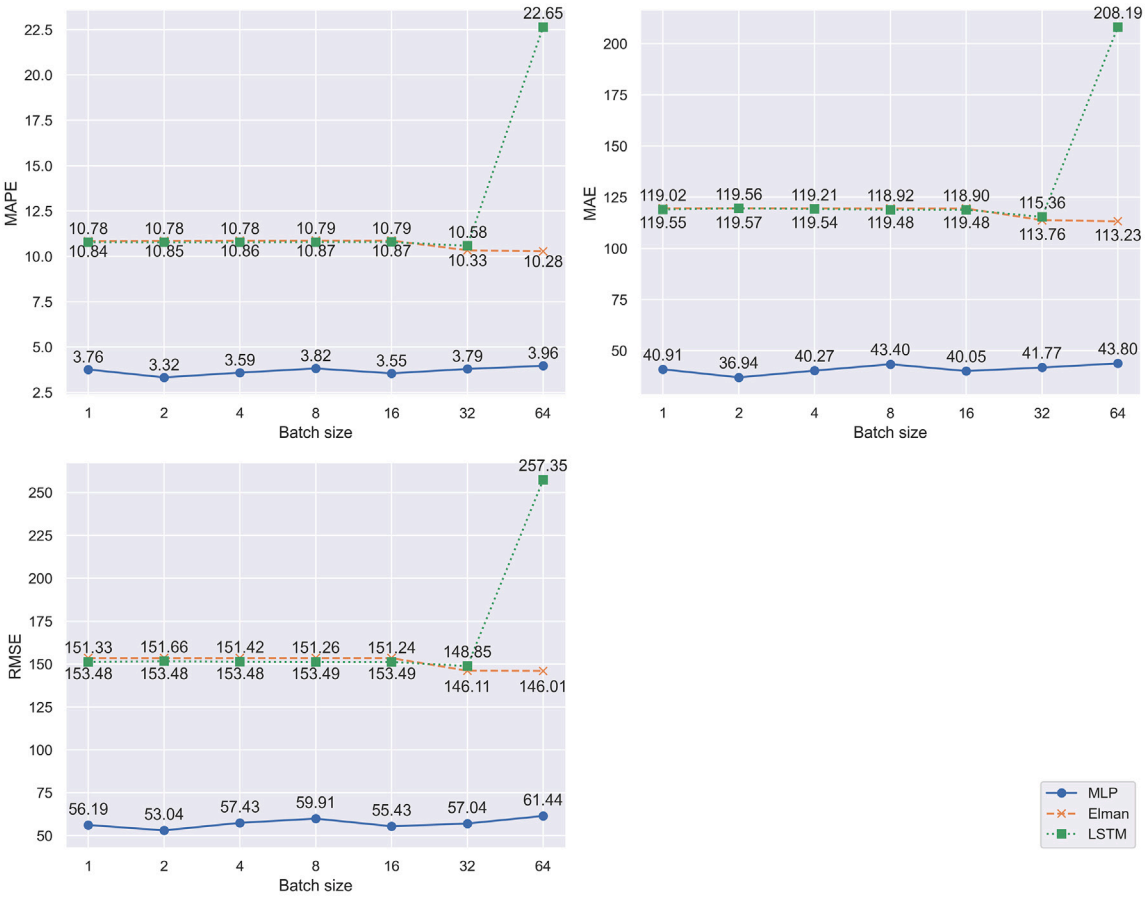


Fig. 6. Evolution of metrics with batch size for the Uruguay dataset.

of the GPU are already used to train one neural network or the data structures for multiple neural networks no longer fit in the GPU memory. This interplay between speedup and complexity can be seen through the comparison of batch sizes, as a larger batch size implies larger data structures and a higher amount of computations that can be done in parallel to train just one neural network.

- The most accurate models across each neural network architecture were generally achieved with lower batch sizes.

The implementation presented in this paper provided an exceptional training speed, yielding results that were up to 3400 times faster than conventional methods using TensorFlow. This remarkable advantage positions our implementation as an ideal choice for scenarios akin to the one examined in this study, where the number of input features is relatively modest. This will usually be the case for most tabular datasets and many time series applications. However, the main limitation of our approach is that it does not scale well in scenarios with larger amounts of data. This implies that our implementation may not be optimal for applications characterized by a vast amount of data, such as those found in Computer Vision or Natural Language Processing. In these instances, where one neural network saturates most of the GPU's resources, the TensorFlow implementation excels as it was designed specifically for that use case. Consequently, the performance of our proposed implementation is contingent on hardware specifics and data volume. Thus, the closer we are to using all CUDA cores or all fast memory locations with just one neural network, the worse our implementation will work. Nonetheless, the proposed implementation will still be the best choice for a large number of applications that do not require processing massive amounts of data simultaneously.

Future works may consider the development and evaluation of parallelized algorithms to guide the hyperparameter search (i.e., metaheuristic algorithms) or extend the methodology to other neural network architectures.

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
LSTM	Long-Short Term Memory
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
ML	Machine Learning
MLP	Multi Layer Perceptron
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network

CRediT authorship contribution statement

D. Criado-Ramón: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **L.G.B. Ruiz:** Methodology, Writing – original draft, Writing – review & editing, Supervision. **M.C. Pegalajar:** Conceptualization, Methodology, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We acknowledge financial support from Ministerio de Ciencia e Innovación (Spain) (Grant PID2020-112495RB-C21 funded by MCIN/AEI /10.13039/501100011033).

Data availability

Data will be made available on request.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2016). TensorFlow: Large-scale machine learning on heterogeneous distributed systems. http://dx.doi.org/10.48550/arXiv.1603.04467, arXiv:1603.04467.

Almeida, L. B. (1997). Multilayer perceptrons. In *Handbook of neural computation*. IOP Publishing Ltd and Oxford University Press.

Alshemali, B., & Kalita, J. (2020). Improving the reliability of deep neural networks in NLP: A review. *Knowledge-Based Systems*, *191*, Article 105210. http://dx.doi.org/10.1016/j.knosys.2019.105210.

Appleyard, J., Kociský, T., & Blunsom, P. (2016). Optimizing performance of recurrent neural networks on GPUs. http://dx.doi.org/10.48550/arXiv.1604.01946, CoRR abs/1604.01946.

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 785–794). New York, NY, USA: ACM, http://dx.doi.org/10.1145/2939672.2939785.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., et al. (2014). cuDNN: Efficient primitives for deep learning. http://dx.doi.org/10.48550/arXiv.1410.0759, CoRR abs/1410.0759.

Elman, J. (1990). Finding structure in time. *Cognitive Science*, *14*, 179–211. http://dx.doi.org/10.1016/0364-0213(90)90002-E.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256). JMLR Workshop and Conference Proceedings.

Harris, M., et al. (2007). Optimizing parallel reduction in CUDA. *Nvidia Developer Technology*, *2*(4), 70.

Hewamalage, H., Bergmeir, C., & Bandara, K. (2021). Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting*, *37*(1), 388–427. http://dx.doi.org/10.1016/j.ijforecast.2020.06.008.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*, 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Iruela, J., Ruiz, L., Pegalajar, M., & Capel, M. (2020). A parallel solution with GPU technology to predict energy consumption in spatially distributed buildings using evolutionary optimization and artificial neural networks. *Energy Conversion and Management*, *207*, Article 112535. http://dx.doi.org/10.1016/j.enconman.2020.112535.

Jang, H., Park, A., & Jung, K. (2008). Neural network implementation using CUDA and OpenMP. In *2008 digital image computing: techniques and applications* (pp. 155–161). http://dx.doi.org/10.1109/DICTA.2008.82.

Jin, N., Yang, F., Mo, Y., Zeng, Y., Zhou, X., Yan, K., et al. (2022). Highly accurate energy consumption forecasting model based on parallel LSTM neural networks. *Advanced Engineering Informatics*, *51*, Article 101442. http://dx.doi.org/10.1016/j.aei.2021.101442.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., et al. (2017). Lightgbm: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems*, *30*, 3146–3154.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). On large-batch training for deep learning: Generalization gap and sharp minima. In *5th international conference on learning representations, ICLR 2017, toulon, France, April 24-26, 2017, conference track proceedings*.

Kim, M., Choi, W., Jeon, Y., & Liu, L. (2019). A hybrid neural network model for power demand forecasting. *Energies*, *12*(5), http://dx.doi.org/10.3390/en12050931.

Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In Y. Bengio, & Y. LeCun (Eds.), *3rd international conference on learning representations, ICLR 2015, san diego, CA, USA, May 7-9, 2015, conference track proceedings*.

Kintsakis, A. M., Chrysopoulos, A., & Mitkas, P. A. (2015). Agent-based short-term load and price forecasting using a parallel implementation of an adaptive PSO-trained local linear wavelet neural network. In *2015 12th international conference on the European energy market* (pp. 1–5). http://dx.doi.org/10.1109/EEM.2015.7216611.

Lu, C., Li, S., & Lu, Z. (2022). Building energy prediction using artificial neural networks: A literature survey. *Energy and Buildings*, *262*, Article 111718. http://dx.doi.org/10.1016/j.enbuild.2021.111718.

Luo, X., & Oyedele, L. O. (2021). Forecasting building energy consumption: Adaptive long-short term memory neural networks driven by genetic algorithm. *Advanced Engineering Informatics*, *50*, Article 101357. http://dx.doi.org/10.1016/j.aei.2021.101357.

Manno, A., Martelli, E., & Amaldi, E. (2022). A shallow neural network approach for the short-term forecast of hourly energy consumption. *Energies*, *15*(3), http://dx.doi.org/10.3390/en15030958.

Maragkos, N., Tzelepi, M., Passalis, N., Adamakos, A., & Tefas, A. (2022). Electric load demand forecasting on greek energy market using lightweight neural networks. In *2022 IEEE 14th image, video, and multidimensional signal processing workshop* (pp. 1–5). http://dx.doi.org/10.1109/IVMSP54334.2022.9816189.

Nassif, A. B., Shahin, I., Attili, I., Azzeh, M., & Shaalan, K. (2019). Speech recognition using deep neural networks: A systematic review. *IEEE Access*, *7*, 19143–19165. http://dx.doi.org/10.1109/ACCESS.2019.2896880.

Nishino, R., & Loomis, S. H. C. (2017). Cupy: A numpy-compatible library for nvidia gpu calculations. In *31st Confernce on Neural Information Processing Systems: vol. 151*, (no. 7).

NVIDIA (2023). cuBLAS. https://docs.nvidia.com/cuda/cublas/. (Accessed: 01 Nov 2024).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., et al. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems: vol. 32*, (pp. 8024–8035). Curran Associates, Inc.

Pérez-Chacón, R., Asencio-Cortés, G., Martínez-Álvarez, F., & Troncoso, A. (2020). Big data time series forecasting based on pattern sequence similarity and its application to the electricity demand. *Information Sciences*, *540*, 160–174. http://dx.doi.org/10.1016/j.ins.2020.06.014, URL https://www.sciencedirect.com/science/article/pii/S0020025520306010.

Raschka, S., Patterson, J., & Nolet, C. (2020). Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, *11*(4), http://dx.doi.org/10.3390/info11040193.

Rick, R., & Berton, L. (2022). Energy forecasting model based on CNN-LSTM-AE for many time series with unequal lengths. *Engineering Applications of Artificial Intelligence*, *113*, Article 104998. http://dx.doi.org/10.1016/j.engappai.2022.104998.

Ting, T. O., Ma, J., Kim, K. S., & Huang, K. (2016). Multicores and GPU utilization in parallel swarm algorithm for parameter estimation of photovoltaic cell model. *Applied Soft Computing*, *40*, 58–63. http://dx.doi.org/10.1016/j.asoc.2015.10.054.

Torres, J., Martínez-Álvarez, F., & Troncoso, A. (2022). A deep LSTM network for the spanish electricity consumption forecasting. *Neural Computing and Applications*, *34*(13), 10533–10545. http://dx.doi.org/10.1007/s00521-021-06773-2.

Uetz, R., & Behnke, S. (2009). Large-scale object recognition with CUDA-accelerated hierarchical neural networks. In *2009 IEEE international conference on intelligent computing and intelligent systems: vol. 1*, (pp. 536–541). http://dx.doi.org/10.1109/ICICISYS.2009.5357786.

Voulodimos, A., Doulamis, N., Doulamis, A., Protopapadakis, E., et al. (2018). Deep learning for computer vision: A brief review. *Computational Intelligence and Neuroscience*, *2018*, http://dx.doi.org/10.1155/2018/7068349.

Wan, A., Chang, Q., AL-Bukhaiti, K., & He, J. (2023). Short-term power load forecasting for combined heat and power using CNN-LSTM enhanced by attention mechanism. *Energy*, *282*, Article 128274. http://dx.doi.org/10.1016/j.energy.2023.128274.

Wang, L., Zhang, Z., Huang, C., & Tsui, K. L. (2018). A GPU-accelerated parallel jaya algorithm for efficiently estimating li-ion battery model parameters. *Applied Soft Computing*, *65*, 12–20. http://dx.doi.org/10.1016/j.asoc.2017.12.041.

Yan, K., Li, W., Ji, Z., Qi, M., & Du, Y. (2019). A hybrid LSTM neural network for energy consumption forecasting of individual households. *IEEE Access*, *7*, 157633–157642. http://dx.doi.org/10.1109/ACCESS.2019.2949065.

Zhou, H., Zhang, Y., Yang, L., Liu, Q., Yan, K., & Du, Y. (2019). Short-term photovoltaic power forecasting based on long short term memory neural network and attention mechanism. *IEEE Access*, *7*, 78063–78074. http://dx.doi.org/10.1109/ACCESS.2019.2923006.

Zhuo, Y., Zhang, T., Du, F., & Liu, R. (2023). A parallel particle swarm optimization algorithm based on GPU/CUDA. *Applied Soft Computing*, *144*, Article 110499. http://dx.doi.org/10.1016/j.asoc.2023.110499.