Parallelization of a Neural Network Algorithm for Handwriting Recognition: Can we Increase the Speed, Keeping the Same Accuracy

D. Todorov*, V. Zdraveski*, M. Kostoska* and M. Gusev*
* Ss. Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering, Skopje, North Macedonia
E-mail: david.todorov@students.finki.ukim.mk, { vladimir.zdraveski, magdalena.kostoska, marjan.gushev }@finki.ukim.mk

***Abstract*—This paper examines the problem of parallelizing neural network training. For our solution we use the back-propagation neural network, as a breakthrough example in the field of deep learning. The challenge of our solution is to twist the algorithm in such a way so it can be executed in parallel, rather than sequentially. In this paper we would like to test validity of a research hypothesis if we can increase the speed by parallelizing the back-propagation algorithm and keep the same accuracy. For this purpose we will develop a use-case of a handwriting recognition algorithm and run several experiments to test the performance, both in execution speed and accuracy. At the end we are going to examine just how much it benefits us to try and write a parallel program for a neural network, with regards to the time it takes to train the neural network and the accuracy of the predictions. Our handwriting problem is that of classification, and in order to implement any sort of solution, we must have data. The MNIST dataset of handwritten digits will provide our necessary data to solve the problem.**

***Index Terms*—message passing interface, neural network, handwriting recognition, multilayer perceptron, parallel processing, distributed processing**

I. Introduction

We live in a world where everyone has a smartphone at their disposal and can write anything that comes to mind on it. Even so, that world is still, and will continue to be filled with handwritten notes, texts and descriptions. The prime example are schools where students write their notes in paper notebooks with pens, rather than on their smartphones with their fingertips. Some students are naturally good at organizing their handwritten notes, but some prefer them to be in a digital format in which they could store them in a cloud, manipulate them etc. Instead of having to rewrite all of their notes, with an app for handwriting recognition one could just scan or take a picture of a page of a notebook, feed it to a piece of software would be able to process the image and give the written text as output. Said output could then be processed and stored as per the user's desire.

The prime tool that would enable us to build such a system is Artificial Intelligence (AI). Using machine learning (ML), researchers have actively been developing new ways and refining old ways of recognising handwritten text. One of, if not the biggest breakthrough in the field of ML is the algorithm known as the multilayer perceptron, more commonly called a neural network (NN). The neural network is an algorithm that works similarly to the human brain and enables a computer to solve problems that would be almost impossible to develop ad hoc solutions for.

Of course, computer scientists would like to speed up and optimize the process, if even just a little bit. The idea for optimizing the training process of the neural network (NN) is to run it in parallel on multiple CPU cores, or even (if the hardware is present) on multiple workstations. Splitting up the work among more processes, the idea is for the program to do more work in a finite amount of time, than it would've done if it ran sequentially, thus speeding up the execution. Naturally, we also have to check the accuracy of the neural network trained in parallel, and how it compares to the accuracy of the sequentially trained NN.

Our goal with this paper is to determine whether the problem of training a neural network can be parallelized in such a way that the time of execution will be sped up, while keeping the accuracy of the predictions the same. There have been multiple approaches to parallelizing a NN. Four approaches are explained in [1], in order of level of granularity, from training session parallelism, to weight parallelism. Training session parallelism consists of no communication between the processes, providing zero overhead. Weight parallelism is the finest grained solution. With it, the input from each synapse is calculated in parallel for each node, and the net input is summed via some suitable communications scheme. Our solution will be based on the exemplar parallelism approach which provides little communication overhead and is very suitable to our experimental environment.

We will start by examining some related scientific papers regarding the problems of neural networks, handwriting recognition, and parallelization, and how they relate to this paper. Then we will give a basic overview of our solution, first by going over existing knowledge of standard NN structure, followed by a bird's eye view of the parallel architecture. The implementation will be discussed in the Experimental methods section, where we will provide an explanation of a standard sequential implementation for the NN classification problem, a parallel implementation, ending the section by giving details on the specific environment being used to run the tests and provide our results. At the end we will compare the results generated by the different implementations, and we will asses the impact and worth of our solution in the conclusion.

II. Related work

The applications of AI are vast. From self driving cars, to medical diagnostics, AI has been providing solutions to some of the world's biggest problems. The invention of neural networks has been one of the biggest breakthroughs in AI and ML. Heckt-Nielsen [2] presents a survey of the basic theory of the back-propagation NN architecture covering architectural design, performance measurement, function approximation capability, and learning. One of, if not the most, useful applications of neural networks is in pattern recognition. Pao [3] has elaborated the nature of the pattern-recognition task and adaptive pattern recognition (and its applications) as one of the most useful applications of AI.

One application in which pattern recognition would be right at home is in the recognition of handwritten text. Oh and Suen [4] introduce the class modularity concept to the feed-forward NN classifier to overcome the conventional feed-forward NN's complex problem of determining the optimal decision boundaries for all the classes involved in a high-dimensional feature space and the limitations that exist in several aspects of the training and recognition processes. Graves and Schmidhuber [5] combine two recent innovations in NN, multidimensional recurrent NN and connectionist temporal classification [6]. They introduce a globally trained offline handwriting recogniser that takes raw pixel data as input. Unlike competing systems, it does not require any alphabet specific pre-processing, and can therefore be used unchanged for any language.

AI requires huge computational power for processing. Seeing as microprocessor manufacturers have struggled to increase the raw computational power of CPUs, the relevancy of Moore's Law has slowly been fading. This in turn has increased the need for engineers to think of new ways to increase the amount of computation that is possible in a finite period of time. This is where parallel computing comes in with [7] and [8], introducing new paradigms for parallel algorithm design, performance analysis and program construction. Akl [9] surveys existing parallel algorithms, with emphasis on design methods and complexity results.

Parallelization is necessary to cope with the high computational and communication demands of NNs, but general purpose parallel machines soon reach performance limitations. Serbedzija [10] explores two approaches: parallel simulation on general purpose computers, and simulation/emulation on neurohardware. Different parallelization methods are discussed, and the most popular techniques are explained. Parallelizable optimization techniques are applied to the problem of pattern recognition and learning in feed-forward neural networks by Kramer [11].

III. Solution overview

A. Neural network structure

Our solution is designed using the standard multilayer perceptron NN, also known as a backpropagation NN shown in Fig. 1. The NN typically consists of an input layer of nodes (perceptrons), an output layer of nodes, and at least one (usually more) intermediate layer. The network is trained on a data set consisting of (x_i, y_i) pairs where x_i is a feature vector used as an input, and y_i is the true output, or the ground truth, for that feature vector.

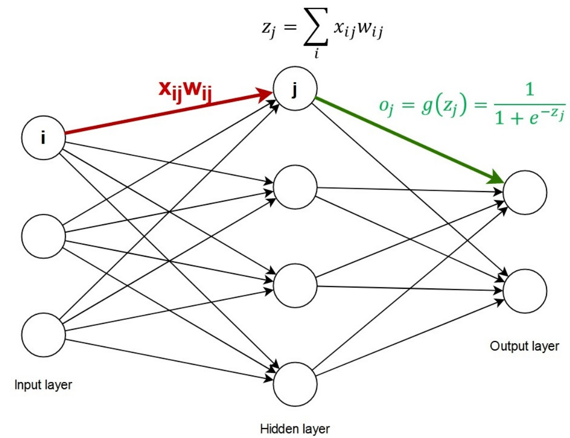

Fig. 1. Backpropagation NN structure

A node in the ith layer of the NN is connected to all nodes in the $(i+1)^{th}$ layer. The connectors that connect a node in a layer to the nodes in the next layer have assigned weights w_{ij}. We denote by x_{ij} the ith input to the node j. A node in one layer takes all of the products $x_{ij}w_{ij}$ that come as inputs from the previous layer, computes their sum z_j and uses it to compute the output (or the input to the next layer).

The NN learns by comparing the computed output with the true output, and adjusting the weights of the nodes accordingly so the computed output is the same as the desired one (if the computed output is already the true output, no adjustment is done). The changes to the weights propagate back through the network (hence the name) up until they reach the input layer. The algorithm stops after a full epoch without change in weights (sometimes we stop the algorithm prematurely to combat the problem of overfitting to the training set). Sometimes the outputs of the problem are not uniformly distributed among the population of the input/output pairs. For this reason we need a BIAS node with its own weight assigned to it, at each non-output layer, that has no inputs going into it. If the population is biased towards a certain output class, the BIAS allows us to skew the evaluation function so that we can better predict the true output.

B. Parallelization

There are multiple strategies to implement a parallel solution for the backpropagation NN. Some of the main strategies are discussed by Pethick, Liddle, Huang and Werstein [1]. We will go with the ”preferred technique” according to Rogers and Skillicorn [12], which is exemplar parallelism, also known training example parallelism. In essence, the training set is

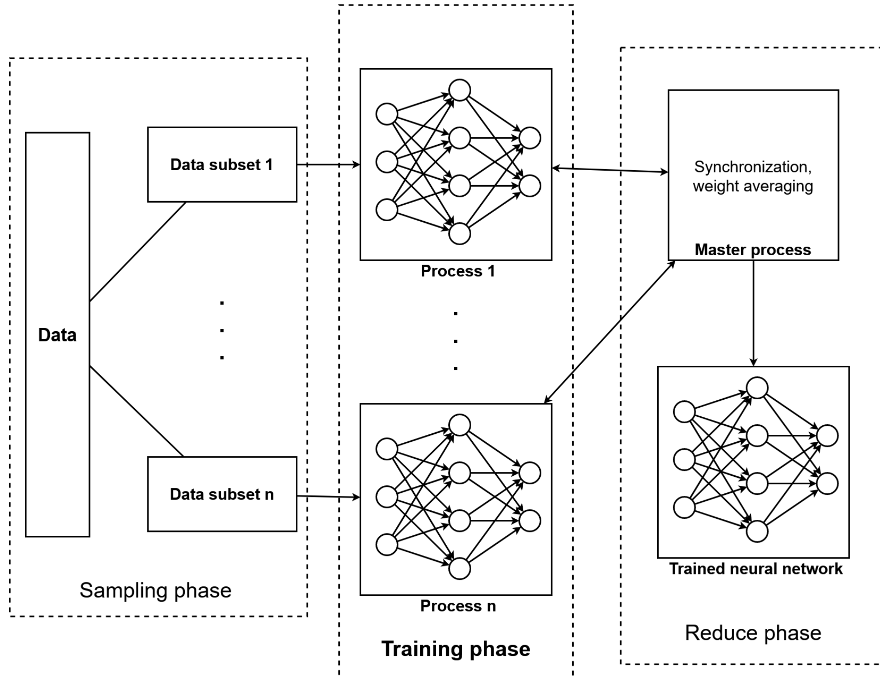

Fig. 2. The parallelization method. The dataset is split equally among the processes, each process trains its own neural network, and at the end all the processes average out their weight vectors.

split into disjoint subsets and each running process (a thread, microprocessor or separate machine) works on only one subset. The processes need to start with initial states identical to each other, meaning the weights associated with each node need to be the same. Usually this means that every weight at the beginning is set to 1. At the end of each epoch, the changes are combined and applied together to the neural network by averaging them out. The diagram for this parallel solution approach is given in Fig. 2.

Two advantages come to mind when we think about this approach to parallelizing a NN. First is the small overhead that occurs because of the communication between processes. Namely, the processes only communicate at the end of an epoch to adjust weights, so relatively few messages are generated. The second advantage is the speed-up during the training phase. Since the data set is split into n smaller subsets s_i, where $i = 1, 2, ..., n$, the time it takes to go through an epoch is the maximum of the times t_i it takes to iterate through a subset s_i (usually called a sub-epoch), or $t_{epoch} = max(t_i)$. A disadvantage to this approach is that it does not provide a performance increase at the layer level, meaning that no two nodes in the same process, working on the same subset of data, work in parallel.

IV. Experimental methods

For the purposes of this research, we will conduct two experiments that correspond to two implementations for the handwriting recognition back-propagation NN. The first one will be a classical sequential solution and the other will be a new parallel implementation. For both experiments, we will use Python with its sklearn library which includes our MLPClassifier to train the NN.

For the sake of simplicity, the dataset we will be using to train the NN is the MNIST dataset [13] of handwritten digits which includes 60,000 labeled 28x28 images of handwritten digits in its training set and 10,000 images in its testing set. The implementation can easily be extended to use the EMNIST dataset [14] which also includes images of handwritten letters.

A. Sequential implementation

The sequential implementation is executed conventionally, starting by reading the training and test data from csv files in the format $(y, x_1, ..., x_{784})$ where y is the digit that corresponds to the image represented by the pixels denoted by x_i where each digit x_i holds values from 0 to 255 representing the amount of the color black in the pixel, 0 being all white and 255 being all black.

After splitting the dataset into training and test data sets, we will again split the training set into a pair of a training and

validation subsets. The training subset will be used to train a vast variety of classifiers, each having a different number of neurons in their hidden layer, and each being run at a different maximum of epochs. The numbers of neurons in the hidden layer and the number of epochs will vary in the interval from 25 to 100. We will determine which of the classifiers makes the most accurate predictions by making predictions on the validation set and checking to see their accuracy.

Finally, having determined the most accurate classifier, it is validated by the test data subset. The goal is to find a NN model that is easy and fast to train but makes the most accurate predictions possible. Note that sometimes the most accurate classifier is not the best choice because it may take longer to train than it's worth. For this reason, the actual classifier that will be used on the parallel implementation will need to be determined as the classifier which has the biggest impact factor out of all. Our impact factor will be defined as (the accuracy)/(the training time).

B. Parallel implementation

In development of a parallel implementation, the goal is to take the training set and distribute it as evenly as possible among the processes in the pool. The master process, usually the one with rank=0 is responsible to read the data from the csv file into memory, to determine the number of training examples per process and to scatter (or broadcast) the data among the other processes. After each process has their share of the training data, they can begin the training independently of each other. First they initialize their own classifier. We have observed that the optimal classifier should be one with 50 neurons in its hidden layer and a maximum of 50 epochs. Afterwards, each process begins fitting (another word for training) the NN to its respective subset of the training data.

When the processes are finished, they need to send their weight matrices to the master process whose job is to compute an average for each respective element in the different weight matrices, and produces an all encompassing weight matrix for the neural network. The master process then uses the test data to predict the class of each training example, comparing it with the real class and thus, determining the accuracy of the neural network.

The main tool that will help us write the parallel algorithm is the Python library $mpi4py$ that offers the means necessary to write a parallel program using a Message Passing Interface.

C. Experimental environment

The solution is implemented in such a way that it can be executed on any multiple processor configuration, so long as the dataset can be split evenly among processes the algorithm is started with. For example, our training data consists of 6,000 training examples, meaning that the program should be started with a number of processes n, such that 6,000 is divisible by n. For our testing purposes, we will run the algorithm on an i7 7500U dual-core CPU, with $n = 2$ processes.

Out of the three classifiers described in the sequential implementation, we will determine the optimal classifier to

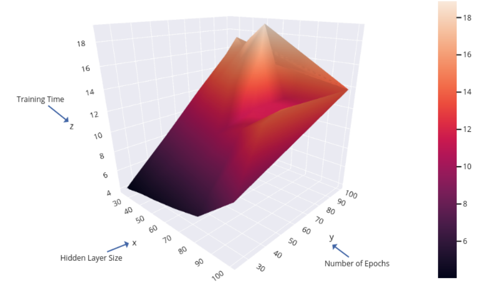

Fig. 3. A 3D mesh depicting the training time with respect to the hidden layer size and number of epochs

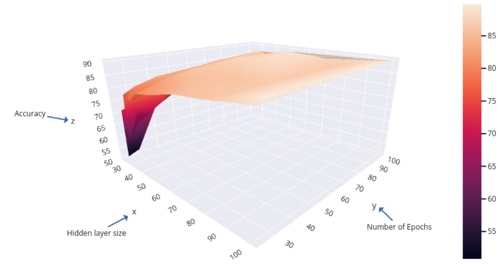

Fig. 4. A 3D mesh depicting the accuracy with respect to the hidden layer size and number of epochs

use for the parallel implementation of the NN. That classifier should provide a good balance between the accuracy of the predictions, and the time it takes to train.

To asses the value of our exercise in parallelization, we can introduce an $impact_factor$ parameter, defined as

$$impact_factor = \frac{prediction_accuracy * s_1}{training_time * s_2} \tag{1}$$

the reasoning being that the profit is proportional to the accuracy, while inversely proportional to the training time (the higher the prediction accuracy, the higher the impact, and the higher the training time, the lower the impact). The $impact_factor$ parameter defined as such, can provide us with a numerical representation of the benefit regarding the parallelization of the neural network algorithm.

The parameters s_1 and s_2 represent significance factors. For the purpose of this paper, we will assume that prediction_accuracy and training_time have equal importance by setting $s_1 = s_2 = 1$. This is not always the case in practice. Usually, the trade off between time and accuracy will vary from situation to situation and so the significance parameters would have to be set accordingly.

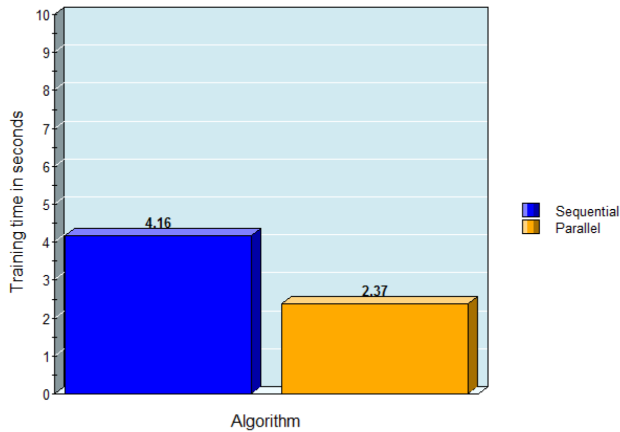

Fig. 5. Comparison of training time between the sequential and parallel algorithms

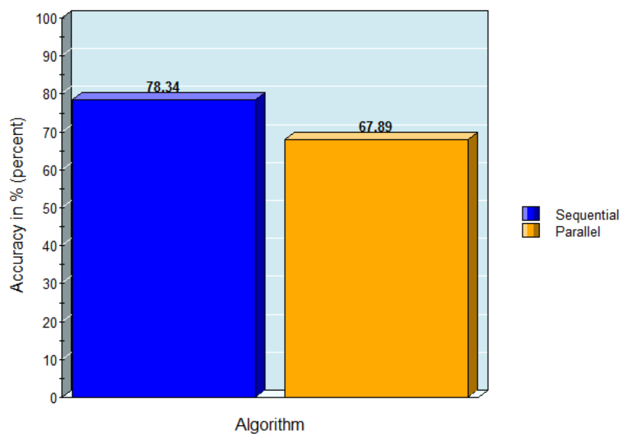

Fig. 6. Comparison of prediction accuracy between the sequential and parallel algorithms

V. Results

A. Sequential implementation

Running the script that contains the sequential algorithm, we can see how much time it takes to train all of the individual classifiers. In Fig. 3 we can see a mesh of all the classifiers and how the training time reacts almost linearly to different hidden layer sizes and different epoch numbers. When it comes to the accuracy of predicting the test set, the same type of mesh is given in Fig. 4, where the accuracy reacts very differently to epoch numbers and hidden layer sizes, from the training time. As we add more neurons to a classifier's hidden layer and we increase the number of epochs, we can surmise that the time it takes to train it isn't worth the negligible increase in accuracy we get. Our impact factor parameter that we defined earlier shows us that the best trade-off in terms of accuracy and training time is accomplished using a classifier with 30 neurons in its hidden layer, with a number of 25 epochs. Such a classifier that gives us a reasonably good accuracy of 78.34 percent using only 4.16 seconds to train, will be used in our parallel implementation, to see if and how much of an increase in speed we can gain, whilst keeping the accuracy in check.

B. Parallel implementation

With the parallel algorithm running in a Python script, we can notice a few interesting details in Fig. 5. First is the speed with which the neural network is trained. We are using a classifier identical to optimal classifier from the sequential implementation (recall that the optimal classifier had 30 neurons in the hidden layer and a maximum of 25 epochs). Our parallel program managed to train the neural network in a little over 2 seconds, which is almost half of that of the sequential algorithm. We can see here that the parallel way of training a neural network using data parallelism can bring great benefits regarding training time.

Unfortunately, the same can not be said for the accuracy of the predictions. Although not bad by any means, we see in Fig. 6 a significant drop-off from the sequential algorithm's 78 percent, to this parallel algorithm's 67 percent accuracy.

We can use our $impact_factor$ parameter, which we previously defined in the experimental environment subsection, to determine the usefulness of our experiment. For the sequential algorithm, $impact_factor = 18.83$ while for the parallel algorithm $impact_factor = 28.64$. Even though the accuracy of our predictions went down using the parallel training, the higher impact of the parallel implementation tells us that the experiment was indeed worthwhile. We still however don't get a clear enough picture of the impact of our solution, until we take into consideration the communication overhead that occurs during the parallel execution. After the master process reads the data, it needs to distribute it evenly among the other processes. Scattering the data takes around 19 seconds, and so, when we inject that into our calculation, the parallel $impact_factor = 3.17$ which is a significant drop-off from the previously calculated 28.64.

VI. Conclusion

As with everything in life, every benefit comes with a price. In our case we wanted to provide an increase in speed for the training process of the neural network intended for predicting handwritten digits. Although we achieved that speed up, it came at the price of accuracy with the predictions.

The algorithm of training neural networks is sequential in its nature, so the end results make sense taking everything into consideration. The experiment provided us with an interesting insight into the world of neural networks and parallel computing and how, sometimes, you can't have your cake and eat it too. In order to accomplish something, some sacrifices also need to be made, whether those be complexity, resources, or in our case, accuracy.

This experiment proves useful as a jumping-off point for further research. Priorities in said research would include refining the computing of the final weight matrix of the trained

NN, rather than just averaging the weight matrices of the separate NNs trained in the individual processes. We would also like to explore finer grained solutions where the processes communicate more often (during or after each epoch, rather than just at the end of the training to computer the weight matrix). Another thing that could prove enlightening is to run the algorithm on different workstations with higher numbers of processes, or on a cluster of multiple different computers. That way we could determine the impact different numbers of processors could have on execution time and whether the time grows linearly with the number of processors or not, and whether or not the overhead of communication in a cluster of computers sharing a network is significantly larger than that of a single multiprocessor machine. Lastly something that would likely provide an interesting read, is the exploration of different ML algorithms. We've seen that parallelizing NNs doesn't breed satisfactory results, but perhaps Naive Bayes classifiers, Decision trees, Linear and Quadratic discriminant analysis (LDA and QDA) etc. could be parallelized in a more satisfactory manner.

REFERENCES

[1] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[2] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[3] Y. Pao, "Adaptive pattern recognition and neural networks," 1989.

[4] I.-S. Oh and C. Y. Suen, "A class-modular feedforward neural network for handwriting recognition," *pattern recognition*, vol. 35, no. 1, pp. 229–244, 2002.

[5] A. Graves and J. Schmidhuber, "Offline handwriting recognition with multidimensional recurrent neural networks," *Advances in neural information processing systems*, vol. 21, pp. 545–552, 2008.

[6] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 369–376.

[7] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Benjamin/Cummings Redwood City, CA, 1994, vol. 110.

[8] W. P. Computing and I. Foster, "Designing and building parallel programs," 1995.

[9] S. G. Aki, "The design and analysis of parallel algorithms," 1989.

[10] N. B. Serbedzija, "Simulating artificial neural networks on parallel architectures," *Computer*, vol. 29, no. 3, pp. 56–63, 1996.

[11] A. H. Kramer and A. Sangiovanni-Vincentelli, "Efficient parallel learning algorithms for neural networks," in *Advances in neural information processing systems*, 1989, pp. 40–48.

[12] R. Rogers and D. Skillicorn, "Strategies for parallelizing supervised and unsupervised learning in artificial neural networks using the bsp cost model," *Queens University, Kingston, Ontario, Tech. Rep*, 1997.

[13] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[14] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik, "Emnist: Extending mnist to handwritten letters," in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2017, pp. 2921–2926.