Fully Distributed Deep Neural Network: F2D2N

Ernesto Leite[1(✉)], Fabrice Mourlin[2], and Pierre Paradinas[1]

[1] Conservatoire National des Arts et Métiers, 292 rue Saint Martin, 75141 Paris, France
{ernesto.leite,pierre.paradinas}@lecnam.net
[2] UPEC, 61 Av. du Général de Gaulle, 94000 Créteil, France
fabrice.mourlin@u-pec.fr

Abstract. Recent advances in Artificial Intelligence (AI) have accelerated the adoption of AI at a pace never seen before. Large Language Models (LLM) trained on tens of billions of parameters show the crucial importance of parallelizing models. Different techniques exist for distributing Deep Neural Networks but they are challenging to implement. The cost of training GPU-based architectures is also becoming prohibitive. In this document we present a distributed approach that is easier to implement where data and model are distributed in processing units hosted on a cluster of machines based on CPUs or GPUs. Communication is done by message passing. The model is distributed over the cluster and stored locally or on a datalake. We prototyped this approach using open sources libraries and we present the benefits this implementation can bring.

Keywords: F2D2N · model parallelism · data parallelism · Deep Neural Network

1 Introduction

Data has become one of the most valuable asset a company can have and it's actually producing a lot of expectation. It brings several challenges like how data should be extracted, stored, cleaned and transformed. When the amount of data is huge or grows exponentially, it brings new challenges like how to scale or how to train AI models in a reasonable amount of time and money. Data volumes are growing every day. Deep learning models are struggling to ingest the increased amount of data without reaching the hardware limits of the training machines. GPUs are intensely used for matrix computation but the increased size of the models makes the training phase no more possible with standard techniques because the size of the models cannot fit anymore in GPUs memory even the largest. Data, model, pipeline and tensor parallelism are widely used to answer these challenges. Tensor parallelism is actually one of the best approaches but it is challenging to implement and requires a lot of engineering effort.

S. Bouzefrane et al. (Eds.): MSPN 2023, LNCS 14482, pp. 222–236, 2024.
https://doi.org/10.1007/978-3-031-52426-4_15

We prototyped the F2D2N where fragments of weights, bias, activation and gradients are fully distributed and stored in remote machines. This allow the model to be distributed over the network. These machines do not require GPUs hardware even GPUs can of course speed up the training process. Feedforward and back-propagation phases are done using an event messaging mechanism. This implementation is also ready for future improvements like Reinforcement Learning, Forward-Forward approach [4] and Generative AI.

Reducing the size of the models negatively impact the precision of the models, that's why continuing exploring how to efficiently distribute the model is an urgent need in the AI field.

But actually implementing model parallelism techniques are not straightforward and require a lot of engineering effort. In this paper we give a brief review of different techniques used to overcome DNN training challenges over the years. We present shortly what data, model, pipeline and tensor parallelism are and we will introduce an innovative way to organize and compute DNN in a distributed cluster.

2 Related Work

Scaling a DNN is hard and a lot of techniques have been implemented over the years to address this challenge. We give a short presentation of those who are actually widely used in the AI field.

2.1 Data Parallelism

The goal of data parallelism [1,2,6,9] is to distribute the training data across multiple processors or machines. The model is replicated on each processor, and each replica processes a different subset of the training data. Each device processes locally different mini-batches and then synchronized its local gradients with the other devices before updating the model parameters. This is probably the easiest way to train a DNN but this brings some important limitations while scaling up the network.

For example, Data parallelism requires each device to hold a copy of the entire model parameters, which can be a memory-intensive task for large models which is also redundant and not efficient in terms of storage. While training DNN on large datasets, it often leads to an out of memory (OOM) exception when the training model exceeds the memory limits of the device. Mini-batches need to remain as small as possible but this ends up in convergence issues.

2.2 Model Parallelism

Model parallelism [2,8] tries to resolve some of the challenges data parallelism brings when the size of the model does not fit anymore on the device. One simple technique is to split the layers of the model among available GPUs. Each GPU will compute the feed-forward *(Fx)* pass of the hidden layer at a time. Then after the back-propagation pass *(Bx)*, it will update the gradients at a time too.

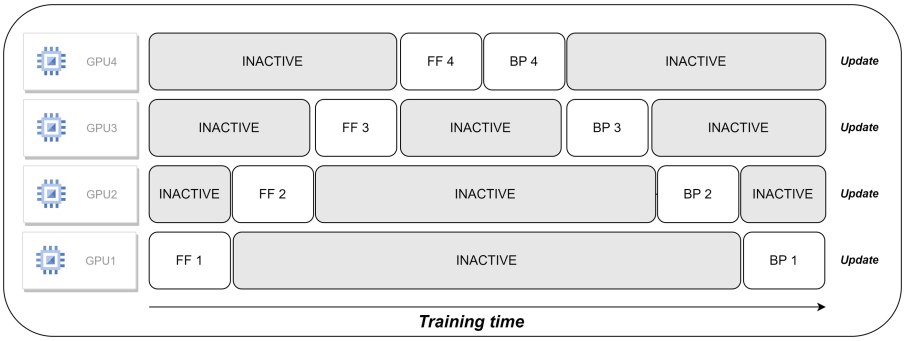

Fig. 1. Feed-Forward (FF) and Back-propagation (BP) tasks usage over 4 GPUs.

The Fig. 1 represents a model with 4 layers placed on 4 different GPUs (vertical axis). The horizontal axis represents training this model through time, demonstrating that only 1 GPU is utilized at a time.

As the graph shows, the main problem of this naive implementation is that it is not efficient and costly for large training as only one GPU is used at a time. It does not resolve the problem it tries to solve if one part of the model overfits again the physical memory of the device.

2.3 Pipeline Parallelism

To overcome model parallelism limits, Google was the first introducing the concept of pipeline parallelism [2,5,10]. GPipe is actually implemented in Pytorch. The goal is to split the mini-batch into multiple micro-batches and pipelines the execution of these micro-batches across multiple GPUs. This approach leads to a better utilization of the hardware resources because it allows the DNN to train the model in parallel. This is outlined in the Fig. 2.

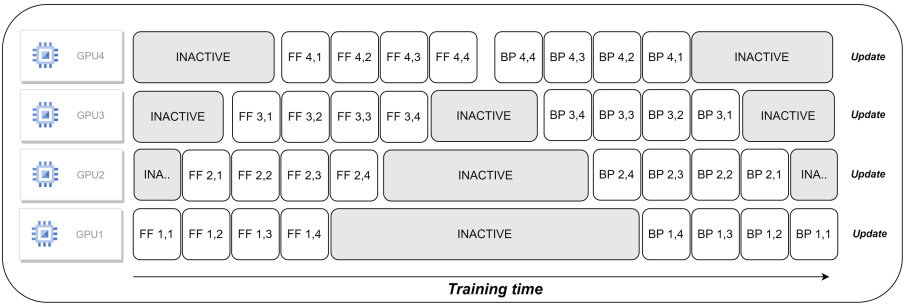

Fig. 2. Optimized pipeline parallelism over 4 GPUs.

The Fig. 2 represents a model with 4 layers placed on 4 different GPUs (vertical axis). The horizontal axis represents training this model through time demonstrating that the GPUs are utilized much more efficiently. Pipeline parallelism divides the input mini-batch into smaller micro-batches, enabling different accelerators to work on different micro-batches simultaneously. Gradients are applied synchronously at the end. However, there still exists a bubble (as demonstrated in the figure) where certain GPUs are not utilized [5].

Gpipe requires additional logic to handle the efficient pipelining of these communication and computation operations, and suffers from pipeline bubbles that reduce efficiency, or changes to the optimizer itself which impact accuracy.

2.4 Tensor Parallelism (TP)

Tensor parallelism [11,12] (and its variants) is a method to speed up the training of deep neural networks by parallelizing the computation of tensors. A tensor is a multi-dimensional array of numerical values that is used to represent data in deep learning. TP helps to scale up the training of deep neural networks to larger datasets and more complex models. TP is widely used while training huge amount of data and it's already implemented by major cloud providers like Amazon Web Services (AWS) (Fig. 3).

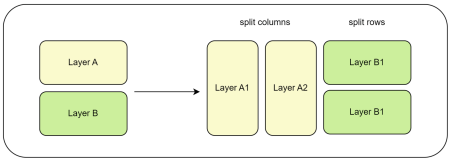

Fig. 3. Tensor parallelism applied to 2 layers A et B

TP involves dividing particular model parameters, gradients, and optimizer states among different devices.

Nethertheless, TP requires more complex software and hardware infrastructure to distribute the computation across multiple devices or GPUs. This can increase the complexity of the training process and require more expertise to set up and maintain. Not all deep learning frameworks and libraries support tensor parallelism (like transformers), so it may not be compatible with all the tools and libraries a Deep Learning project requires.

3 Proposed Architecture

3.1 Concepts

As we saw in the introduction, data is already huge in terms of volumes and it will continue to grow. An important aspect is to be able to **parallelize** the DNN as much as possible and to make it *more easy* to distribute and train.

Tensor Parallelism is actually one of the most widely solutions used to scale up and distribute the computation of a DNN, but it suffers from being complex to implement. Parallelizing data or model with these different techniques is often challenging because of the complexity of the configuration, the costly architecture and the engineering efforts it implies.

Based on that need, we intend to simplify the process of parallelization as much as possible for three reasons:

- First of all: to be able to train complex and big DNN (or Convolutional Neural Networks) with CPUs and/or GPUs machines.
- Secondly: to split the DNN in small units of computing (UC).
- Thirdly: to use open source libraries that can work in different environments like Windows, Linux or Mac OS.

A basic DNN is composed by 3 types of layers: an input layer, one or more hidden layers and an output layer which outputs the final predictions.

Each hidden layer will compute on the feed-forward pass an activation function σ (relu, leakyRelu, sigmoid, tanH, etc.) having w (weights), x (inputs) and b (bias):

$$a = \sigma(w_1 x_1 + ... + w_n x_n + b) \tag{1}$$

The weighted function is a sum of a matrix product. So we can distribute this function across a cluster of machines as the sum is commutative. The position of each UC determines the position of the matrix array the UC will compute. For the back-propagation, the derivative can be applied to the portion of weights or bias hosted by each UC as it produces the same results if it was applied to the whole set.

L is the layer, y the true label, z (weighted sum + bias) and σ' is the derivative of the activation function:

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \tag{2}$$

The error can be propagated from the output layer to the first input layer using the transpose of the upper layer $l + 1$ of w.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{3}$$

By combining (3) with (2) we can compute the error δ^l for any layer in the network.

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{4}$$

In the context of an UC, we need to compute the portion of the matrix using the chain rule.

$$\delta^{l,uc} = ((w^{l,uc+1})^T \delta^{l,uc+1}) \odot \sigma'(z^{l,uc}) \tag{5}$$

n.b.: More detailed information of the formulas can be found here [9].

3.2 Parallelized Approach

We introduce in this paper a Fully Distributed Deep Neural Network (*F2D2N*). The goal is to have units of computing (UC) that can handle a small piece of the DNN. This architecture is close to micro-services. Each UC is responsible of initializing the portion of weights or bias using the best algorithm like random, Xavier, HE, etc. It stores its data (weights, bias, activation, gradients, etc.) locally or in a data lake. The grouping of all these pieces constitute the model itself.

This approach allows the model to grow without reaching the hardware limits of any device because the architecture can be scaled in *advance* to cover the final training size of the model. This because we can calculate the number of parameters the model will hold at its final stages and calibrate the needed network to compute it.

For distributing the matrix computation (Eq. 1), we divided the hidden layer into two different layers:

Weighted Layer (WL): this layer holds the portion of weights. The size and the dimension of the matrix is calculated considering how the next layer is divided and the size of the activation matrix it receives. It is exclusively connected to its corresponding Activation layer (same index) and fully connected to the next Activation layer. WL basically distributes the computation of the splitted weighted matrix over a cluster of UCs. Each UC is dedicated to a specific portion of the matrix. In Eq. 6, s is the startIndex and e is the lastIndex of the portion of the matrix. This operation is repeated for each UC in the Layer.

$$wm_{uc} = (w_s * x_s + ... + w_e * x_e) \tag{6}$$

Activation Layer (AL): this layer holds the portion of bias and activation. The size and the dimension of the matrix is calculated considering how the next weighted layer is divided and the size of the activation matrix it receives from the previous layer. It is fully connected to the previous weighted layer and exclusively connected to the weighted layer of it same index position.

Equation 7: AL sums all the weighted matrix wm from layer $l-1$. uc_1 is the first UC and uc_n is the last UC in the layer $l-1$. Then it add a bias b and compute the activation function σ.

$$a = \sigma((wm_{uc_1} + ... + wm_{uc_n}) + b) \tag{7}$$

Each UC processes its small piece of work (weighted computation or activation computation) then triggers an event to the next UC until reaching the output layer. Figure 4 shows a simplified schema of the F2D2N based on a DNN with 1 hidden layer. It contains: one Input layer (green), 2 weighted layers (purple):

one for the input layer, one for the activation layer, 1 activation layer (orange) and one output layer (blue)

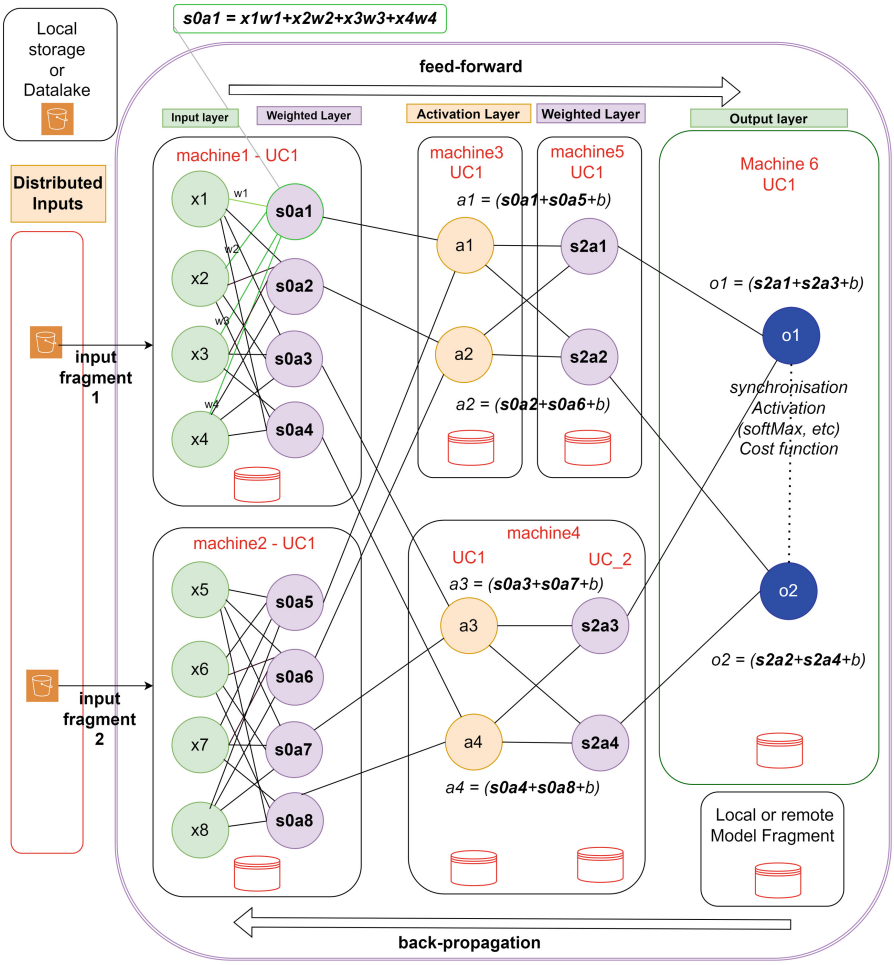

Fig. 4. Simple F2D2N with 1 hidden layer using a cluster of 6 machines (Color figure online)

As other approaches, F2D2N has some *synchronous* states. For example, an AL UC has to wait until receiving the *complete* set of the weighted matrix from the previous layer before computing the activation function in the forward pass. This is only for a training sample but does not block the UC to continue to process another sample.

Figure 5 shows activation neuron *a1* has to wait to receive weighted matrix *s0a1* from actor *machine1-UC1* and weighted matrix *soa5* from *machine2-UC1* to compute the activation function.

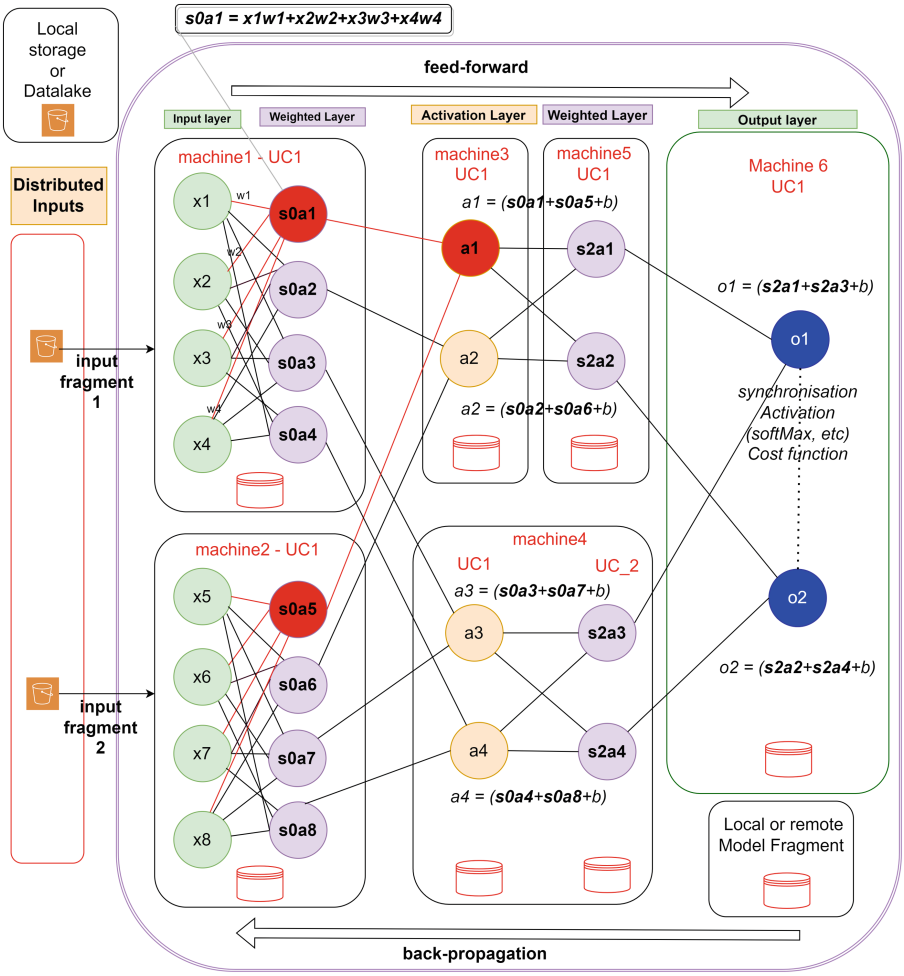

Fig. 5. Activation layer in *machine3-UC1* waiting for Weighted matrix from 2 UCs *machine1-UC1*, *machine2-UC1*

For the back-propagation pass, the AL and WL UC have to detect when it receives the last sample of a mini-batch to apply the gradients to the portions of weights or bias it holds. Each UC has an internal buffer that tracks how many back-propagation events it received and when the counter is equal to the size of the mini-batch it will apply the gradients to the local weight or bias it contains.

Algorithm 1. Feed-forward AL

Each sample has its unique correlationId
Initialize b matrix
Initialize UC hashMap variables: activation, weighted, shardsReceived
Layer l is the current layer
$ucIndex$ is the current vertical position of the UC in the L l
Step 1 : sum $weightedInput$ matrix received from layer -1 and update $weighted$ matrix
Step 2 : check if all the portions of the weightedInput matrix have been received
Step 3 : When shardsReceived are complete do :
Step 3.1 : $z[correlationId] = weighted(\text{correlationId}) + b$
Step 3.2 : $a[correlationId] = \sigma(z)$
Step 3.3 : Send $a[correlationId]$ to WL(l+1, $ucIndex$)

Algorithm 2. Back-propagation WL

Updating weights during back-propagation
Each sample has its unique correlationId
$regularisation$: hyperparameter
$learningRate$: hyperparameter
$nInput$: size of the training sample
Init local variables
$neuronCount = NeuronsInLayer$.count
$ucCount = UCInLayer$.count
$layerStep = neuronCount/ucCount$
Initialize UC hashMap variables: $weights$, $nablasw$
Step 1. Receive gradients from activation layer +1 for a specific sample.
Step 2. Compute $newDelta[layerSize]$ = weights($UCSender$) * $delta$) ▷ dot product
Step 3. Compute $nablaw[delta.length]$ = delta * activation ▷ dot product
Step 4. Case UC received all the sample of the current mini-batch
Step 4.1. Sum all the matrices of the mini-batch
Step 4.2. Compute gradients
for $i \leftarrow 0 ucCount - 1$ **do**
  for $j \leftarrow 0$ $layerStep - 1$ **do**
    $tmp \leftarrow$ weights$[i][j]$;
    $tmp1 \leftarrow (1 - \text{learningRate} \cdot (\text{regularisation/nInput})) \cdot \text{tmp})$;
    $tmp2 \leftarrow (\text{learningRate/MiniBatch}) \cdot \text{nabla}_w[i][j]$;
    weights$[i][j] \leftarrow (tmp1 - tmp2)$;
  end for
end for
Step 4.3. clean buffer

Figure 6 shows the events flow during the training process of an epoch. Some context parameters are sent during the feed-forward or back-propagation event (min, max, avg, etc.).

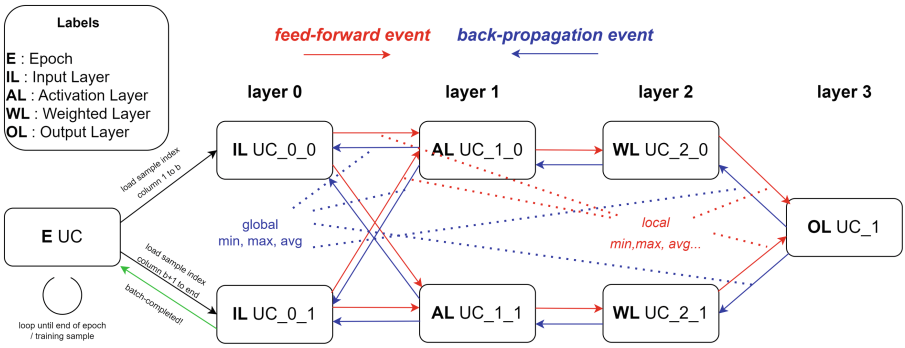

Fig. 6. Feed-forward and back-propagation event flow

4 Communication Scheme

4.1 Graph Messaging Overview

A F2D2N is a graph that computes the feed-forward pass and back-propagation pass by a message passing system. The neural network is divided *horizontally* (the layers) and *vertically* (the UCs). If we want to scale up the cluster we need to add vertically more UCs. Theoretically each UC can reach another UC in the cluster. It is easy to scale up verticaly the cluster by adding new UCs to a specific layer.

Epoch UC is the *root* of the graph. It is connected to at least one input UC which will load and compute the initial input weighted matrix. Input UC will be fully connected to at least one activation UC. This UC will sum all the weighted matrix it received from the previous layer, will add the bias and will send the activation results to its dedicated weighed UC. Weighted UC will be fully connected to all the activation UC of the next level until reaching the output UC. This architecture can also support vertical communication when UC in the same layer needs to synchronized their data like batch normalization. This will allow the architecture to support also Recurrent Neural Networks, Variational Autoencoders (VAE), Generative Adversarial Networks (GAN), etc.

4.2 Benefits

This architecture brings some interesting benefits (Fig. 7):

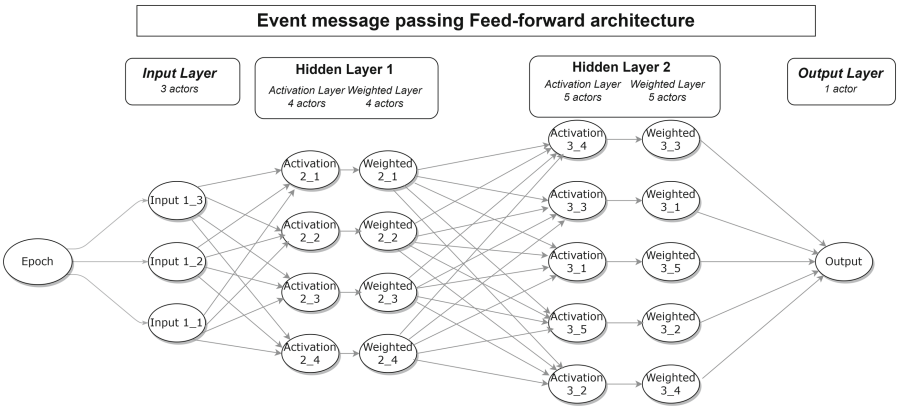

Fig. 7. F2D2N as a graph from the epoch UC to Output UC

- Model size: it is distributed on a data lake and can grow linearly with the size of cluster. It is not limited to the physical size of a machine as the load is distributed across UCs.
- Check-pointing: each UC can save its memory state on a storage like a data lake and the node manager can recreate the dead UC from it.
- Contextual variables: min, max, avg, and many more information can be transmitted in the message event. This allows the UC that receives these information to determine the global min, max, avg of the full matrix because it has the full vision of it. This removes the need for a costly parameter server [7] for example as regularization or normalization techniques can be applied during the back-propagation stage.
- The F2D2N can remain live allowing new training phases to be processed like reinforcement learning or to respond live to a request.
- The F2D2N can be reconstructed in advance based on a specific configuration.
- Modularity: each piece of the F2D2N is a local or remote UC. This allow the F2D2N to be completely divided into small units of computating and to decide where they would be placed over the network. This allows also the architect to determine which intensive compute units of the network should go to specific machines (GPUs, CPUs).
- Asynchronous: all the training samples of a batch are triggered asynchronously allowing part of the feed-forward pass to parallelize the computation of the training samples.
- Continuous Integration /Continuous deployment (CI/CD) ready: F2D2N is close to a micro-service architecture. This allow the entire architecture to be stored in a git repository, hosted in a development project under an IDE like Intellij, tested, validated and delivered using a CI/CD pipeline

5 Performance Evaluation

We prototyped this approach using Akka server implementation [14] which can scale to 400 nodes without the need to put a lot of engineering effort. Akka cluster can go beyong that size (over 1000 nodes) but this will need the help of technical specialists. We believe that for most usecases Akka implementation will handle any F2D2N implementation. But this choice allowed us to not focus on the cluster implementation itself. Note that Lightbend recently changed Akka product licensing (https://www.lightbend.com/akka/pricing). The feedforward and backpropagation pass are based on the Akka event messaging system. An actor (UC) runs in a machine and it holds its *own* memory. A machine contains one or several actors.

We prototyped a sample of a F2D2N using the MNIST dataset [3]. The solution is designed in scala and needs Simple Build Tool (SBT) to compile and run the solution. The external libraries (Maven) are defined in the build.sbt file. Additional libraries are needed as NVIDIA drivers (CUDA) if we want to activate the GPU support. This dataset has: 784 input neurons, 60 000 training samples, 10 000 test samples.

We defined 2 actors for the input layer, 2 actors for the first hidden layer, 2 actors for the 2nd input layer and 1 actor for the output layer using a cluster of 3 machines.

We ran the lab in 3 modes.

Table 1 shows the training duration of the MNIST database. Actually, the cluster mode takes longer compared to the local mode because of the messages being passed through the network and the size of the data not really significant. The cluster mode will perform better on bigger training volumes where the model would need to be splitted among the cluster.

We trained the prototype in 3 DELL R730 E5-2690 v4 @ 2.60GHz with 96GB Ram & 1Tb SSD. The instances need to run a Java runtime (11 or higher) and Scala Build Tool (SBT) in order to compile the git sources. A build.sbt file contains the references to the maven packages.

Table 1. Training duration

Mode	Duration (min)
Local: Pytorch/Jupyter	13.3
Local: F2D2N	24.2
Cluster (3 machines): F2D2N	29.2

Table 2 shows the parameters used to train the MNIST samples. We used the Categorical Cross Entropy as we are dealing with a multi-class classification problem. The prototype will handle more cost functions in the future.

Table 2. Hyperparameters

Parameters	Value
Epochs	50
Learning rate	0.195
Activation	Relu, Relu, Softmax
Cost function	CategoricalCrossEntropy
Minibatch	50
Regularisation	0.5
Input Neurons	784
Hidden Layer 1 Neurons	128
Hidden Layer 2 Neurons	64
Output Neurons	10

Table 3 shows F2D2N using 9 UCs on the local mode and the cluster mode to train the MNIST database.

Table 3. Cluster configuration

Layer	UC
Input	4 UCs
Hidden 1	2 UCs
Hidden 2	2 UCs
Output	1 UCs
Total	9 UCs (local or remote)

Table 4 shows the list of the main libraries used in the prototype. They are listed in the build.sbt and available in maven repository.

Figure 8 shows the training results for a F2D2N in a local mode. Blue columns represent the execution time of an epoch (average of 36 s per epoch) and the curve represents the cost function (categorical cross entropy) which decrease over the time. The accuracy for that training is 96.10% for a total execution time of 29 min.

Table 4. Main scala libraries

Library	Version
sbt	1.9.2
scala	2.13.12
akka	2.8.5
Amazon Coretto	20
IDE	Intellij 2023
ML	ai.dlj

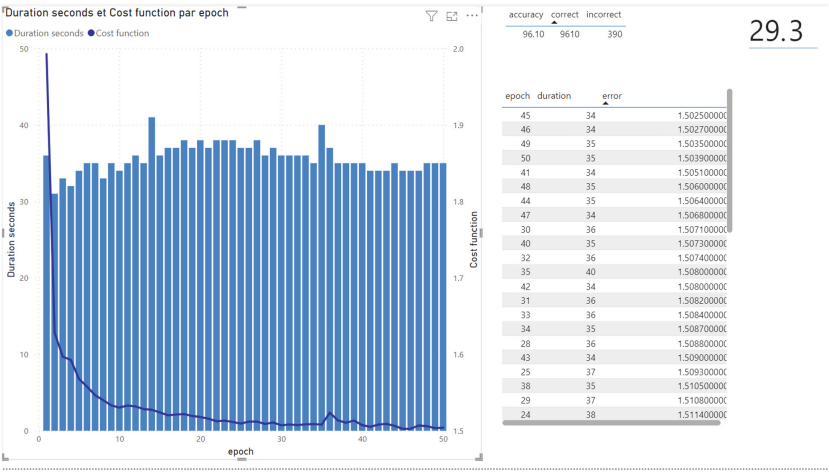

Fig. 8. Training of a F2D2N during 50 epochs (Color figure online)

6 Conclusion

In this paper, we reviewed different techniques that implement data and model parallelism. Tensor parallelism is actually widely used to tackle the challenges of model parallelism but it suffers from being very complex to implement or to extend it with new features.

We demonstrate that it is possible to parallelize a DNN into small units of computing (UCs) and to distribute the training process in an event messaging system. This modularity makes it more easy to implement new features like encoders, RNN and reinforcement learning. It also makes possible to integrate the solution in a DevOps workflow (CI/CD, tests, git, etc.).

Our tests showed that expanding horizontally the DNN can have better performance compared to vertical scaling even it can bring other challenges like exploding gradients.

As future work, we aim to train F2D2N with larger datasets and make some improvements like reducing the number of messages for better efficiency.

References

1. Ben-Nun, T., Hoefler, T.: Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis (2018)
2. Chen, C.-C., Yang, C.-L., Cheng, H.-Y.: Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform (2019)
3. Deng, L.: The MNIST database of handwritten digit images for machine learning research [best of the web]. IEEE Signal Process. Mag. **29**(6), 141–142 (2012)
4. Hinton, G.: The Forward-Forward Algorithm: Some Preliminary Investigations (2022)
5. Huang, Y., et al.: GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism (2019)
6. Jiang, Y., Fu, F., Miao, X., Nie, X., Cui, B.: OSDP: Optimal Sharded Data Parallel for Distributed Deep Learning (2023)
7. Li, M.: Scaling distributed machine learning with the parameter server. In: Proceedings of the 2014 International Conference on Big Data Science and Computing, Beijing China, p. 1. ACM (2014)
8. Li, S., et al.: PyTorch Distributed: Experiences on Accelerating Data Parallel Training (2020)
9. Nielsen, M.A.: Neural Networks and Deep Learning (2015)
10. Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., Catanzaro, B.: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism (2020)
11. Singh, S., Sating, Z., Bhatele, A.: Communication-minimizing Asynchronous Tensor Parallelism (2023). arXiv:2305.13525 [cs]
12. Wang, B., Xu, Q., Bian, Z., You, Y.: Tesseract: parallelize the tensor parallelism efficiently. In: Proceedings of the 51st International Conference on Parallel Processing, pp. 1–11 (2022)