Managing RNN State in Bounded Asynchronous Environments

António Venâncio[1]

Instituto Superior Técnico, Av. Rovisco Pais 1, 1049-001 Lisboa
antonio.venancio@tecnico.ulisboa.pt

Abstract. Recently stateful Neural Network (SNN) have achieved state-of-the-art performance in a wide range of application domains, such as language processing, time-series prediction, credit card fraud detection and many others. This class of NNs can retain their internal state across several queries (e.g. inputs), which allows for capturing temporal relationships among events. However, SNNs raise non-trivial challenges when employed to perform inference in large-scale deployments, e.g., when they are distributed across large cluster of machines. In this context, the need to update and retain their internal state across different user queries raises the problem of how to ensure that this state is efficiently and effectively synchronized among the different machines used to serve user requests. This thesis aims to investigate how robust SNNs are to the use of relaxed synchronization schemes which have the potential to enable high scalability in distributed settings. This document reviews the current start-of-the art in this area and plans how to carry out the work ahead for my master dissertation.

Keywords: Recurrent Neural Networks · Asynchronous Communication · Distributed Systems

Table of Contents

1 Introduction

Over the years there have been a great number of breakthroughs [2,7,27,32] in the realm of Machine Learning (ML). Thanks to such advances, these algorithms have been successfully employed to tackle more and more complex problems in a wide range of application domains [5,33]. First introduced in 1943, Neural Network [21] have undergone dramatic advances over the last decades [7,26].

These breakthroughs have been enabled also by the constant improvement of hardware technologies [16,23] and by the availability of evergrowing volumes of data [8].

Among the many different types of ML approaches that have been investigated in the literature in recent years, Stateful Neural Networks (SNNs) [5,7,10,11,25,32] have emerged as a powerful tool to model and predict the complex temporal relations that arise in a broad number of application domains, ranging from financial fraud detection to natural language processing [25] and financial stock predictions [19].

As their name suggests, SNNs maintain an internal state in order to answer incoming queries. The use of internal state allows the model to capture temporal relations between sequences of inputs, which is key to enhancing their predictive quality. However, it also raises the problem of how to synchronize the model's state in production environments where a cluster of machines is used to serve user (inference) requests in parallel.

In these settings, the execution of a user request on a node of the cluster leads to an update of the internal state of the model. This update has not only to be disseminated to other cluster nodes, but also synchronized with conflicting state updates produced by the execution of concurrent requests at different nodes. The many state synchronization mechanisms studied in the literature on distributed systems exhibit very different trade-offs regarding (i) the consistency guarantees (e.g., from strong transactional isolation semantics [13] to weak eventual consistency [6]) vs (ii) the efficiency/scalability they provide.

The goal of this dissertation is precisely to investigate the impact of use of different synchronization mechanisms on the prediction quality (e.g., loss) and efficiency (e.g., response time, throughput) when employed to regulate concurrent state updates of SNNs deployed in distributed production enviroments. The main hypothesis that I intend to study in my work is whether the use of relaxed synchronization techniques, e.g., bounded staleness consistency models [14,20], which are known to attain much higher scalability levels that strong synchronization methods, can be effectively used in the specific context of SNNs, i.e., without excessively compromising the model's predictive quality. To this end, I plan to focus my thesis on the use case of SNNs employed in the context of financial fraud detection applications [5]. This application domain is an ideal context to investigate the research problem targeted by my thesis, given (i) the challenging performance and scalability requirements that they need to face in real settings, and (ii) that SNN approaches are a natural fit to capture the temporal relations that existing among the transactions generated by fraudsters or legitimate users.

The remainder of this report is structure as follows: Section 2 described the background and state of the art that is going to be used for this thesis. Section 3 proposes the work to be developed to achieve the goals mentioned previously. Section 4 looks into the methods used to develop the solution and its resources and Section 5 organizes the development process onto a schedule from mid January until the end of October.

2 Background & State of the Art

This Chapter will cover the theoretical background and the concepts that will serve as the foundation for this thesis proposal. For that it is organized into two main sections:

– **Stateless & stateful ML approaches**, where the theoretical background will be presented for the ML algorithms that will be used in this proposal.

– **Distribution of ML Models**, Covering the aspects to be taken into account when deploying aforementioned ML solutions into distributed settings.

2.1 Stateless & stateful ML approaches

Introduced by McCulloch and Pitts in 1943 [21], Neural Networks have since become the foundation for the state of the art in many application domains ranging from autonomous driving [34], finance [5], language processing [11,25] and many others. This state of the art performance was enabled through more data availabbility [1], better frameworks [2,35] and optimizations discovered during the 2010's with the introduction of new architectures that greatly increase the overall capabilities of these technologies such as Gated Recurrent Units (GRUs) [7], Convolutional Neural Networks [26], LSTMs [15] and transformer networks [32].

For the sake of this thesis proposal different NN approaches will be categorized in two main categories: stateless and stateful approaches. A stateless approach translates to a ML model in which the previously processed input does not affect the outcome of the next input. Whereas a stateful model uses the previously received input in order to better process the next one.

2.1.1. Stateless Neural Networks A NN aims to loosely resemble the behavior of real-life neurons in order to process inputs and provide accurate predictions. It is a supervised ML method. This implies that development of the model is divided into two phases, the training and inference model. During training the model takes in input data of the problem it aims to learn to predict, along with the label of the input indicating the correct prediction. The error of the model prediction (loss) is measured and is used to modify the model's internal adjustable parameters, often called weights, with the objective to minimize the error for the current prediction as well as for future ones.

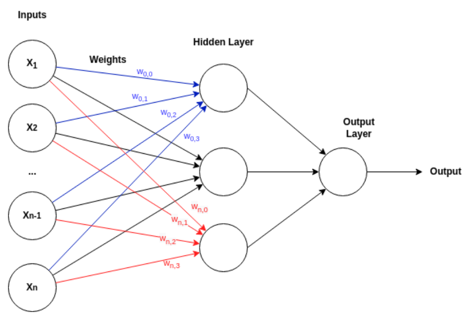

Fig. 1: Structure of a Neural Network Architecture, each edge represents a number of adjustable weight parameters that can be tuned during training. The layers and neurons in each are variable depending on the problem, a single layer with three neurons was illustrated for simplicity. The number of neurons in the output depends on the kind of problem to be solved.

A simple or *stateless* neural network contains three layers: an input layer, hidden layer and an output layer, as shown in Figure 1. Each layer consists of individual units named *neurons* [31]. The input layer receives the input given to the network containing the information necessary to solve a given problem, which, in abstract terms, can be represented by a vector of floats x of variable size n. The hidden layer receives the input and computes its representation through its weight parameters. Finally the output layer is responsible for generating the output of the network. This output varies depending on the type of problem at hand. For classification (prediction of a Boolean value) or regression problems, a single neuron translating the probability of truth or indicating the predicted value, respectively, will suffice. However, for classification tasks that require distinguishing among a larger number of classes, several neurons are typically used to encode the predicted probability of each output class [11].

A single layer of a NN is comprised of a weight matrix W of size ($number\ of\ neurons \times input\ size$), a bias vector b, and an activation function $g : X \subseteq \mathbb{R} \to \mathbb{R}$. The output of a layer is described with the following function $f : X \subseteq \mathbb{R}^{input\ size} \to \mathbb{R}^{number\ of\ neurons}$:

$$f(x) = g(z(x)) \tag{1}$$

$$z(x) = Wx + b = \tag{2}$$

$$= \begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \dots & \dots & \dots & \dots \\ w_{p,0} & \dots & \dots & w_{p,n} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_n \end{bmatrix} \tag{3}$$

Through the stacking of several layers, with each one taking input from the previous layer's output, it becomes possible to solve complex problems and make accurate predictions with properly trained weight and bias parameters. In order to learn these values, during the training phase, an error/cost function is used to determine the correctness of the model's prediction. By calculating the gradient of the cost function according to all weights and biases in the neural network, it is possible to determine the adjustments necessary in order for the neural network to minimize the error by adding the negative of the gradient values to each weight. This adjustment is often realized iteratively through batches on a set of training data. This process is named Stochastic Gradient Descent (SGD) [37].

The aforementioned computations are executed in two steps: the first one, named *feed forward*, corresponds to compute the model prediction based on the provided input; the second one, named *backpropagation*, computes the gradient values and iterivately determines the adjustments to apply to each layer of the NN starting from the output layer, and propagating backwards the adjustments up to the initial layer.

2.1.2. Stateful Neural Networks Also known as *Recurrent Neural Networks* (RNN) [10], SNN build on the original Neural Network architecture by aiming to capture relationships between sequential data. Some examples are words in sentences, stock market returns, credit card transactions and many others. These relationships can be captured by adding a state matrix, updated whenever an input is processed, thus allowing previous inputs to affect the processing of future ones. Assuming a simple feed-forward network defined by the function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$:

$$f(x) = Wh(x) + b \tag{4}$$

$$h(x) = g(Vx + c) \tag{5}$$

Where $g(x)$ is the activation function, V and W are the weight matrices of the neural network, while b and c are the respective bias vectors. A Recurrent Neural Network proposes an architecture defined as:

$$f(x_t) = Wh(x_t) + b \tag{6}$$

$$h(x_t) = g(Vx_t + Uh(x_{t-1}) + c) \tag{7}$$

The added matrix state $h(x)$ stores patterns and relationships between the past inputs as seen in Figure 2, generating an internal state in the NN that is kept throughout the sequence. Without this state, the model would only be able to make predictions based on the incoming input.

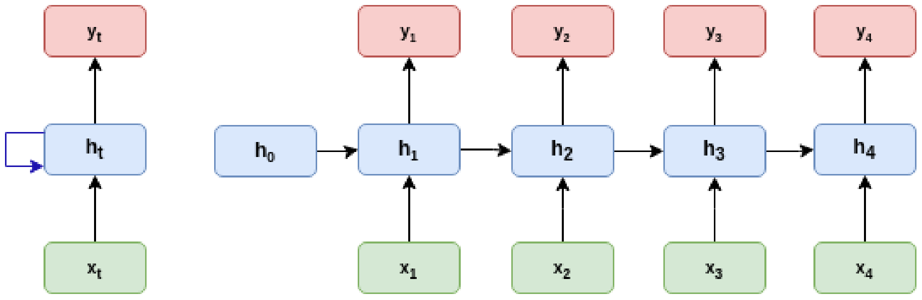

Fig. 2: Architecture of a recurrent neural network, unrolled across several inputs

Gated Recurrent Unit (GRU) Y. Bengio *et. al* [4] have identified a key issue with RNNs, namely the *The vanishing gradient problem*. As seen in the previous Section, the output of a Recurrent layer is dependent on the output of earlier inputs in the sequence. As such, when performing backpropation on RNNs the gradients have to be propagated throughout the different time-steps. Since backpropagation corresponds to a multiplicative process, depending on how large or small the gradients computed are, the gradients related early to inputs in the sequence can vanish rapidly to zero or explode to infinity, making the model more difficult to be trained. Cho *et. al* tackled this problem with the development of Gated Recurrent Unit (GRUs) [7], a recurrent unit where the state matrix described in equation 7 becomes instead:

$$h_t = \mathbf{u_t} \odot h'_t + (1 - \mathbf{u_t}) \tag{8}$$

$$\textbf{Candidate update: } \mathbf{h'_t} = g(Vx_t + U(\mathbf{r_t} \odot h_{t-1}) + b) \tag{9}$$

$$\textbf{Reset gate: } \mathbf{r_t} = \sigma(V_r x_t + U_r h_{t-1} + b_r) \tag{10}$$

$$\textbf{Update gate: } \mathbf{u_t} = \sigma(V_r x_t + U_u h_{t-1} + b_u \tag{11}$$

Where V, V_r, U_r, U_u are learnable weight parameters and b_r and b_u learnable bias parameters. The architecture of the layer can be visualized in the Figure 3. Intuitively, these gates are the solution to support the capture of long term relationships between inputs. In fact, the reset gate determines how to combine the new input with the previous memory, and the update gate defines how much of the previous memory to keep for the future.

2.2 Distribution of ML Models

Recent years have witnessed a proliferation of ML technologies used in increasingly complex applications. The increasing demand in the performance of these models in large scale environments have proven impossible to be executed by a single machines, because of hardware restrictions. With this, companies and system architects have turned to using the combined resources of clusters of independent machines in order to enable parallelization and increase the total

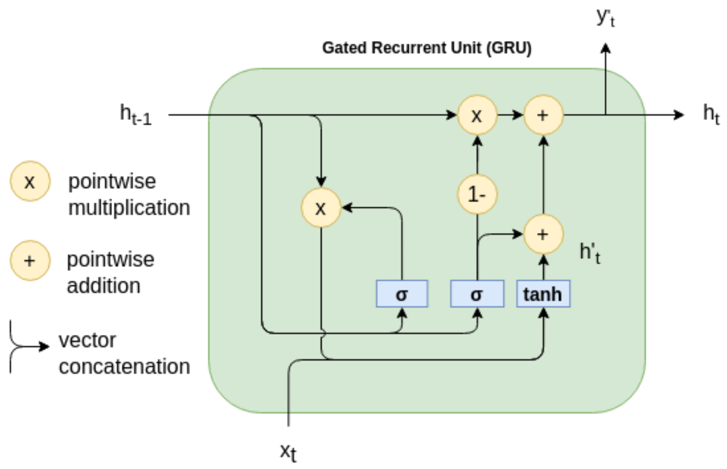

Fig. 3: Architecture of a GRU layer

amount of computational, storage and networking resources available, both in the training and inference stages. I will be referring to this type of systems as Distributed NN systems (DNNS).

This Chapter can be categorized in two main components: distribution of the training phase in Section 2.2.1 and distributing the model for inference of incoming data in Section 2.2.2.

2.2.1. Training This Section will use the taxonomy defined by M. Langer, Z. He *et. al* [18] in order categorize the architecture of Distributed deep learning systems (DDLS), its characteristics and consequences precisely. Focusing on the key design choices that need to be taken into account when distributing the training process of a ML model, the following aspects will be discussed next:

1. Data Parallelism versus Model Parallelism.
2. Centralized versus decentralized updating of model parameters.
3. Synchronous and asynchronous scheduling.

Data Parallelism versus Model Parallelism: The first decision regarding the scaling of deep learning models relates to how the workload is distributed with the addition of new machines to the system. This scaling out can be performed via two different approaches:

Model Parallelism: The model is split into partitions, which are then processed in parallel in separate machines. To perform inference or train the model, signals have to be transmitted between depending partitions sequentially to compute the final output of the model. To train a deep learning model with SGD, it is necessary to store the intermediate layers outputs temporally during inference. Often the combination of the outputs with the model parameters can prove too large to fit in the memory of a single machine. Thus, it is often a necessity to apply model parallel approaches to systems whose machines do not possess the required memory.

The model can be partitioned by either applying splits between neural network layers, where each machine hosts a given model's layer, or by splitting the layers themselves, named *vertical partioning* and *horizontal partitioning*, respectively. Vertical partitioning can be applied to any model, since the layers themselves are unaffected by the partitioning strategy. Current state-of-the-art tools like Tensorflow [2] help create DNNS systems by employing algorithms to determine efficient vertical partitioning structures with its TensorFlow serving component [24]. As for horizontal partitioning, it is usually seen as a last resort for when even the layers by themselves are to large to be fit in a single machine. In the context of this thesis, it is the plan to focus on models that fit in a single machine in order to simplify the model-building and evaluation process.

Data parallelism: The basic idea underpinning data parallelism is to increase the overall sample throughput rate by replicating the model onto multiple machines, where forward and backpropagation can be performed in parallel, to accelerate computation.

Conceptually, on stateless models (as well as stateful models where the state is reset before each batch or input sequence), scaling out of these systems can be done by adding more machines and distributing the incoming data accordingly, a stark contrast to model parallelism. The key challenge in implementing data parallelism is coordinating updates to the model's weights computed by each computing node. This often requires the use of techniques to aggregate the respective gradients and share them across all machines. Most recent developments in the domain of DNNS have focused primarily on data parellism [37,17,27].

Centralized versus Decentralized updating of model parameters: The optimization of the model parameters during the training phase of a DNNS can be realized in the two following ways: 1) *Centralized optimization:* The updating of the parameters is done in a central machine while the gradient computations are distributed to several 'worker' nodes. 2) *Decentralized optimization:* both parameter updates and gradient computation are distributed to every node while simultaneously synchronization is performed cooperatively. The flow of data and communication can be seen in Figures 4 and 5.

Centralized updating: Updating of the parameters is done in a central, single instance (often called the *parameter server* [27]. This process is done while several 'worker' machines are tasked to compute the incoming input and, if in the training phase, calculate the respective gradients to be sent to the parameter server. Figure 4 illustrates incoming and outgoing flow of data of the parameter server.

The core necessity for the success of the parameters server is to communicate parameter changes to the worker nodes with the goal of avoiding outdated parameters and as such, produce relevant gradients. As such, the degree of communication that is demanded at the same network endpoint, i.e. the parameter server can quickly become a bottleneck. For that reason, in many DNNS approaches, the role of parameter server is distributed across different machines.

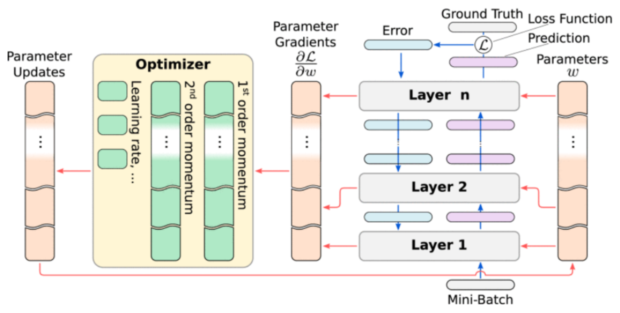

Fig. 4: Data flow in a centralized system, arrows in blue indicate the gradient computation cycle, arrows in red indicate the messages exchanged between machines to update model parameters [18].

Decentralized updating: The decentralized system removes the idea of a central machine responsible for updating the parameter values and assigns that task to all workers, where updates are shared in between them and applied immediately. Conceptually, each worker runs a replica of the parameter server in combination with performing the gradient computation. [22]

Synchronization To apply SGD during the training process of a NN model, the DNNS is required to frequently synchronize parameter values and exchange intermediate representations of the model [30]. Simultaneously, GPU-based ML greatly increases computation speeds, leading often to network communication between machines becoming the primary bottleneck in distributed setups [35]. DNNSs can be distinguished by the level of synchronization present when managing the parameter values in each worker. They can be distinguished between synchronous, asynchronous and bounded asynchronous systems.

Synchronous systems allow all workers computations to occur simultaneously, where a global synchronization barrier prevents workers from progressing until all the remaining ones in the system reach the same state, i.e. the same parameter values. The imposition of these barriers can result in the under-utilization of the available hardware and potentially increase overhead.

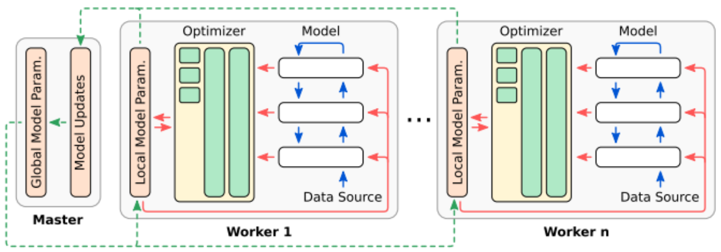

Fig. 5: Data flow in a decentralized system, arrows in blue indicate the gradient computation cycle, arrows in red indicate the flow of information for parameter updates and the arrows in green are messages communicated in the network. The master node forms the next global model state by combining local model replicas [18].

Asynchronous systems remove the aforementioned barriers and allow workers to compute incoming inputs at different states of the model. It allows to take advantage of the full capabilities of all available hardware. Ensuring no downtime between executions and deal with deviations of training between workers as a side effect.

Bounded asynchronous systems represent a hybrid approach between the two ends of the spectrum [14,20,18]. They maintain the relaxed component of asynchronous systems by allowing workers to operate at different states, but define synchronization limits for workers operating at different paces. Meaning they are allowed to operate asynchronously within certain *bounds*.

There are different motivations and effects of establishing synchronous or asynchronous implementations in centralized and decentralized systems. In this document, the discussion will be focused on bounded asynchronous approaches applied to centralized systems.

Bounded Asynchronous Systems The drawback of full synchronization is the dependency on the speed of the slowest worker in the system, which will hold back faster ones. This is the main issue that the fully asynchronous approach aims to avoid. What makes such asynchrony work in DNNSs is highlighted by SGDs robustness against lack of frequent updates and still be able to converge to a minimum of the loss function [30], allowing for models to keep their accuracy and not sacrifice computation time for communication overhead. Nonetheless, the exchange of severely stale updates can yield to suboptimal adjustments to the model's weights.

As a result, modern deep learning systems tackle the issue of staleness by ensuring that a maximum bound exists on the staleness of the state (i.e., model parameters) replicated across workers. There are many proposals for implementing such a system [14,17], which are based on two key approaches.

Value bounds limits parameter updates that have not been yet shared with other workers. To manage this, a copy of all versions of the model that are used in use throughout the cluster. Let $w_fastest$ be the model version of the fastest worker and $w_slowest$ the model version of the slowest worker and $||t_{fastest} - t_{slowest}||$ be the number of changes to the parameters currently in transit that is currently not known by the slowest worker. If a worker triggers an update that would increment the model version and violates the following formula:

$$||t_{fastest} - t_{slowest}|| \geq \delta_{max} \tag{12}$$

The worker is delayed until the condition holds again. Choosing a reliable metric and limit for a value bound can be difficult, as it is dependent on the context of the problem at hand. [9].

Delay bounds (e. g. Stale Synchronous Paralell (SSP)) [14]: Each worker (i) maintains a clock t_i. t_i is increased every time the respective worker submits gradients to the parameter server. If the slowest worker's current clock lags behind by a given threshold of s timesteps, then the update is delayed until the slowest worker has caught up.

Both approaches are quite useful to mitigate occasional delays in small and medium-sized clusters. Nevertheless, in systems with severely delayed workers (stragglers), this approach may still reduce the throughput of the entire cluster.

2.2.2. Inference Deploying the inference phase occurs after the model is fully trained and is ready to receive queries from users. A phase whose the difficulty of its distribution is dependant on the nature of the NN.

If the NN is stateless, since there is no state to be kept, there is no need for communication between workers. For that reason, distribution is fairly easy as it is only necessary to add more machines to the DNNS and distribute the incoming queries accordingly. To reduce even further the need for communication, model compression methods are also used [29,36]. The goal is to allow large models to be kept onto a single machine and avoid the need for model parallelism.

In the realm of developing DNNSs for SNN's, these become more complex with the presence of a state to be maintained between workers. Usually this issue is bypassed by grouping sequential data and isolating it onto a single machine. This allows for the state to stay isolated in a single worker and after fully computing the sequence, the state is reset. If this solution is used then scaling becomes just as trivial as with stateless networks.

This is not the case though if the internal state of the SNN needs to be retained across subsequent queries. This is the case if all incoming queries to the DNNS are part of the same sequence or workers are forced to shared the same input sequence. An example of this class of system was proposed by Bernardo Branco *et. al* [5] in 2020. This work presents a new approach to dealing with sequential time series data to detect credit card fraud by distinguishing incoming credit card transactions based of a given feature, in particular the credit card performing the transaction, as seen in Figure 6.

Conceptually, by creating a recurrent state for each credit card and only process its respective transactions using the respective state, it becomes possible to give the model the ability to track the spending habits of each credit card holder to better detect fraudulent transactions, which do not fit the holder's patterns. This state is stored in a database and loaded into the model when an incoming transaction corresponds to the respective credit card (Figure 7). After processing the state it is updated in the database for future use.

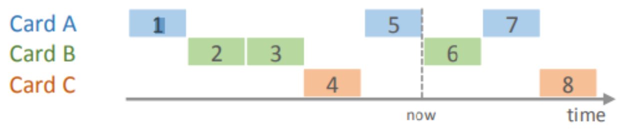

Fig. 6: Interleaved transaction history of three cards [5].

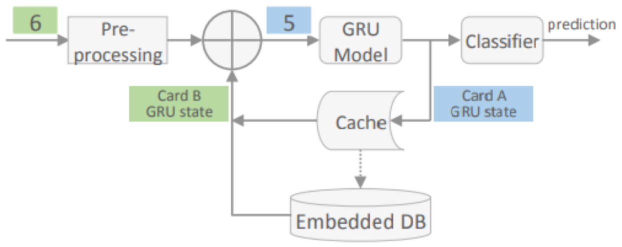

Fig. 7: At run-time, the system joins transaction 6 with the previous Card B GRU state (last changed in transaction 3). Previously, it processed transaction 5 and stored the Card A GRU state in an embedded DB and memory cache [5].

This system allows capturing the spending patterns of credit card holders, where it is possible for the proposal to be expanded to other application domains where a model can benefit from correlating inputs that share common values for one or more features. It is an example of a model where the state evolves and needs to be persisted in the system over time.

To scale out this type of stateful networks in inference one needs to tackle the problem of how to replicate and synchronize the model's internal state, which is updated whenever the model is consulted, across a possibly large cluster of machines. Bernardo Branco *et. al* [5] tackle this issue by distributing transactions (i.e., queries) to each worker based on the identifier of the credit card given as input. Since the model's state is partitioned by credit card (i.e., each credit card is associated with its distinct model's state), this approach ensures that the persistent state remains isolated in one single machine, thus removing the need for workers to communicate during the inference phase.

This technique, however, can only be applicable in case it is possible to perfectly match the partitioning of the model's internal state with the distribution of queries (i.e., inputs) across machines. This is not the case, however, if the model's internal state is partitioned based on input feature values (e.g., transactions issued towards a common merchant or associated with a range of values) that are shared among a large number of concurrent user requests. In such a case, in order to avoid inter-node communication, it would be necessary to route towards the same node the entire flow of requests that is affected/affects by some shared model's state. Considering the example of fraud detection, if the internal model's state was partitioned by merchant, instead than by credit card, the nodes associated with very popular merchants might have to cope with unacceptably large volumes of requests and ultimately hinder the scalability of the entire system.

To the best of my knowledge, the problem of how to scale out SNNs that share internal states across different user requests has not received much attention in the literature. In fact, the only work that we are aware is the above-mentioned system by Bernardo Branco *et. al* [5], which, as discussed, limits requests that share any state to be executed on the same machine.

Model Compression Methods As mentioned previously, models have increased in size and complexity over the years. The demand for such computational resources and memory is often unfeasible in hardware-restricted environments like mobile phones, smart cameras, etc. It is desirable to reduce the model size as much as possible to improve computation time, memory usage and keep accuracy simultaneously [29].

Existing literature has experimented with the compression of SNN's with proven success in increasing the model's performance while reducing its memory usage and computation requirements. This is typicaly achieved by reducing the sparsity of the weight matrices, i.e., reducing the number of entries with null values (zeros) in the weight matrix [36],

Model compression can be extremely useful in distributed settings, since a reduced model size can reduce the state size stored in the model, making it easier to communicate changes to nodes in the cluster with a smaller-sized payload.

3 Work Proposal

As discussed in the previous section, the problem of to scale out the inference phase of SNNs has received limited attention in the literature and existing approaches suffer from a key limitation: they limit queries that share any internal state of the model to be executed on the same machine. This can lead to severe load unbalances and cripple scalabilty in scenarios where most (or, to the limit, all) requests share some internal state.

The key problem to tackle is how to synchronize the internal state such that it is possible for two different machines to access the same state. With the goal being to determine whether the use of synchronization techniques that ensure bounded staleness among replicas, similar to the ones used in the training phase of DNNS, can be just as well be effectively and efficiently employed in this scenario.

3.1 Considered Model Architecture

The SNN that I plan to consider, at least initially, to pursue the goals of this dissertation, is an architecture based on the one proposed by Bernardo *et al.* [5], but extended to enable the sharing of additional state between transactions, such as merchant, zip code, transaction amount. This architecture is illustrated in Figure 8. The key difference with respect to the original network is the use of 2 GRU layers in parallel, which are meant to store, respectively: i) the state associated with all transactions so far issued by this credit card, which is meant to capture the spending habit of each holder; ii) the state associated with all transactions so far issued that share a common value for a (set of) features.

Before being processed through the GRU layers the transaction is first transformed into integer indexes and normalized, i.e. data values are aligned to a common scale. Where it is then processed through a set of embeddings to compile the representation of the input into a feature vector. And after being processed through the GRU layers, it is concatenated with the original feature vector so the output is dependant on the initial input directly and on the state.

This architecture is generic, in the sense that it allows for experimenting with different definitions of shared state across requests. This is desirable as it allows us to consider the use of different definitions of share state, which will entail different requirements in terms of inter-node synchronization frequency, e.g., corresponding to scenarios in which it is more or less likely that concurrent requests share state.

Although LSTMs are also a type SNN's, in comparison to GRU they present a greater level of complexity with superior parameter count, this characteristic results in LSTMs needing more data to be fully trained in contrast to GRUs, added to a smaller size of the internal state, facilitating state updates in between nodes, hence the decision to use GRU layers. By placing both GRU layers in parallel it allows the processes of fetching and loading both states to be performed in parallel and thus, minimize downtime.

3.2 Distribution of the Model

The main goal of this work is to analyze the impact of bounded staleness on persistent recurrent states in a DNNS. The problem will be tested using DNNS as seen in Figure 9. Workers perform inferences on incoming queries that update their internal state. This updates are sent immediately to a Model State Storage (MSS) system which stores and aggregates all the incoming GRU states. The states are then broadcast to the different workers.

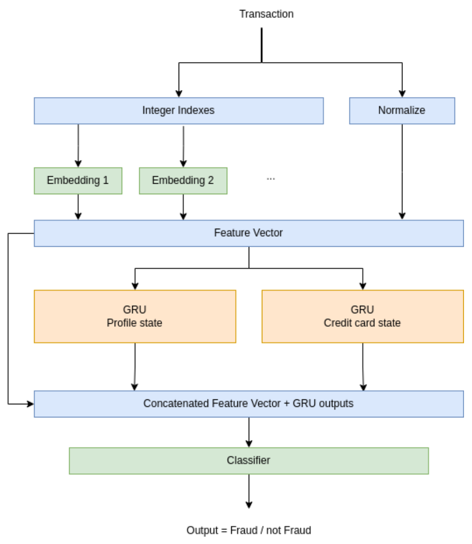

Fig. 8: Model Architecture proposed for this thesis.

By using bounded asynchronous techniques, the workers will be able to keep computing incoming queries without a strict synchronous delay, the parameter will impose bounded staleness by using either Delay bounds or Value bounds to set the threshold of staleness in the workers.

The DNNS will be tested using both centralized and decentralized approaches to the MSS. Different synchrony techniques will be used, in particular synchronous, asynchronous and bounded asynchronous. All these characteristics will be combined and evaluated to understand the impact of staleness as best as possible.

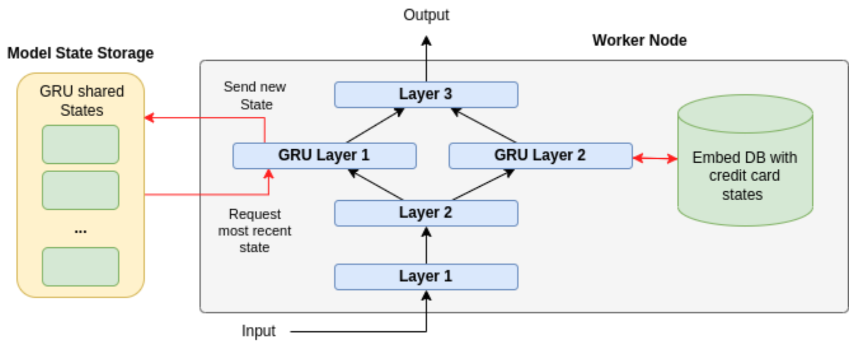

Fig. 9: The proposed distribution of the model. Arrows in red indicate the switching of GRU states depending on the incoming input.

4 Work Methodology

The work to be developed in this thesis can be distinguished in two different stages to be executed in order:

Model Development stage: The proposed model will be constructed and tested according to the standards and evaluation metrics defined in Chapter 5.2. This will allow also for evaluating the impact of using additional shared internal states on model's accuracy and training/inference performance with respect to the baseline model by Bernardo *et al.* [5]. Note that the goal of this work is not to build new SNN architecture that outperform prior models via the use of shared state. As already mentioned, my focus is instead on studying the impact of synchronization on the quality and performance of the model during inference phase.

Model Distribution stage: With the pre-trained trained, it will be copied and placed in the developed DNNS and the robustness of GRU states will be tested using the different bounded asynchronous methods in the inference stage.

4.1 Technologies Used

There are several frameworks available for the development of such models such as Tensorflow [2], PyTorch [28] and many others, as well as frameworks to distribute such models, for example Spark ML [35] and Tensorflow Serving [24]. For development of deep learning models, Tensorflow [2] will be main choice for its performance and widespread usage in production environments. As for the distribution aspect, one possibility is to use Apache Spark [35] for its high performance handling large amounts of data. Specially necessary to broadcast GRU states across machines.

4.2 Dataset

To train Deep Learning models, it is important to use good datasets that translate correctly the problem to be solved at hand. In my thesis, I plan to use the following datasets:

1. The IEEE card fraud detection dataset [3], where the goal is to accurately predict the value of isfraud which holds 1 for fraud and 0 otherwise. The data is broken into two files, identity and transaction, which are joined by TransactionID. Not all transactions have information corresponding to identity. Transaction files have features regarding the contents of the transaction like ProductCD - product code, TransactionDT - referencing the datetime, TransactionAMT - amount in USD just to name a few. On the other hand, identity files contain features about the transaction process like DeviceType and DeviceInfo.

2. The ULB (Université Libre de Bruxelles) dataset on big data mining and fraud detection [1]. The dataset is composed of 284,807 transactions made by Europeans credit cards during two days of Setember 2013. From these 284,807 transactions only 492 were fraudulent reflecting on a highly unbalanced dataset with a positive ratio of 0.172%. All the 31 variables are numerical, from these 28 are the result of a PCA transformation. Only “Time” and “Amount” variables were not transformed. “Time” contains the seconds elapsed between each transaction and the first transaction in the dataset. “Amount” is the transaction amount. The feature “Class” is binary and responsible for the classification of the transaction, 1 in case of fraud and 0 otherwise.

4.3 Evaluation Metrics

An important aspect to consider when evaluating ML models is the use of the correct metric to quantify their predictive quality. Given the models to be built are for the purpose of credit card fraud detection, the data to be used to train these models is severely imbalanced. This means that the ratio between the data of transactions that are labeled as fraudulent and non-fraudulent is extremely high. For this reason, the use of metrics that can assess the model's ability to correctly predict fraudulent transactions are more important than metrics that only analyze the percentage of incoming transactions that were labeled correctly.

This model solve classification problems. This means the output can be seen as either positive or negative, which in the case of fraud detection corresponds to a fraudulent or non-fraudulent transaction, respectively. All possible predictions can then be classified as True positive (TP), True negative (TN), False positive (FP) and False negative (FN). To evaluate the model's performance following metrics will be used [12]:

- $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$ Corresponding the fraction of predictions the model performed correctly

- $Recall = \frac{TP}{TP+FN}$ The recall measures the model's ability to detect Positive samples. The higher the recall, the more positive samples detected.

- $Precision = \frac{TP}{TP+FP}$ Summarizes the fraction of examples assigned the positive class that belong to the positive class.

- $F\_score = 2\frac{Precision.Recall}{Precision+Recall}$ Combines precision and recall into a single score, allowing to measure the model's performance in detecting correctly positive values in imbalanced data.

Through these metrics, it is possible to assess how much the staleness of SNN's affects the performance of these models in comparison to when executed on a single machine. Not only that, it is necessary to take into account the ratio between the time spent computing incoming inputs and time spent communicating, or the *overhead* generated by distributing the system.

5 Scheduling

This section discusses the plan of action to be followed for the upcoming months for the development of the master thesis. The planned schedule is reported in Figure 10. The first approximately three months starting from the time of submission of this document until at most the end of May will be devoted to training and evaluating a set of SNNs using alternative states. Evaluation will be done offline, considering transaction injected sequentially during the inference phase. With the model ready for distribution, the months from June to September are dedicated to building the DNNS and evaluating it according to the defined metrics. with the final month of October left open for any unexpected issues during development. The writing of the thesis will be a constant process that will take place during the entire development process.

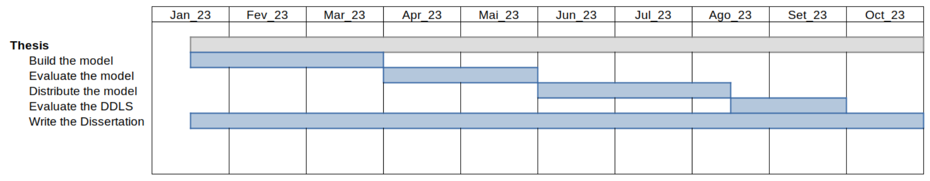

Fig. 10: Gantt chart for thesis plan

6 Conclusion

In this report it is exposed the different components involved in the development of a Distributed System tailored for Deep Learning Models. With the goal to explore the hypothesis that bounded asynchronous protocols during the inference stage of Stateful Neural Networks will grant a reduction of communication overhead while not sacrificing the model's performance in processing queries. Such protocols will not only improve performance but also allow for more scalable systems to be developed.

References

1. Ulb (universit e libre de bruxelles) dataset on big data mining and fraud detection, 2013.
2. Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
3. IEEE CIS inversion John Lei Lynn@Vesta Marcus2010 Prof. Hussein Abbass Addison Howard, Bernadette Bouchon-Meunier. Ieee-cis fraud detection, 2019.
4. Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.
5. Bernardo Branco, Pedro Abreu, Ana Sofia Gomes, Mariana SC Almeida, João Tiago Ascensão, and Pedro Bizarro. Interleaved sequence rnns for fraud detection. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3101–3109, 2020.
6. Sebastian Burckhardt et al. Principles of eventual consistency. *Foundations and Trends® in Programming Languages*, 1(1-2):1–150, 2014.
7. Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.
8. Inc. HSN Consultants. The nilson report., 2019.
9. Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric Xing. High-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.
10. Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
11. Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.
12. Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and f-score, with implication for evaluation. In *European conference on information retrieval*, pages 345–359. Springer, 2005.
13. Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.
14. Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. *Advances in neural information processing systems*, 26, 2013.
15. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
16. Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.
17. Hanjoo Kim, Jaehong Park, Jaehee Jang, and Sungroh Yoon. Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility. *arXiv preprint arXiv:1602.08191*, 3, 2016.
18. Matthias Langer, Zhen He, Wenny Rahayu, and Yanbo Xue. Distributed training of deep learning models: A taxonomic perspective. *IEEE Transactions on Parallel and Distributed Systems*, 31(12):2802–2818, 2020.

19. Jae Won Lee. Stock price prediction using reinforcement learning. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570)*, volume 1, pages 690–695. IEEE, 2001.
20. Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 241–252. IEEE, 2015.
21. Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.
22. Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.
23. NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.
24. Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.
25. openAI. Chatgpt: Optimizing language models for dialogue, 2022.
26. Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
27. LiM AndersenDG ParkJW et al. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation OSDI*, volume 14, pages 583–598, 2014.
28. Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
29. Qing Qin, Jie Ren, JiaLong Yu, Hai Wang, Ling Gao, Jie Zheng, Yansong Feng, Jianbin Fang, and Zheng Wang. To compress, or not to compress: Characterizing deep learning model compression for embedded inference. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 729–736, 2018.
30. Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.
31. Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
32. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
33. Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.
34. Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. Automated runtime-aware scheduling for multi-tenant dnn inference on gpu. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.
35. Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

36. Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.
37. Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. *Advances in neural information processing systems*, 23, 2010.