Managing RNN State in Bounded Asynchronous Environments

António Venâncio

Thesis to obtain the Master of Science Degree in

Computer Science & Engineering

Supervisor: Prof./Dr. Paolo Romano

October 2023

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to express my gratitude towards the most important people in my life. To my mother, father, brother and sister, who have supported me since the very beginning, who provided me with everything and so much more, all so I could thrive. To Gonçalo Barbas, Francisco Rodrigues, Miguel Silva and Ana Aurora, your friendship, patience and wisdom in all aspects of life has helped me develop my strengths and acknowledge my weaknesses, reaching heights never considered possible, allowing me to be who I am today.

Your belief in my abilities, even in times when I doubted myself the most, is greatest blessing one could have the privilege of having. Thank you everyone, from the bottom of my heart <3

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.

Keywords: Machine Learning, Distributed Systems, Recurrent Neural Networks

Resumo

Recentemente, as redes neuronais com estado (SNN) alcançaram um desempenho de ponta numa vasta gama de domínios de aplicação, como Linguagem Natural, previsão temporal, a deteção de fraude bancária com cartões de crédito, entre outros. Esta classe de NNs pode reter o seu estado interno em várias consultas (por exemplo, transações), o que permite captar relações temporais entre eventos. No entanto, as SNNs levantam desafios não triviais quando utilizadas para efetuar inferências em implementações a grande escala, por exemplo, quando são distribuídas por grandes grupos de máquinas. Neste contexto, a necessidade de atualizar e reter o seu estado interno ao longo de diferentes predidos de utilizadores levanta o problema de como garantir que este estado é sincronizado de forma eficiente e eficaz entre as diferentes máquinas, utilizadas para servir os pedidos dos utilizadores. Esta tese tem como objetivo investigar a robustez das SNNs ao uso de esquemas de sincronização relaxados que têm o potencial de permitir uma elevada escalabilidade em ambientes distribuídos.

Palavras-Chave: Aprendizagem Automática, Sistemas Distribuídos, Redes Neuronais Recorrentes

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the years there have been a great number of breakthroughs [3, 4, 5, 6] 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 [2, 7]. First introduced in 1943, Neural Networks [8] have undergone dramatic advances over the last decades [4, 9].

These breakthroughs have been enabled also by the constant improvement of hardware technologies [10, 11] and by the availability of evergrowing volumes of data [12].

Among the many different types of ML approaches that have been investigated in the literature in recent years, Stateful Neural Networks (SNNs) [2, 4, 13, 14, 15, 6] have emerged as powerful tools 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 [15] and financial stock predictions [16].

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 [17] to weak eventual consistency [18]) vs (ii) the efficiency/scalability they provide.

1.2 Objectives

The goal of this dissertation is precisely to investigate the impact of use of different synchronization mechanisms on the prediction quality (e.g., loss, precision, recall) and efficiency (e.g., response time, throughput) when employed to regulate concurrent state updates of SNNs deployed in distributed production environments. The main hypothesis that is intended to study in this work is whether the use of relaxed synchronization techniques, e.g., bounded staleness consistency models [19, 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, it is planned to focus this thesis on the use case of SNNs employed in the context of financial fraud detection applications [2]. 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.

1.3 Contributions

This disseration's main contributions to the current state-of-the-art Distribution of Stateful Models are:

- Recreation of Feedzai's model [2] by implementing a model which uses different RNN states based on incoming transactions
- Expansion of the model by using additional layer to capture patterns of specific types of transactions.
- Experimentation on bounded staleness of such RNN States in a distributed setting using different approaches to merging RNN state.

1.4 Document Structure

This document is structured as follows:

- **Chapter 2:** Present the *Background & State of the Art* of works of ML in the context of sequential data and Distribution of such models, with a main focus on the ones that inspire and set a baseline for this dissertation.
- **Chapter 3:** Present a new approach to the state-of-art, specifying goals, the model *Architecture* and implementation.
- **Chapter 4:** Present the implementation's *Results* after running the system, while also performing some case studies in order to evaluate the performance and impact of some of this implementation's major features.
- **Chapter 5:** Take some *Conclusions* from the work made along this period and some takeaways for possible future improvements.

Chapter 2

Background

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 three main sections:

- **Stateless & Stateful ML approaches**, where the theoretical background will be presented for the ML algorithms that will be used in this proposal.
- **Intervaled Sequence RNN**, Feedzai's proposed model as solution to credit card fraud detection, the model in which this Thesis is based on.
- **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 [8], Neural Networks have since become the foundation for the state of the art in many application domains ranging from autonomous driving [21], finance [2], language processing [14, 15] and many others. This state of the art performance was enabled through more data availability [22], better frameworks [3, 23] 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) [4], Convolutional Neural Networks [9], LSTMs [24] and transformer networks [6].

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

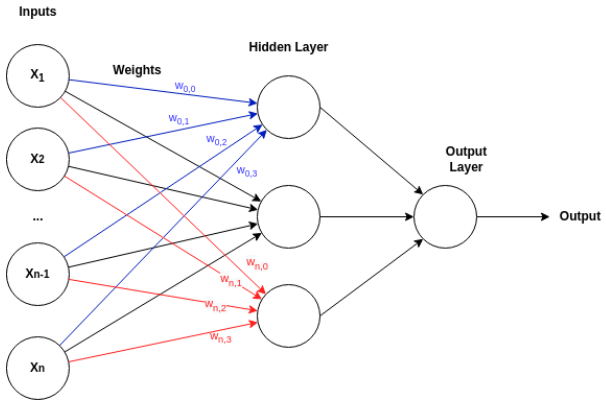


Figure 2.1: Structure of a Neural Network Architecture, each edge represents a number of adjustable weight parameters that can be tuned during training. The layers and neurons in each are variable depending on the problem, a single layer with three neurons was illustrated for simplicity. The number of neurons in the output depends on the kind of problem to be solved.

parameters, often called weights, with the objective to minimize the error for the current prediction as well as for future ones.

A simple or *stateless* neural network contains three layers: an input layer, hidden layer and an output layer, as shown in Figure 2.1. Each layer consists of individual units named *neurons* [25]. 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 [14].

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 $g : X \subseteq R \to R$. The output of a layer is described with the following function $f : X \subseteq R^{input\ size} \to R^{number\ of\ neurons}$:

$$f(x) = g(z(x)) \tag{2.1}$$

$$z(x) = Wx + b = \tag{2.2}$$

$$= \begin{bmatrix} w_{0,0} & w_{0,1} & ... & w_{0,n} \\ w_{1,0} & w_{1,1} & ... & w_{1,n} \\ ... & ... & ... & ... \\ w_{p,0} & ... & ... & w_{p,n} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ ... \\ x_n \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ ... \\ b_n \end{bmatrix} \tag{2.3}$$

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) [26].

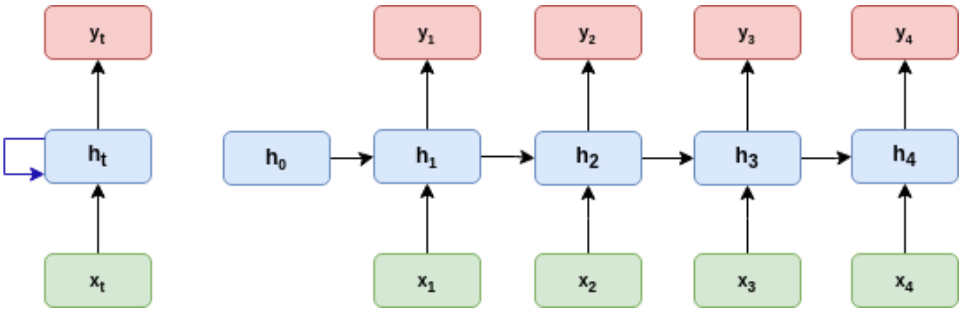


Figure 2.2: Architecture of a recurrent neural network, unrolled across several inputs

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

2.1.2.1 Recurrent Neural Networks

Also known as RNN [13], it is a SNN that builds on the original Neural Network architecture by aiming to capture relationships between sequential data. Some examples are words in sentences in Natural Language Processing [27], stock market returns, credit card transactions [2] 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 $f : X \subseteq R^n \rightarrow R^m$:

$$f(x) = Wh(x) + b \tag{2.4}$$

$$h(x) = g(Vx + c) \tag{2.5}$$

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:

$$f(x_t) = Wh(x_t) + b \tag{2.6}$$

$$h(x_t) = g(Vx_t + Uh(x_{t-1}) + c) \tag{2.7}$$

The added matrix state $h(x)$ stores patterns and relationships between the past inputs as seen in Figure 2.2, 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. Y. Bengio *et. al* [28] 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. Since then, discoveries have proposed possible solutions to deal with this issue , from different approaches to backpropagation[29] to new Neural Networks with more sophisticated architectures, such as LSTMs

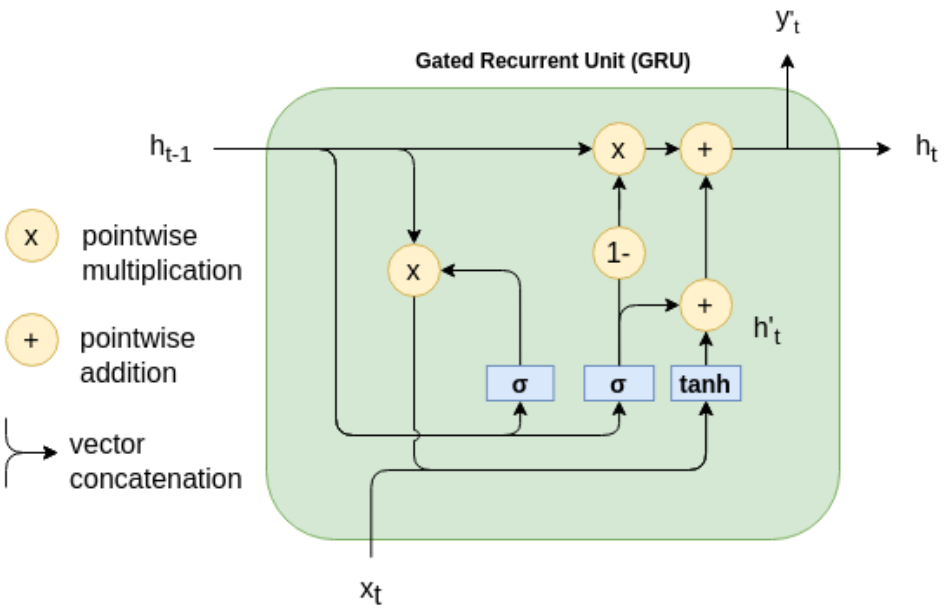


Figure 2.3: Architecture of a GRU layer

[24] and GRUs [4]

2.1.2.2 Gated Recurrent Unit (GRU)

Cho *et. al* tackled the aforementioned problem with the development of Gated Recurrent Unit (GRUs) [4], a recurrent unit where the state matrix described in equation 7 becomes instead:

$$h_t = \mathbf{u_t} \odot h'_t + (1 - \mathbf{u_t}) \tag{2.8}$$

$$\textbf{Candidate update: } \mathbf{h'_t} = tanh(Vx_t + U(\mathbf{r_t} \odot h_{t-1}) + b) \tag{2.9}$$

$$\textbf{Reset gate: } \mathbf{r_t} = \sigma(V_r x_t + U_r h_{t-1} + b_r) \tag{2.10}$$

$$\textbf{Update gate: } \mathbf{u_t} = \sigma(V_r x_t + U_u h_{t-1} + b_u) \tag{2.11}$$

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 2.3. 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.

2.1.3 Evaluation Metrics for ML Models

In order to measure the results during model training one needs to select the appropriate evaluation metric. Such metrics are dependant on the nature of the problem. Being fraud detection a binary classification problem, one could assume to use **Accuracy** as the primary metric for evaluation, however, such metric does not take the imbalance into account [30], i.e. the distribution of different label values in the dataset.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ noumber\ of\ predictions} \tag{2.12}$$

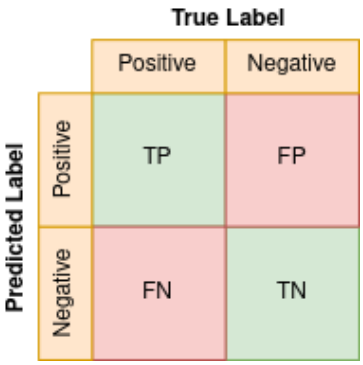


Figure 2.4: Confusion Matrix

It is common the presence of problems whose datasets that can be used for training are extremely imbalanced. For example, in credit card fraud detection the number of fraudulent transactions is much smaller compared to non-fraudulent [31]. Considering a dumb model, it could learn to classify all incoming transactions as ”fraud”. Even though this is a terrible model, the accuracy evaluation metric would indicate 99% accuracy, which may suggest the model's performance is great when it is actually of no practical use.

As such, in order properly evaluate the model in the presence of ”rare” labels, alternative metrics have come into widespread use for such problems:

2.1.3.1 Precision

Describes how well a model classifies a specific label correctly, with the terminology from the confusion matrix in figure 2.4, the formula for precision is the following:

$$Precision = \frac{TP}{TP + FP} \tag{2.13}$$

where TP is the number of true positive predictions and FP the number of false positive predictions. Precision provides an idea on the quality of the positive predictions given by the model. It is good to use when aiming to minimize false positive rates.

2.1.3.2 Recall

Describes the percentage of data the model correctly identifies as belonging to the class of interest, the positive class, displayed by the formula:

$$Recall = \frac{TP}{TP + FN} \tag{2.14}$$

Where TP is the number of true positives predictions and FN the number of false negative predictions. Useful when maximizing the positives predictions.

2.1.3.3 F1-Score

To aim the relation between precision and recall, F1-Score can be described as follows:

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{2.15}$$

The f1-score is the harmonic mean between precision and recall, with a value of range of $[0, 1]$. Used when precision and recall are important to the problem and it is necessary to reach the best balance between both.

Though these metrics help paint a clearer picture on the performance of the model. They only focus on the outcomes for each label, what is lacking is in measuring the **degree of confidence** in the model's prediction (expressed as probabilities).

To exemplify this, there could be two different models which the exact same number of correct and incorrect predictions, for both positive and negative labels, with the difference being the first model's probabilities displayed outputs being much closer to zero and one, compared to the second's. This displays model one to be much more confident in predicting its inputs in comparison to the second one.

2.1.3.4 The Receiver Operator Characteristic (ROC)

Perhaps the most common metric used for this purpose. The ROC curve and its associated Area under the ROC curve (AUC) [32] is a method to graphically represent the overall classification performance. AUC does not place more emphasis on one class over the other, so it is not biased against the minority class. As seen in figure 2.5, AUC is a graph in which TP rate is plotted on the y-axis and FP rate is plotted on the x-axis.

$$True\ Positive\ Rate = Recall = \frac{TP}{TP + FN} \tag{2.16}$$

$$False\ Positive\ Rate = \frac{FP}{TN + FP} \tag{2.17}$$

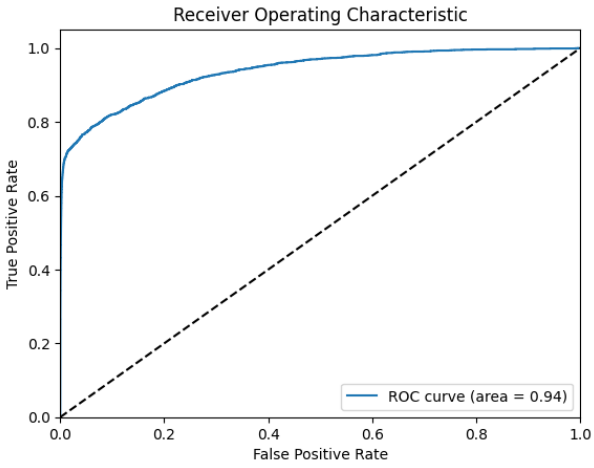


Figure 2.5: An example of a ROC curve, along with the respective AUC value

The curve is then obtained through iteration through a threshold of the minimum probability to consider a prediction positive, from zero to one, it possible to achieve the True and False positive rates at each threshold and subsequently plot the graph. The goal is to maximize the area the curve represents in the graph, i.e. the AUC, the higher the its value the better and more confident the model is at distinguishing between classes.

The combination of all the metrics mentioned here will be the ones used to measure the model's performance in terms of prediction ability and will be displayed for the different models in the following chapter for evaluation.

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 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* [1] 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.

2.2.1.1 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 [3] help create DNNS systems by employing algorithms to determine efficient vertical partitioning structures with its TensorFlow serving component [33]. 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.

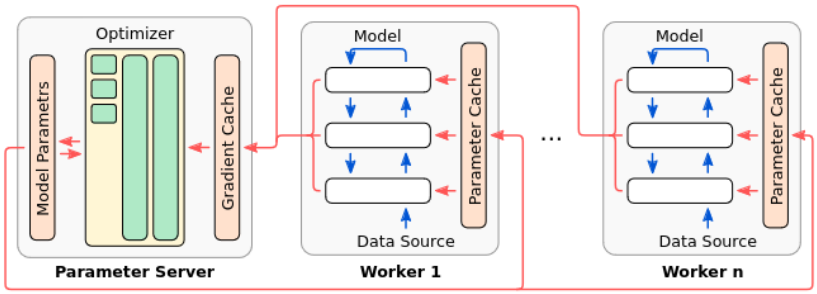


Figure 2.6: Data flow in a centralized system, arrows in blue indicate the gradient computation cycle, arrows in red indicate the messages exchanged between machines to update model parameters. Workers evaluate the model to generate gradients. The parameters server consumes them to update the model [1].

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 [26, 34, 5].

2.2.1.2 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 2.6 and 2.7.

Centralized updating: Updating of the parameters is done in a central, single instance (often called the *parameter server* [5]. 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 2.6 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.

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. [35]

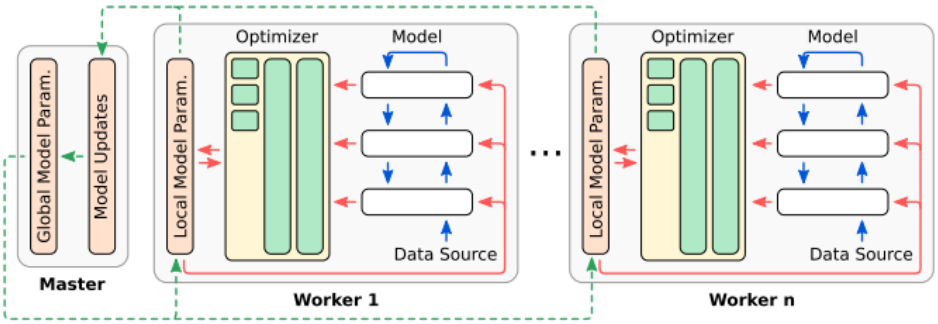


Figure 2.7: Data flow in a decentralized system, arrows in blue indicate the gradient computation cycle, arrows in red indicate the flow of information for parameter updates and the arrows in green are messages communicated in the network. The master node forms the next global model state by combining local model replicas [1].

2.2.1.3 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 [36]. Simultaneously, GPU-based ML greatly increases computation speeds, leading often to network communication between machines becoming the primary bottleneck in distributed setups [23]. 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 system Allows 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.

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 [19, 20, 1]. 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.

2.2.1.4 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 [36], 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 [19, 34], 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:

$$||t_{fastest} - t_{slowest}|| \geq \delta_{max} \tag{2.18}$$

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. [37].

Delay bounds (e. g. Stale Synchronous Paralell (SSP)) [19]: 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 [38, 39]. 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* [2] 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 2.8.

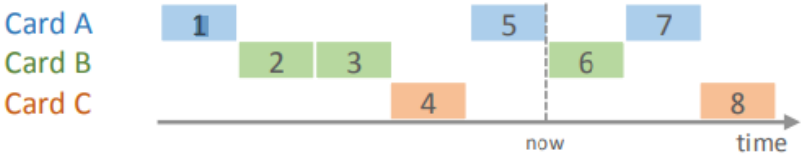


Figure 2.8: Interleaved transaction history of three cards [2].

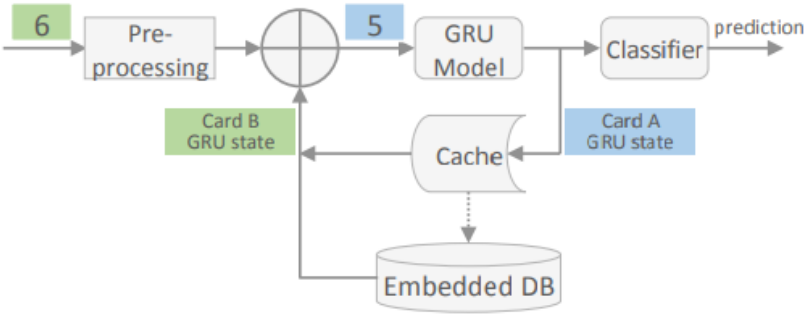


Figure 2.9: At run-time, the system joins transaction 6 with the previous Card B GRU state (last changed in transaction 3). Previously, it processed transaction 5 and stored the Card A GRU state in an embedded DB and memory cache [2].

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 2.9). After processing the state it is updated in the database for future use.

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* [2] 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* [2], which, as discussed, limits requests that share any state to be executed on the same machine.

2.2.2.1 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 [38].

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 [39],

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.

Chapter 3

Work Proposal & Implementation

As discussed in the previous section, the problem of scaling 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 scalability in scenario 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 that maintain replicas of the same shared state. With the goal being to determine whether the use of synchronization techniques on RNN state ensure promising results, similar to the ones used in the training phase of DNNS, that can be just as well be effectively end efficiently employed in this scenario.

Throughout this chapter it will presented how this thesis explores the previously mentioned problem. Starting with the context of the problem, the dataset used for such context.. Afterwards, we dive into the implementation of the model, describing how the models were trained and set up for inference in a production scenario and finally, their deployment in a distributed system, highlighting its architecture and methods chosen.

3.1 Dataset

The application domain that this thesis will work on is Credit Card Fraud detection, inspired by the work proposed by Feedzai [2]. With it, comes with the choice of the dataset, an important step, which has important requirements in order to be best suitable for the experiments that we intend to conduct. These requirements are:

- Be a time series dataset: a time feature had to be present to be able to define the problem forecasting problem, suitable for SNNs to predict the classification of a transaction as fraudulent or not based on past transactions.
- The dataset should include at least a feature to logically associate the transactions submitted by different users. Possible example in this domain are features like the target merchant of a transactions, the category of goods purchased or the geographical region in which transactions are submitted. The existence of this type of features allows then for building SNN where part of the state is used to correlate the activities of transaction submitted by different users but that share some common aspect (e.g., are transactions used to purchase the same category of goods).
- The dataset be based on real world data, for the experiments be as realistic as possible, while avoiding the bias of simulated data.

However, it was not possible to find a publicly available dataset containing all 3 characteristics, in particular the third requirement, as real world data is only made available after feature engineering, in order to protect the privacy of the users that performed those transactions, like the IEEE fraud dataset [40]. Such feature engineering looses the possibility of understanding whether some of the features in the dataset can be logically used to correlate the activities of different users (Requirement 2), making the dataset unsuitable. With this problem revealed, it was necessary to compromise and proceed with finding a synthetically generated dataset, which satisfies the remaining requirements. This led us to opt for using the *Credit Card Transactions Fraud Detection Dataset* [31]. A generated dataset using the SparkovProgram. Details about this dataset can be seen in section 3.1.2.

3.1.1 Data Processing

An important step in Machine Learning is the preprocessing of data before training and evaluation of the model, as the data needs to be converted to an adequate format. To this end, new features are created and existing ones are transformed into appropriate numerical formats depending on their semantics.In our work we used the same transformations as the ones performed by Pedro Abreu *et al.* [2]:

3.1.1.1 Numerical Features

Transformations for numerical fields use on of two possible strategies:

1. Z-scoring with outlier clipping for features with distributions that are not very skewed and with which it is expected the fraud risk to vary smoothly, such as the amount in US dollars:

$$x_{n_i} = \frac{x_i - \mu_{x_i}}{\sigma_{x_{x_i}}}$$

$$x'_{n_i} = max(min(x*_{n_i}, T_o), -T_o$$

Where μ_{x_i} and σ_{x_i} denote, respectively, the mean and standard deviation of the values of features x_i in the training set, and T_o is the number of standard deviations from the mean above which to consider a value to be an outlier (the value chosen as suggested in the paper was $T_o = 3$).

2. Percentile bucketing for features with multimodal distributions, or with which it is not expected the fraud risk to vary smoothly, such as latitude or longitude. Percentile bucketing amounts to creating bins between every pair of consecutive percentiles computed from the training set, and transforming feature values to the index of the bin in which they land:

$$x'_{n_i} = \begin{cases} 0 \text{ if } x_i < P^1_{x_i} \\ 1 \text{ if } P^1_{x_i} < x_i < P^2_{x_i} \\ ... \\ 99 \text{ if } P^{99}_{x_i} \leq x_i \\ 100 \text{ if } x_i \text{has a missing or invalid value} \end{cases}$$

Where $P^k_{x_i}$ denotes the k^{th} percentile computed over the values of feature x_i in the training set. These transformed features are interpreted later as categorical.

3.1.1.2 Categorical Features

Each categorical feature is indexed by mapping each possible value into an integer based on the number of occurrences in the dataset. For a given categorical feature, x_{cj}, the l^{th} most frequent value is mapped to the integer $x'_{cj} = l - 1$. All values below a certain number of occurrences map to the same integer l_{max}.

3.1.1.3 Timestamp features

The event timestamp feature is transformed into the sine and cosine of its projection into daily, weekly, and monthly seasonality circles, i.e., a timestamp x_{t_k} generates:

- h_k = hour_from_timestamp$(x_{t_k}).\frac{2\pi}{24}$;
- d_{w_k} = day_from_timestamp$(x_{t_k}).\frac{2\pi}{7}$;
- d_{m_k} = day_of_month_from_timestamp$(x_{t_k}).\frac{2\pi}{30}$;

All of the features are transformed through the z-scoring and outlier clipping process previously described before. It is not applied yearly seasonality since the dataset used only spans a maximum of one year

3.1.2 Data Profiling

After defining the techniques that will be used to engineer the data, the profiling and identification of the features and which technique was suited for their characteristics. In table 3.1 each feature is identified, described and the method chosen for its preprocessing.

It is important to note the dropping of features due to their direct correlation between them, such as the first and last name, in which the credit card would indicate directly the person it is referring to. As well as Street, Zip Code, City and State features, redundant with the presence of geographical coordinates. For ease of development of this dissertation, the transformations were applied to the totality of the dataset before spliting the dataset into training, validation and test sets.

Feature Name	Type of Feature	Action used
transaction date	Timestamp	Sin and cosine projections
credit card number	Categorical	Map categories to a unique value
merchant	Categorical	Map categories to a unique value
category	Categorical	Map categories to a unique value
amount	Numerical	Z-Scoring
first name	Categorical	Dropped
last name	Categorical	Dropped
gender	Categorical	Binarize to zero or one
street	Categorical	Dropped
city	Categorical	Dropped
state	Categorical	Dropped
zip	Numerical	Dropped
latitude	Numerical	Feature Bucketing
longitude	Numerical	Feature Bucketing
city population	Numerical	Z-Scoring
job	Categorical	Dropped
date of birth	Timestamp	Convert to age
transaction number	Categorical	Feature dropped
unix time	Numerical	Create feature, time between transactions
merchant latitude	Numerical	Percentile bucketing
merchant longitude	Numerical	Percentile Bucketing
is fraud	Label	None

Table 3.1: Description of features in the dataset and transformations performed

Feature	Nr. unique values	Mode
Credit Card Number	999	693
Merchant	4392	6262
Category	14	188029

Table 3.2: Table indicating characteristics regarding features with potential to be used as shared state.

In combination to preprocessing, it is necessary to choose which of features will be used as the parameter for creating different shared state profiles. Two features will be chosen since, as we will see in the next section, our proposed architecture will allow the presence of two different shared states. The features should be ones that are easily categorized, with potential to define profiles in transactions of the same feature value. One of the features to be used will be the credit card number of the user, the same feature used by Bernardo *et al.* [2]. As for the second feature, the two features with most potential are the merchant and the category of transaction, as for this work, we opted to use category as the second feature due to its smaller range of unique values in the dataset, as we can see in figure 3.1 and table 3.2.

3.2 Model Architecture

Usually SNN models use a sequence of inputs as the main input of the model in production, with the purpose of building the RNN state for the model's last input classification [27, 15, 39]. However, in a context where large amounts of transactions arrive per second in production, processing an entire sequence at incoming input will generate unacceptable overhead. This problem is amplified when the model has a shared state, which can be affected by any previous transaction, with respect to the scenario in which (as in Bernard et al. [2]) the model maintains state only at the level of each credit card (as the state's value is affected only by previous transactions issued by the same credit card holder). To meet

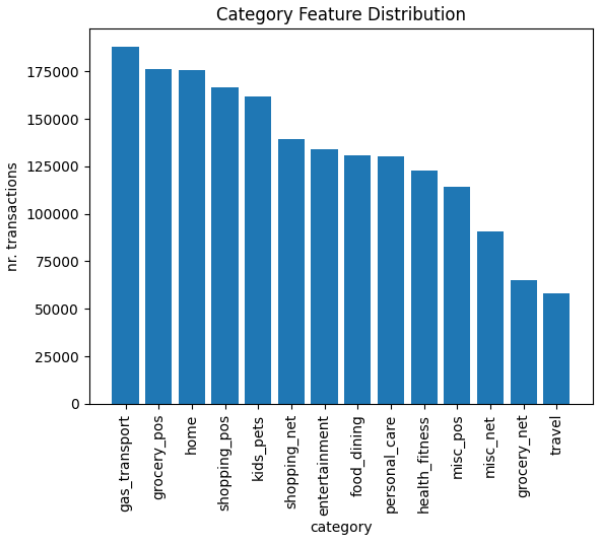


Figure 3.1: Chart plotting the distribution of feature *Category*.

these requirements, the model used in production saves the model state obtained after processing any query; when a new query is received for a new transaction T the state associated with T is loaded before querying the model for T. This allows for using as input for the production model only the single transaction for which a prediction is needed.

The SNN that was considered to pursue the goals of this dissertation is an architecture based on the one proposed by Bernardo *et al.* [2], but extended to enable the sharing of additional state between transactions. The key difference with respect to the original network is the use of 2 GRU cells in parallel, which are meant to store, respectively:

1. The state associated with all transactions so far issued by this credit card, which is meant to capture the spending habit of each holder (as in Bernardo *et al.* [2]);

2. The state with all transactions so far issued that share a common value for a feature. In this case, the feature chosen was the category of purchase of the transaction (see table3.1).

To best assess the impact of distributing shared state, a total of four models were developed. The rationale was to consider an initial very simple model and then consider more complex models in order to evaluate the effects of synchronization of shared state over a set of diverse architectures. Specifically, the first model (Figure3.2) simply feeds the input features to the two GRU cells responsible for maintaining the state of a given credit card and of a category, respectively. The output of the GRU cells is then concatenated and fed to a very simple classifier composed of a single neuron. The output of this neuron produces the score that serves to classify the transactions as fraudulent or legitimate.

Next we considered the possibility of enriching this base model architecture in two orthogonal ways: 1) adding a skip connection that concatenates the input features to the output of the GRU cells (Figure 3.3), as also done in Bernardo *et al.* [2]. 2) Considering a more complex final classifier, obtained by adding an additional dense layer right before the final output neuron (Figure 3.4). By considering models with different degrees of complexity, our goal is to investigate whether the impact of state synchronization methods is dependant of the model's topology. The final model (Figure 3.5) builds on the previous two models by adding both additions together. We have opted for using GRU [4] instead of alternative stateful cells, and in particular LSTM [24], for efficiency reasons and GRU being the same approach used in Feedzai's paper [2]. Each GRU cell has a size of 48 neurons and the additional feed-forward layer has a size 24 neurons. The reasoning behind the size of the layers was simplicity, it is the goal to

develop models whose shared state have a significant impact on performance. Larger and more complex models could compromise this goal by generating models in which the RNN state may contribute little to the output of each transaction.

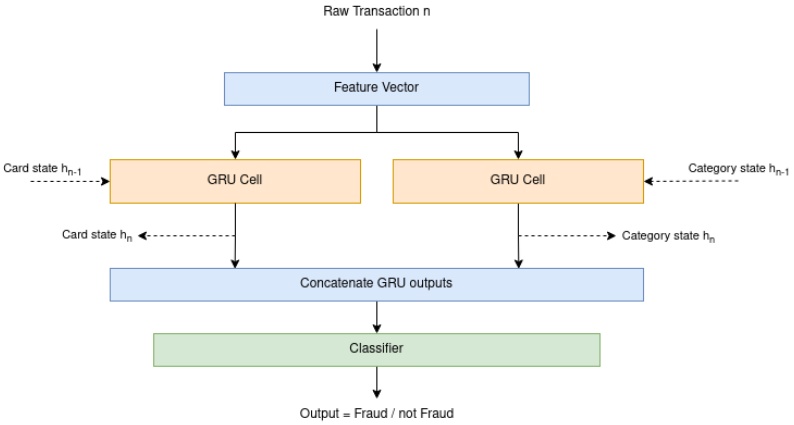


Figure 3.2: Illustration of the architecture of the model with two GRU cell in parallel, whose outputs are concatenated and fed to the classifier to generate an output.

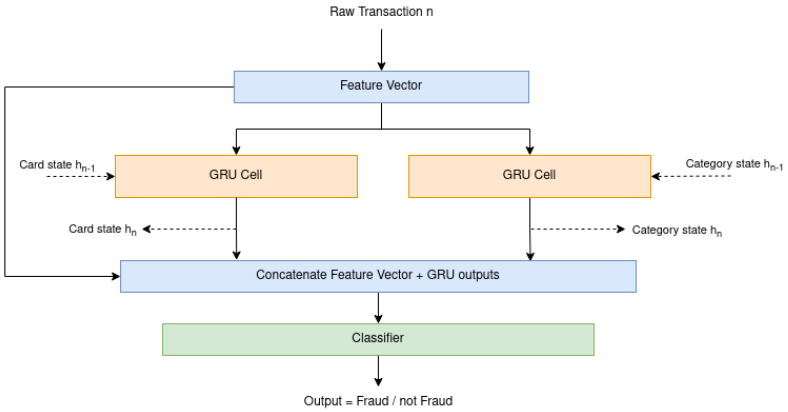


Figure 3.3: Illustration of the architecture of the model with double GRU and a skip connection, where additionally, the inputs are concatenated also with the GRU cell outputs

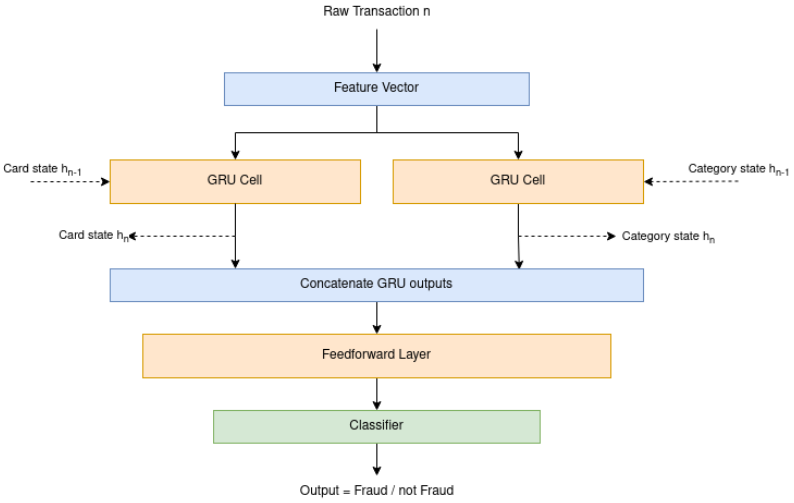


Figure 3.4: Illustration of the architecture of the model with an additional feedforward layer.

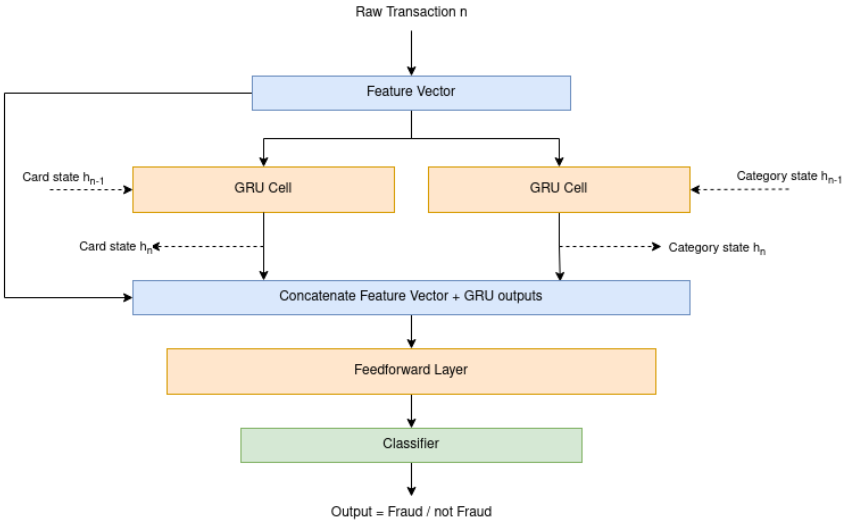


Figure 3.5: Illustration of the architecture of the model with two GRU cell in parallel, whose outputs are concatenated and fed to the classifier to generate an output.

3.2.1 Training

After fully setting up the data and defined the model, one's finally ready to train the learning model. The models' weights were trained using Binary Cross Entropy as a loss function, suited for classification problems. Updated using ADAM optimizer, with the default learning rate of 0.001. The batch size used for training was 1024. The first step is to split the data into two distinct datasets and a testing set. The former will correspond approximately to the first 80% of the original dataset and will be used for training the model. The latter, the remaining 20%, will serve to test the results of of the fitted model. The two sets must be completely separate from each other, to prevent information leakage.

However, to choose the model best suited to go to production, the test set can not be used to measure its performance. The goal of the test set is to simulate a production environment with never seen data, making the test set impossible to use. To validate the model, we have no choice but to part of the training set as validation data to evaluate the model's performance. One could simple partition part of the dataset to use as a validation set, with the caveat that it only works if the dataset is large enough to deal with possible **overfitting**, which is when the model fits the model parameters too much around the training data and achieves poor generalization capabilities, performing poorly in unseen data. This issue is magnified when the dataset is highly imbalanced, increasing dramatically the probability of overfitting.

To solve this issue one has to find ways that help generalize the model, increasing inference ability. One of the most popular solutions, specially on imbalanced datasets, is through the use of *k-fold cross validation*.

3.2.1.1 K-Fold Cross-Validation

Cross validation is a re-sampling procedure used to evaluate machine learning models on a limited data sample. The training set is divided into k subsets or folds. The split can be seen in figure 3.6. The model is trained and evaluated k times, using a different fold as the validation set each time. Performance metrics from each fold are averaged to estimate the model's generalization performance.

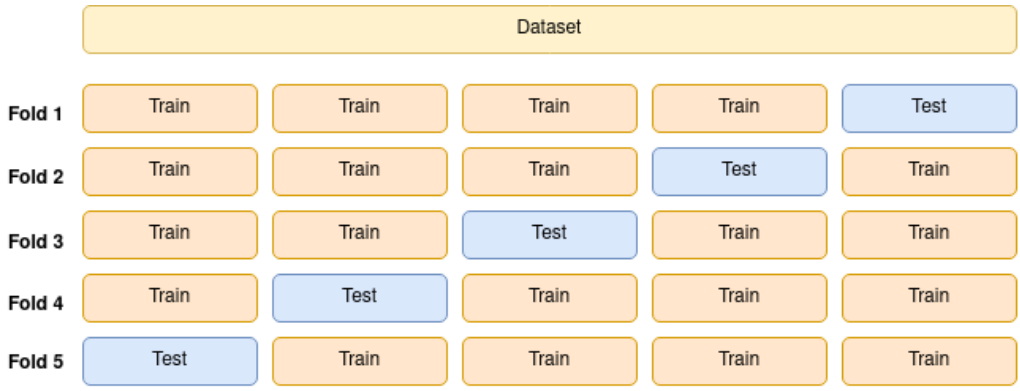


Figure 3.6: Illustration of train/test splits in each fold using k-fold cross validation, where $k = 5$

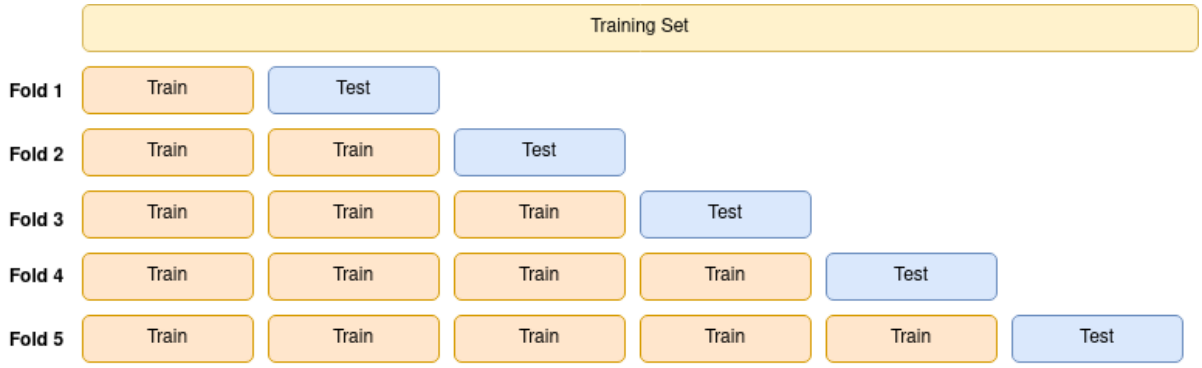


Figure 3.7: Illustration of train/test splits using time series split k-fold cross validation, applied on the training set

3.2.1.2 Time Series Split Cross-Validation

However, in the case of time series, the cross-validation is not trivial. One cannot choose random samples and assign them to either the test set or the train set because it makes no sense to use the values from the future to forecast values in the past. In simple words we want to avoid future-looking when we train our model. There is a temporal dependency between observations, and we must preserve that relation during testing.

The method that can be used for cross-validating our time-series model is cross-validation on a rolling basis. Starting with a small subset of data for training purpose, forecast for the later transactions and then checking the accuracy for the forecasted transactions. The same forecasted transactions are then included as part of the next training dataset and subsequent transactions are forecasted, just as described in figure 3.7

Note this procedure/split is only applied on the **training dataset**, after the initial train/test split. Resulting in k folds of training and validation set. Adding it all up, a full training of a single model will consist of training the model k times with different training and validation sets and averaging the evaluation metrics of each fold to obtain the overall performance of the model. The different models will each do this process and be compared with each other.

3.2.1.3 Sequence Generation

Although the models being built aim to be able to capture an entire history of transactions in production, such thing can not be reproduced during training. Indeed, providing a sequence of possibly thousands of transactions to just make a single prediction and then calculate its gradient becomes very inefficient.

To tackle this issues, the approach typically adopted [2] consists in using sequences of fixed size during training: the sequence sequence size should be set short enough to avoid excessive overhead, but also large enough to capture sufficient historic background for the final transaction in the sequence, i.e., the one we want to train the model to predict correctly. In particular, given the specific stateful models that we consider in this work, the sequence should contain sufficient information to allow tracking previous relevant patterns across transactions issued on the same credit card and targeting the same category of goods.

The sequences should be built such as they contains transactions that can build profiles the respective credit card number and category labeled in the last transaction of the sequence i.e., the transaction to be predicted as fraud or non-fraud. For that, we sequence must contain transactions that contain the respective feature values. However, it is not possible to just use transactions with that have both the correct credit card number and category, because the sequence then could miss out on transactions, for example, that belonged to the same credit card, but since it had a different category of goods, it would be discard. As we can see in Figure 3.8, each sequence is generated by using transactions, ordered chronologically, all with the same credit card number. A sequence can have different categories, but only the transactions which have the same final category will update the state that will be used for the prediction, as such the remaining states will not be used. This one of the factors using category of purchased goods as the feature for the second shared state proved to be useful. Its low range of possible values gives a high probability that plenty of transactions in the sequence will have same category as the transaction to be predicted, allowing the sequences to have the highest amount of transactions which will up useful for contribution to the final predictions. According to Bernardo *et al.* [2], the sequence size shown most effective in training were between 100 and 140 transactions, which is also enough for a high probability that several transactions will contain the category relevant for the final prediction.

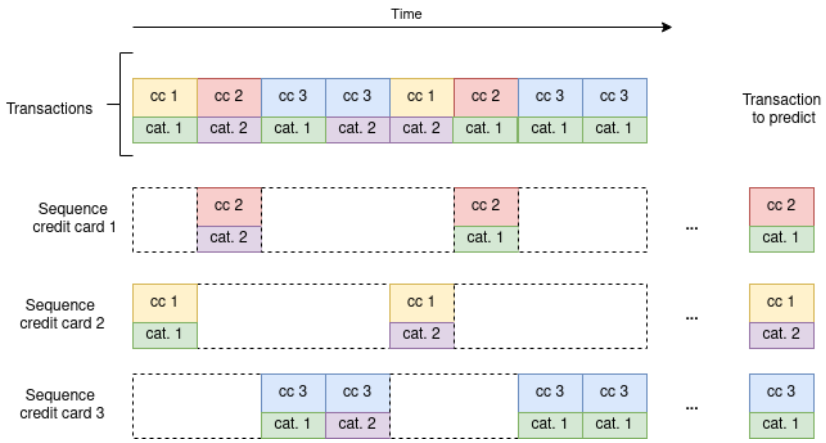


Figure 3.8: Illustration representing sequence generation. A sliding window iterates through all transactions belonging to a single credit card and generate sequences accordingly. Each larger block the credit card associated with the transaction, while the smaller block represents the category. Sequences all have transactions with the same credit card, but can have different categories

One other challenge is when training our proposed models in batches. By design, the models will only have one RNN state for each credit card/category When training. However, the model processes training input in *batches* of inputs, or in this case batches of sequences, to update the gradients. A batch of sequences is processed in parallel to allow faster training. This is an issue in our case, since there is the possibility that at a given index in the sequence, when executing in parallel, two transactions might have the same credit card or category of purchased goods. As a result, they will change the same state, which is not intended. To counteract this, the GRU cells were replicated as many times as the batch

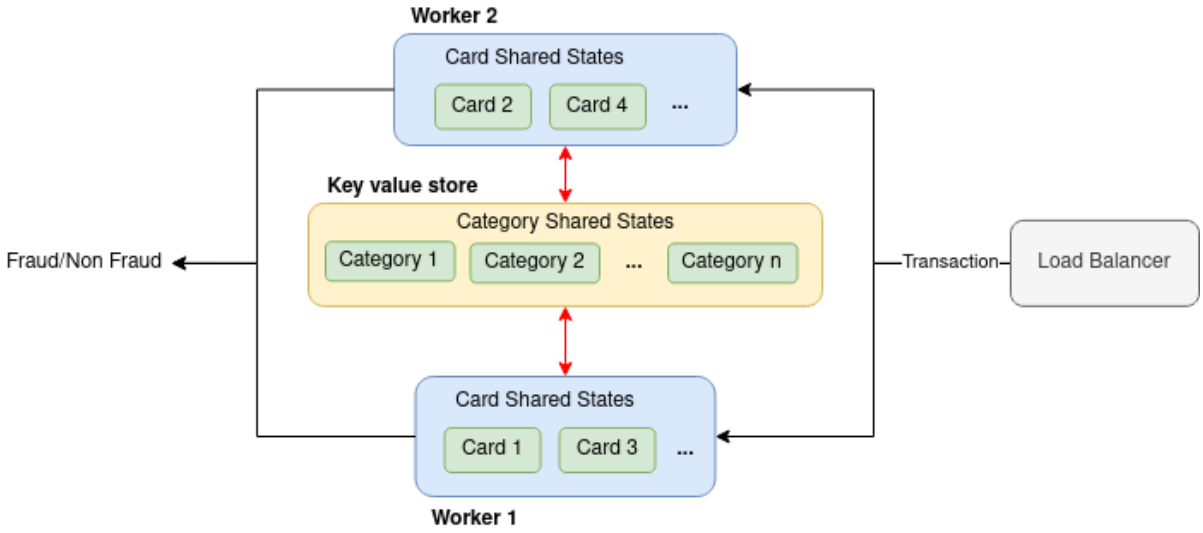


Figure 3.9: Illustration of the proposed DNNS. The arrows in red indicate communication with the key value store to manage RNN state.

size, such that for each sequence in the batch, the shared states it will remain isolated to that specific sequence alone.

3.3 Model in Production

With a fully trained model, ready for production, it is up to us to design the system that will host such model. In such a system, one aims to maximize rate of processed transactions per unit of time. To achieve that, one can scale out the system by adding workers to process transactions in parallel, ensuring simultaneously that the quality of models' prediction does not degrade. As mentioned before in chapter 2, the scaling out of Stateful models requires careful management of the internal state. Our model, in production, has two types of shared states it needs to maintained: the shared states of credit cards, and the shared states of transaction categories. While it is possible to isolate the credit card state to single worker instance, as done in the solution applied by Feedzai [2], this becomes impossible with category states. This is the exact goal of this thesis, to manage such state, maximizing the system's performance, while minimizing predictive quality loss.

3.3.1 Proposed System Architecture

The problem will be tested using DNNS as seen in Figure 3.9. Each worker will support a model built using *Tensorflow* [3], where it performs inferences on incoming queries that will update the shared states they have stored internally for the respective credit card and category. The workers will need to exchange information regarding their states for categories, this exchange can be implemented in a number of ways, e.g., via direct message passing among the workers using a message passing interface or via a shared datastore state. In this thesis, we opted for using a popular distributed key-value store, Redis, using keyspace notifications to notify each workers of incoming changes in the database. Redis was chosen for its high performance and low latency in writing and reading when deployed in production environments.

In this system, it is assumed no faults will occur, so it is not necessary for our system to deal with worker failures, communication failures, network delays, and other unforeseen issues caused by them,

to keep the work of this thesis focused on the impact of staleness in RNN state. Due to time constraints, the experiments were ran on a single machine using multiprocessing, which imply some limitations regarding our study, mainly regarding the lack of network latency in the system, so absolute throughput may be overestimated. Also, running multiple workers on the same machine might affect throughput results, since the machine's resources will be split across processed, yielding less realistic results. It is with these limitations in mind that Redis was used as a communication medium across the nodes. Allowing the current implementation to be easily deployed over a distributed system with no or little adaptations for future work.

To simulate the load balancer aspect of the system, the test set was split among the different workers based on their credit card id. if the remainder of the division between the card id and the number of workers present in the system matched the id of the worker, the transaction would be processed by that respective worker. This ensures all transactions of the given credit card are isolated to a single worker, as intended. For ease of development and to ensure all transactions would run concurrently, the dataset was pre-loaded and split on each worker before beginning execution of the system.

3.3.2 Synchronization

A key aspect of the system is how synchronization amongst workers will occur and, as discussed in the background, can be performed either asynchronously or asynchronously. In both approaches, a worker decides to trigger the synchronization process after having processed a fixed, predetermined number of input transactions, which we will call threshold. The key difference lies in if the synchronization process is blocking for the worker or not. As seen in Figure 3.10, the asynchronous approach does not block processing and synchronization is performed in parallel. Conversely, the synchronous approach blocks the process making sure that the synchronization is complete before resuming its task; this allows processing of transactions to not be interrupted, hence improving throughput. However asynchronous approaches are expected to lead to larger divergences of the states at different replicas, which may possibly reduce the predictive quality. Unfortunately, due to time restrictions, it was not possible to develop for this thesis system which could support asynchronous merging of shared state. Nevertheless, asynchronous approaches should be taken in consideration for possible future work to be done.

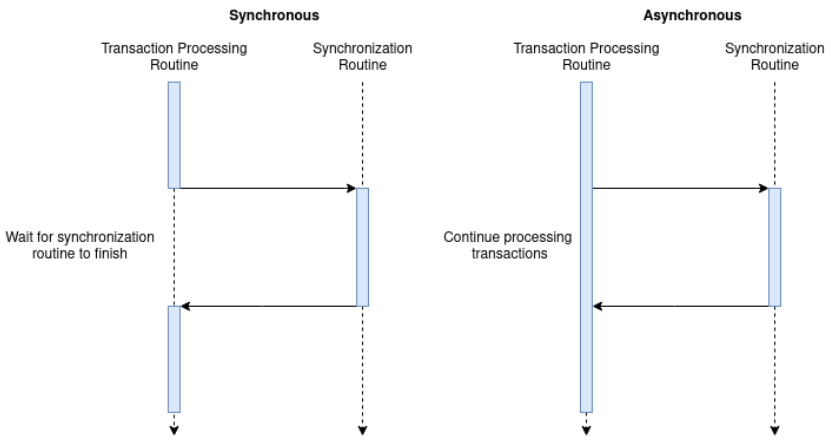


Figure 3.10: Illustration synchronization techniques in a single worker.

Having said this, the algorithm that would implement such merging algorithms, synchronous and asynchronous, the pseudocode describing in more detail the synchronous and asynchronous approaches are reported in Algorithms 1 and 2, respectively.

Algorithm 1 Algorithm for merging shared state synchronously

Synchronous Merging of Shared State

```
1: procedure PROCESS TRANSACTIONS
2:     staleness ← 0
3:     while input do
4:         h_t ← h_{t−1}(input)
5:         staleness ← staleness + 1
6:         if staleness ≥ threshold then
7:             staleness ← 0
8:             Send_to_workers(data)
9:             h_t ← Merge_states_from_workers(h_t)
10:        end if
11:    end while
12: end procedure
```

Algorithm 2 Algorithm for merging shared state asynchronously

Asynchronous Merging of Shared State

```
1: procedure PROCESS TRANSACTIONS
2:     staleness ← 0
3:     while input do
4:         h_t ← h_{t−1}(input)
5:         staleness ← staleness + 1
6:         if staleness ≥ threshold then
7:             staleness ← 0
8:             Send_to_workers(data)
9:         end if
10:    end while
11: end procedure
12: procedure RECEIVE UPDATES FROM WORKERS CONCURRENTLY
13:    while update do
14:        h_t ← merge_operation(h_t, update)
15:    end while
16: end procedure
```

In both synchronization approaches, they assume that once an update (in the case of asynchronous) or a set of updates (synchronous) is received the worker has access to a *merge_state_from_workers* function that it can be used to update its own replica of the shared state. However, defining such merge function not trivial, with the sole reason being, different transactions are sent to different workers in production, those transactions could all belong to the same category, requiring to update the internal state for that given category. As discussed in Chapter 2, given state at any given time t is dependant on the state at $t-1$ and the current input at time t: $h_t = h_{t-1}(x_t)$. Given two workers p and n, their states at time t are written as h_t^n and h_t^p, respectively. Assuming each worker will receive different transactions by the load balancer, the resulting internal state for each worker will have diverged from the original state value, this means $h_t^n \neq h_t^p$. However, it is not possible to discard the state, even when they have diverged, since it would result in the loss of important information that helps predict further transactions in the future.

Gathering all this key points, one could call this not a "synchronization" problem, but a "approximation" problem, with a set of state vectors, find a method to merge them all onto a single state, as closest to the state's representation as if all transactions were processed sequentially. We considers two possible candidates that are based on the idea of i) summing the variations in the states at each node with relation to previous synchronization round and ii) averaging the states maintained at all workers. Each of these two methods is described in the following sections.

3.3.2.1 Summing deltas

The first proposal for a merge operation is based on the idea of using the difference between the current state and the previous one, which we refer to as *delta*. This delta in the context of a single model contains all the information transferred to create the model's most recent state. With this idea, it could be possible to share this delta between workers and be able to transfer the same information, this is the goal of this approach. To determine the information accumulated at two different times, one would just need to to calculate the difference between the two states, both belonging to the same credit card or category, but at different points in time. Recall that each of these states is a vector so the resulting delta is a vector as well, this value is then sent to a worker and added to the current state. However, one can not simply add delta, since the GRU cell uses Hyperbolic Tangent and Sigmoid functions as activation functions to generate the state, resulting in a state whose values are in the range $[-1, 1]$, simply adding deltas can lead to breach these bounds, ruining future feed forward computations. To combat this, addition is bound in order to not break this rule. Given two workers n and p, after reaching a certain threshold, worker p sends its delta, Δ_p, to worker n, which will update its internal state h_n based on the equations 3.1 and 3.2. If the synchronization approach is synchronous, the worker will fetch the deltas from all workers at once, if it is asynchronous, it will apply deltas as updates arrive.

$$h_n = max(min(h_n + \Delta_p, 1), -1) = \tag{3.1}$$

$$= max(min\left(\begin{bmatrix} h_n^1 \\ h_n^2 \\ ... \\ h_n^n \end{bmatrix} + \begin{bmatrix} \Delta_p^1 \\ \Delta_p^2 \\ ... \\ \Delta_p^n \end{bmatrix}, 1\right), -1) \tag{3.2}$$

3.3.2.2 Averaging State

The second approach aims to compute the mean of all shared states present in the system, the objective being to reach for a good representation of the credit card's or category's state based on the states based in each workers. The main formula to merge states, defined as m, n being the number of workers in the DNNS and p the shared state's size as described in equations from 3.15 to 3.17.

Algorithm 3 Asynchronous Averaging of states

procedure WRITE STATE
  $streaming\_versions \leftarrow \{\}$
  $staleness \leftarrow 0$
  while *input* **do**
    $h_t \leftarrow h_{t-1}(input)$
    $staleness \leftarrow staleness + 1$
    if $staleness \geq threshold$ **then**
      $staleness \leftarrow 0$
      $Send\_to\_workers(h_t,)$
    end if
  end while
end procedure
procedure RECEIVE STATE FROM A WORKER(h_p, $version$)
  if $version$ not registered **then**
    $streaming\_versions[version] \leftarrow 0$
  end if
  $count = streaming\_versions[version]$
  $value1 \leftarrow h_t.\frac{count}{count+1}$
  $value2 \leftarrow h_p.\frac{count}{count+1}$
  $h_{t+1} \leftarrow value1 + value2$
  $streaming\_versions[version] \leftarrow streaming\_versions[version] + 1$
end procedure

$$m(h_1, h_2, ..., h_n) = \frac{h_1 + h_2 + \ ... \ + h_n}{n} = \tag{3.3}$$

$$= \left(\begin{bmatrix} h_1^1 \\ h_1^2 \\ ... \\ h_1^p \end{bmatrix} + \begin{bmatrix} h_2^1 \\ h_2^2 \\ ... \\ h_2^p \end{bmatrix} + \ ... \ + \begin{bmatrix} h_n^1 \\ h_n^2 \\ ... \\ h_n^p \end{bmatrix} \right) \frac{1}{n} = \tag{3.4}$$

$$\begin{bmatrix} \frac{h_1^1 + h_2^1 + \ ... \ h_n^1}{n} \\ \frac{h_1^2 + h_2^2 + \ ... \ h_n^2}{n} \\ ... \\ \frac{h_1^p + h_2^p + \ ... \ h_n^p}{n} \end{bmatrix} \tag{3.5}$$

Translating this formula directly into the DNNS results in a **Synchronous** system, because to perform this operation we would need block and fetch the RNN states of each worker before performing the merge operation. This blocking behavior can cause slight overhead in the system, however there is a way to make use of this method while maintaining the system **fully asynchronous**, through a streaming average: the average is calculated as new values reach the worker:

$$mean_{t+1} = mean_t.\frac{count}{count + 1} + \frac{x}{count + 1} \tag{3.6}$$

$$h_{t+1} = h_t . \frac{count}{count + 1} + \frac{h^p_{t-1}}{count + 1} = \tag{3.7}$$

$$\begin{bmatrix} h^1_t \\ h^2_t \\ ... \\ h^p_t \end{bmatrix} . \frac{count}{count + 1} + \begin{bmatrix} h^1_{p,t-1} \\ h^2_{p,t-1} \\ ... \\ h^p_{p,t-1} \end{bmatrix} . \frac{1}{count + 1} \tag{3.8}$$

To find the new average for the incoming mean at $t + 1$. It is necessary to count the number of numbers seen so far in the stream of values for the mean, lets call it $count$, the previous average $mean_t$ and x be the new number being added. At a each threshold, the internal state will be shared to workers in the system, each worker will write multiple times its internal state in the database, resulting in different *versions* present. As each worker receive the different updates, each will perform the streaming average according to the set of internal states of each worker, all with the exact same version, so the streaming average is not affected by future versions of the shared state. Only will apply merging operation for a new version after processing the full streaming average. The resulting algorithm associated to this asynchronous approach of averaging states will look like algorithm 3.

Chapter 4

Evaluation

In the previous chapters we described the architecture and inner workings of our proposed solution for managing shared states in stateful ML models that are deployed over a set of distributed nodes, minimizing model performance loss. It is, therefore, of the utmost importance to assess the quality of the developed solution by performing evaluation tests and take conclusions according to the obtained results. In this chapter, we will evaluate and discuss our solution. We will also analyze the quality and practical viability of the generated solutions.

When training, the models use sequences to predict transactions and thus learn how to update the model's internal state. When the model goes into production or is evaluated in the test set, the model stops receiving sequences and its inputs become single transactions. If the input format remained the same, there would be no issue, as by the time the model predicts its output, it would have updated the internal model state a shared state strong enough to make a fair prediction. This is not the case here, the model in the beginning of the test set, it would initialize with no state whatsoever, on credit cards that were already present in the training set. This is information the model could use for its predictions. For this purpose, the models, before evaluating, process the entire training set of transactions beforehand to compute the internal model states shared state for each credit card and category. This states are then stored with the saved models, and are made ready for evaluation, in offline and distributed settings.

4.1 No distribution baselines

We begin experiments with training on each of the developed models, for this, each model was trained along 20 epochs. At the end of each epoch of training, the models were evaluated on the test set, using metrics mentioned previously in chapter 3.2. The following table 4.1 presents all the results gathered from each model, with a preloaded state.

Model Description	Loss	Precision (%)	Recall (%)	F1-score (%)	AUC
Double GRU	0.020993	0.324618	0.376224	0.348521	0.948328
Double GRU + skip	0.015524	0.543807	0.167832	0.256502	0.851261
Double GRU + extra layer	0.015750	0.511565	0.175291	0.261111	0.857532
Double GRU + skip + extra layer	0.017256	0.383772	0.081585	0.134564	0.843788

Table 4.1: Training and test results of each model. Label *skip* implies a skip connection of the input to concatenating of GRU outputs.

By comparing the various alternative models considered in this work, we can note two main results:

1. Looking at the loss, as the model complexity grows (i.e., moving from the first row to the last row on Table 4.1), the prediction quality does not always increase. In particular, loss is seen to improve for the models with a only a extra layer or skip connection in comparison to the 1st, simpler model. However, loss increases when extra layer and skip connection are present, i.e. the 4th model. In other words, a higher model complexity does not necessarily reflect into better predictive quality, at least if loss is used as quality metric.

2. Looking at AUC and F1-score the above observation is confirmed and indeed the more complex model performs worse compared to the 2nd and 3rd models, highlighted by the f1-score collapsing as a consequence of recall reducing by a large amount in the 4th model.

Unfortunately, due to time constraints, it was not possible to attempt improvements on the 4th model's recall/f1-score with techniques that compensate for the unbalancedness of the dataset, such as over-sampling of the positive class). Since the objective of this dissertation is not to propose new model architectures for this specific domain, but rather to evaluate the impact of shared state management in ML models deployed in a distributed cluster. So, even though the 4th model is not as competitive as the other models, it is still useful for this dissertation as it allows us to study the impact of distribution and synchronization in a broader range of models.

4.2 Inference

Now with the models trained and the state preloaded, they are ready to go into production and be placed in a distributed system for inference. The experiment will be ran multiple times, by running different combinations of values from the variables described below. The variables, and their respective implications to the system are as follows:

1. **Model**: The models being placed in production. We wish to verify if complexity or key differences in the structure of the model can change the overall behavior of the systems. The models used were the ones described by the diagrams in section 3.2;

2. **Merging Strategy**: The strategy for merging shared states when the threshold is reached. The strategies used were the ones presented in chapter 3.3.3;

3. **Number of workers**: With more workers to distribute incoming transactions, the greater is the theoretical lack of synchronization across workers, i.e., the number of concurrent updates of the shared states that will be missed by each worker. of information each worker will receive at any given time. Sets of 2, 4 and 8 workers were used for this experiment;

4. **Synchronization Threshold**: After a given threshold on the number of transaction processed by a worker node without synchronization of transactions processed, write on the database to begin merge operation between workers. The values ranged from 1, meaning at every transaction we merge the state, and increase in power of 2 until reaching 1024 as the maximum threshold.

In the upcoming model evaluation, we will adhere to the same set of evaluation metrics as in our previous assessments. We will calculate their values by averaging the results obtained from each worker, combined with the resulting standard deviation from all workers. This approach will enable us to analyze the distribution of results effectively and will also involve the recording of the time spent on reading and writing to the database, in conjunction with tracking the throughput of transactions throughout the execution process. Due to time constraints it was only possible to run once this study.

4.2.1 Benchmarking Performance

The objective of this work is to study to what extent the proposed synchronization techniques impact the predictive quality and execution performance of ML models deplyed over a cluster of machines. For that one can conclude, in an ideal scenario, that the perfect merge strategy would merge states into a state at any given point in time that is exactly the same as each transaction were executed sequentially. We can already have results of such ideal scenario of performance, when running the model on a single worker. At the same time, the worst merge operation possible would be one equivalent of creating a state with random values, within the range acceptable by the shared state. Such random vectors don't help in model prediction, but rather add noise for the model that is expected to hinder its predictive quality its performance. Based on tehse considerations, it becomes possible to set two experiments to be compared to when running models in production:

- **Lower bound baseline**: The hypothetically limit where the model is expected perform in the worst way possible, recreated by running the model, on the test set, where after each transaction, the shared states were set at random values, distributed between $[-1, 1]$;
- **Upper bound baseline**: The hypothetical limit where the model is expected to perform best. A standard run of the model, on the test set, in a single machine. Each transaction in this case arrives in the right order and the resulting state is updated based on all prior transactions in the input stream, becoming the one best suited to make the prediction.

The following tables 4.2 and 4.3 will report the full set of metrics used to evaluate the models' predictive quality for the lower bound and upper bound baselines:

Model	Loss	Precision (%)	Recall (%)	f1-score (%)	AUC
Double GRU	0.020993	0.324618	0.324618	0.376224	0.948328
Double GRU + skip	0.015524	0.543807	0.543807	0.167832	0.851261
Double GRU + extra layer	0.015750	0.511565	0.511565	0.175291	0.857532
Double GRU + skip + extra layer	0.017256	0.383772	0.081585	0.843788	0.134564

Table 4.2: Results for upper bound performance for each model to be run in production

Model	Loss	Precision (%)	Recall (%)	f1-score (%)	AUC
Double GRU	0.096494	0.069203	0.715618	0.946229	0.126202
Double GRU + skip	0.545953	0.013363	0.540326	0.777476	0.026081
Double GRU + extra layer	0.165568	0.025630	0.423776	0.783792	0.048337
Double GRU + skip + extra layer	0.350612	0.013196	0.398601	0.709936	0.025546

Table 4.3: Results for lower bound performance for each model to be run in production

When comparing the values side by side, one can observe a substantial difference in performance when the state becomes completely randomized. Loss increases up to five times its original value, while recall more than doubles. However, precision approaches values near zero, with thousands of transactions being falsely predicted as positive. These benchmarks are promising as they suggest a strong correlation between the RNN state and prediction outputs. Consequently, it is important to consider the possibility that the proposed shared state merging methods might contribute to model instability, albeit to a much lesser extent.

4.3 Evaluation of the Emulated Distributed Production Environment

4.3.1 No Synchronization Baseline

The first set of results to be extracted is without employing any synchronization to shared states, when distributing transactions. This essential phase will allow one to gauge the inherent limitations by having workers be unaware of given transactions that are essential to perform correct predictions, providing valuable insights into the system's responsiveness and reliability. Given multiple workers involved, the results shown from this point forward will represent the mean of their contributions along with the standard deviation. This approach will assist us in identifying potential outliers that could extrapolate results, as well as understanding variability in each workers predictive quality.

Since this thesis focus is on managing RNN state in distributed environments, it is invaluable to assess the performance implications of system to handle those states. This assessment is achieved through tracking the DNNS's resources in its various aspects: processing transactions, access to the database, communication and many others. For this goal: the following metrics were chosen, averaged among how many workers were instantiated:

- **Throughput**: How many transactions were processed per unit of time
- **Forward Pass time**: Time spent processing transactions
- **Database access time**: Time spent reading and writing in the database

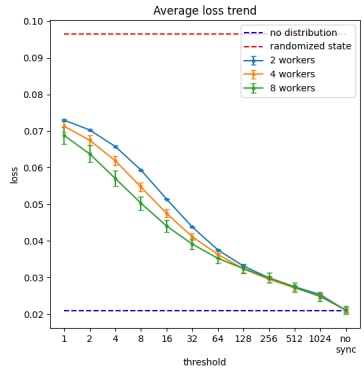
It is recalled that this system was emulated on a single machine using multiprocessing. Because of that, one must consider the possibility of fluctuations in the before mentioned metrics from running the study in a system with no network overhead and shared resources from different workers on the same machine.

The findings seen in table 4.4 reveal near-insignificant variance when the number of workers in the system varies. Additionally, there is almost no variance concerning non-distributed execution. These results can be explained by either a low dependency of state on model outcomes, although this is unlikely, as our lower bound benchmark demonstrated significant variations, or by the model's ability to construct a strong RNN state for making accurate predictions even with missing transactions and limited information. From this insight, it is not impossible to consider the possibility that merging states may yield unexpected results, as the absence of synchronization maintains consistent performance. Merging states may, in fact, be more detrimental than beneficial to the model's performance. Upcoming results will take this hypothesis into account, in the case of the appearance of such results, then for future work it will be imperative as first area of improvement the development of such models to be done in contexts where missing data points present more significant impact in predictive quality.

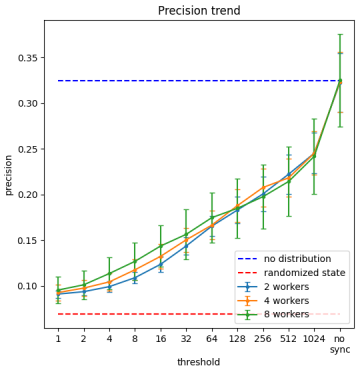
One interesting insight seen as well is in the values of standard deviation at each metric. Regarding loss, there is near zero deviation from the mean values, as for precision, recall and f1-score, the deviation is more noticeable and meaningful, with coefficient of variation reaching up as high as 33%, as seen in precision for the model with skip connection and extra layer, which indicates a larger relative fluctuations around the mean value. That being said it, there is a pattern of increased deviation in relation to the number of workers. Its significance depends on the model observed, as the above mentioned model incurs a large variance model has a lot of variance, whereas a simple Double GRU model presents the opposite scenario.

Model	Nr. Workers	[HTML]000000 Loss	Loss Std. Dev.	Precision (%)	Precision Std. Dev.	Recall (%)	Recall Std. Dev.	f1-score (%)	f1-score Std. Dev.	AUC	AUC Std. Dev.
Double GRU	1 (no sync)	0.020993	-	0.324618	-	0.376224	-	0.348521	-	0.948328	-
	2	0.021021	0.000393	0.322359	0.032282	0.375157	0.045041	0.346759	0.037921	0.947900	0.010629
	4	0.021036	0.000796	0.323043	0.032598	0.378028	0.048060	0.348379	0.038159	0.948003	0.011581
	8	0.021091	0.001051	0.324981	0.050592	0.383694	0.059118	0.351905	0.051132	0.948350	0.014782
Double GRU + extra layer	1 (no sync)	0.015524	-	0.543807	-	0.167832	-	0.256502	-	0.851261	-
	2	0.015492	0.000067	0.534374	0.022469	0.170464	0.045946	0.258475	0.055688	0.853668	0.009079
	4	0.015392	0.001334	0.529004	0.049161	0.176866	0.051312	0.265100	0.065068	0.852758	0.009280
	8	0.015331	0.001513	0.549381	0.084683	0.183796	0.047722	0.275442	0.059028	0.850689	0.021856
Double GRU + skip connection	1 (no sync)	0.015524	-	0.511565	-	0.175291	-	0.261111	-	0.857532	-
	2	0.015762	0.000076	0.511113	0.003421	0.174694	0.003844	0.260389	0.003827	0.857253	0.008143
	4	0.0157670	0.0013344	0.510004	0.021816	0.174759	0.014704	0.260317	0.018775	0.857427	0.008906
	8	0.015770	0.001513	0.510657	0.070237	0.175322	0.025503	0.261027	0.034380	0.857264	0.011693
Double GRU + extra layer + skip connection	1 (no sync)	0.017256	-	0.383772	-	0.081585	-	0.134564	-	0.843788	-
	2	[HTML]000000 0.017265	0.000062	0.384079	0.020798	0.081987	0.001871	0.135129	0.001250	0.844054	0.006097
	4	[HTML]000000 0.017273	0.001261	0.382153	0.041699	0.082126	0.012490	0.135198	0.019098	0.843624	0.006814
	8	[HTML]000000 0.017285	0.001816	0.389644	0.122458	0.081420	0.023019	0.134695	0.036673	0.843319	0.009482

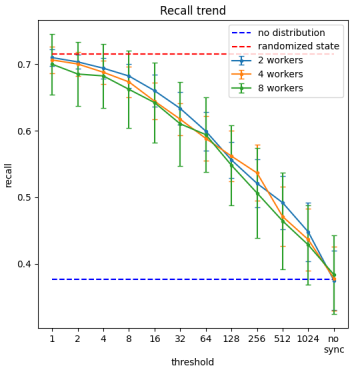
Table 4.4: Mean results from distribution without employing any synchronization techniques, when the number of worker is one, results are equivalent to no distribution



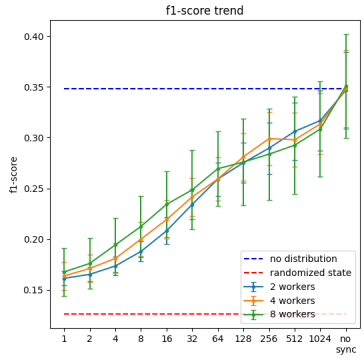
(a) Loss



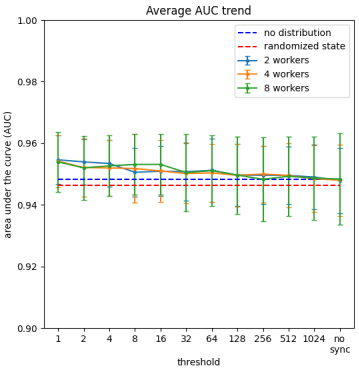
(b) Precision (%)



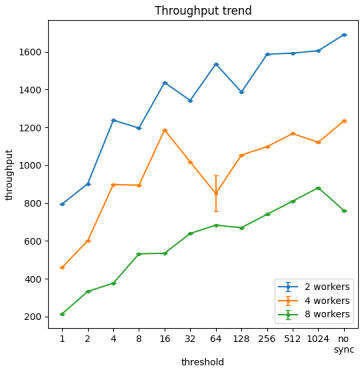
(c) Recall (%)



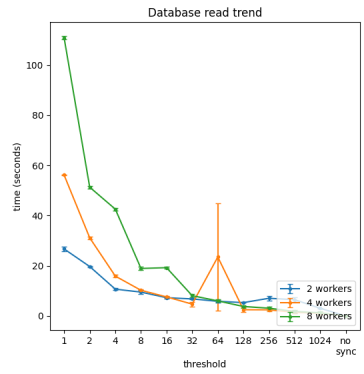
(d) F1-Score (%)



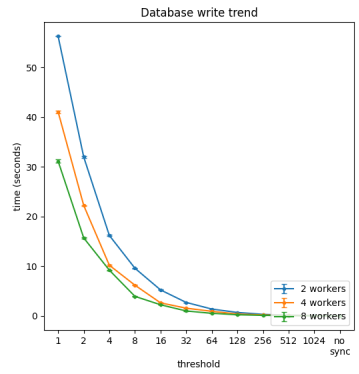
(e) Area Under the Curve (AUC)



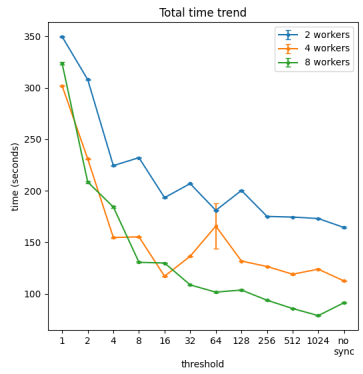
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

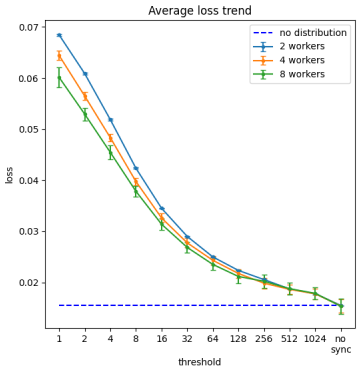


(h) Database write time (seconds)

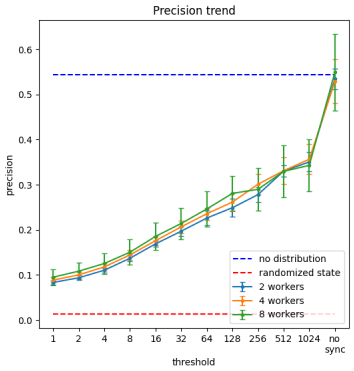


(i) Total execution time (seconds)

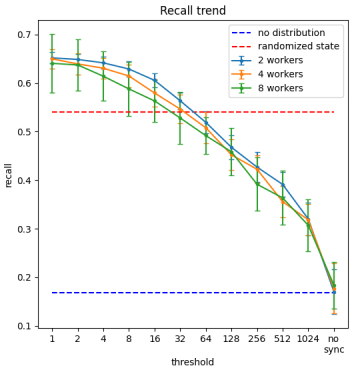
Figure 4.1: Results from distributing Double GRU model, using Synchronous Bound Sum synchronization



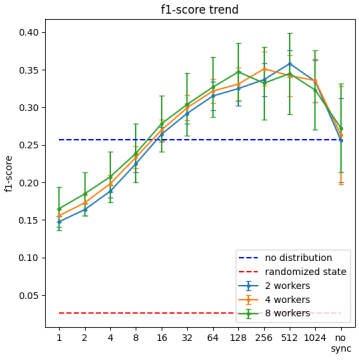
(a) Loss, lower bound not shown due its higher order of magnitude (lower bound = 0.545953)



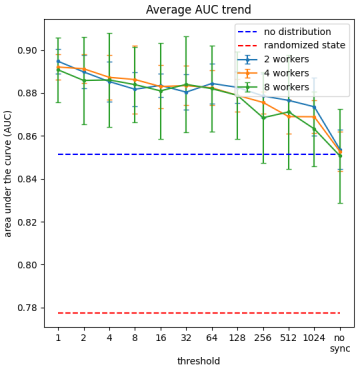
(b) Precision (%)



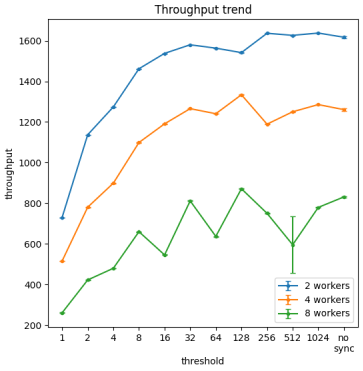
(c) Recall (%)



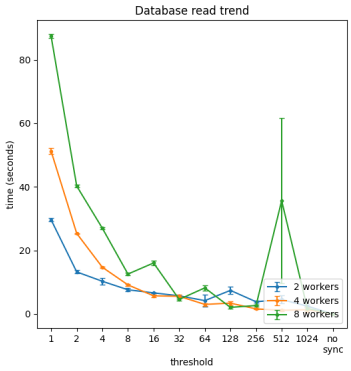
(d) F1-Score (%)



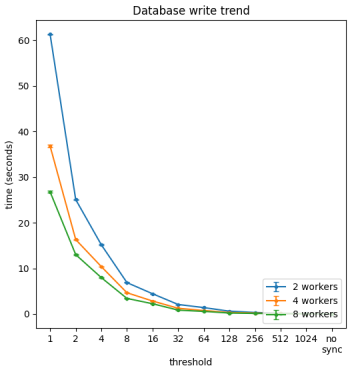
(e) Area Under the Curve (AUC)



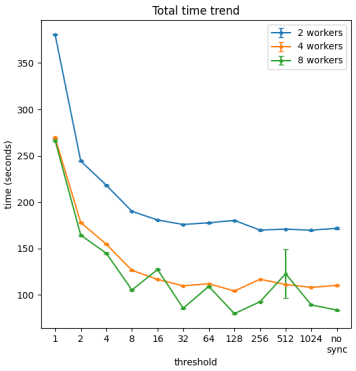
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

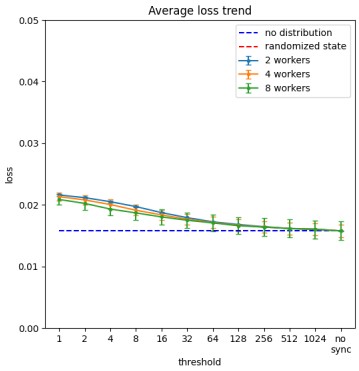


(h) Database write time (seconds)

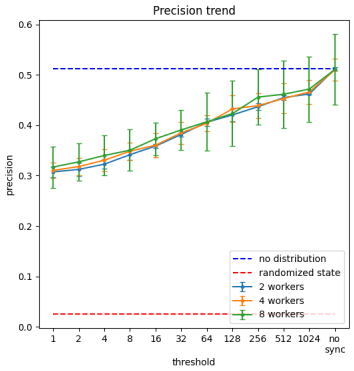


(i) Total execution time (seconds)

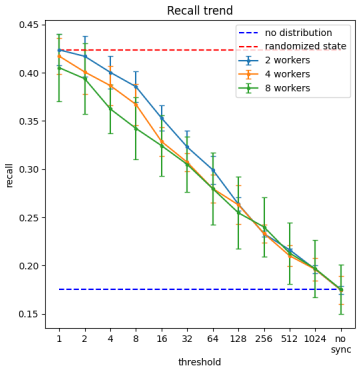
Figure 4.2: Results from distributing Double GRU model with extra layer, using Synchronous Bound Sum synchronization



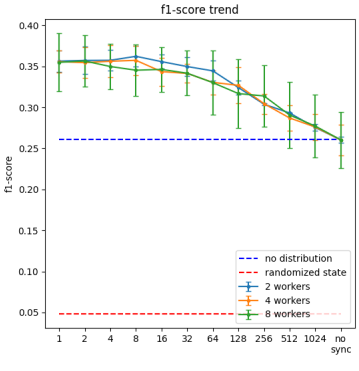
(a) Loss, lower bound was not shown due to its highest order of magnitude (lower bound = 0.165568)



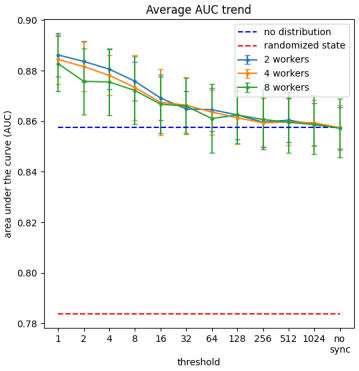
(b) Precision (%)



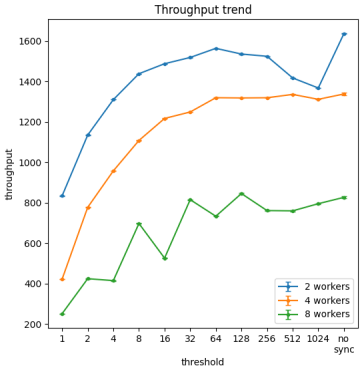
(c) Recall (%)



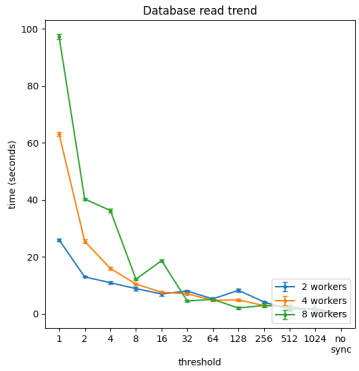
(d) F1-Score (%)



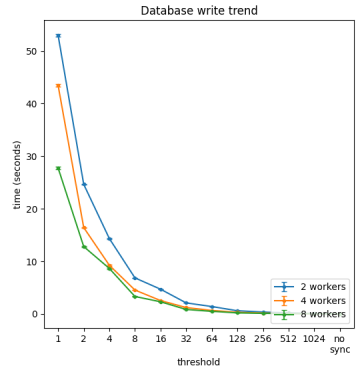
(e) Area Under the Curve (AUC)



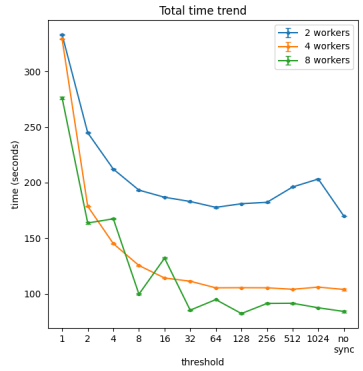
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

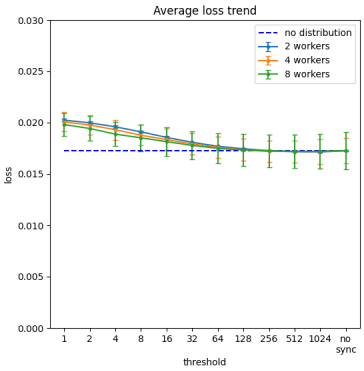


(h) Database write time (seconds)

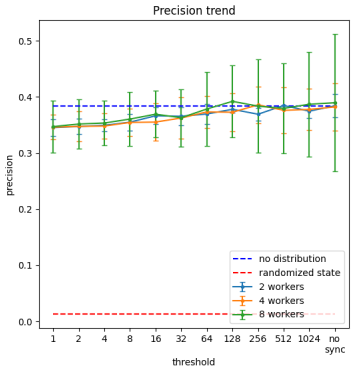


(i) Total execution time (seconds)

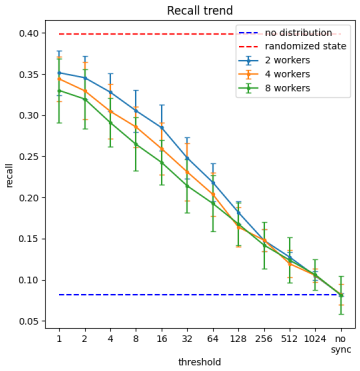
Figure 4.3: Results from distributing Double GRU model with skip connection, using Synchronous Bound Sum synchronization



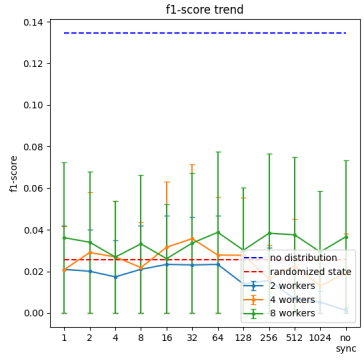
(a) Loss, lower bound not shwon due to its higher order of magnitude (lower bound = 0.350612)



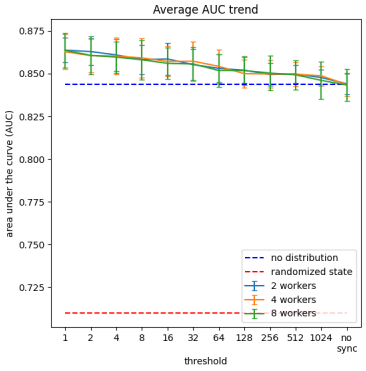
(b) Precision (%)



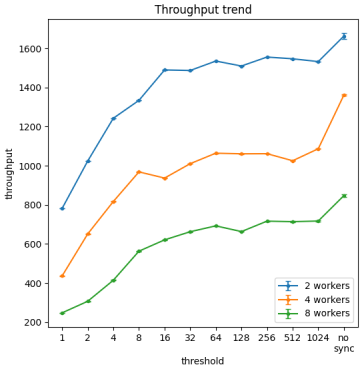
(c) Recall (%)



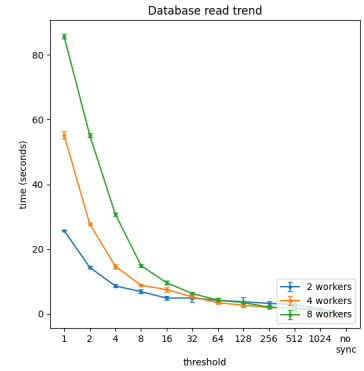
(d) F1-Score (%)



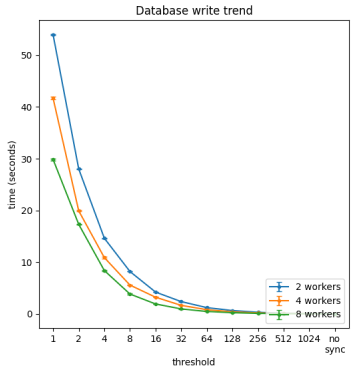
(e) Area Under the Curve (AUC)



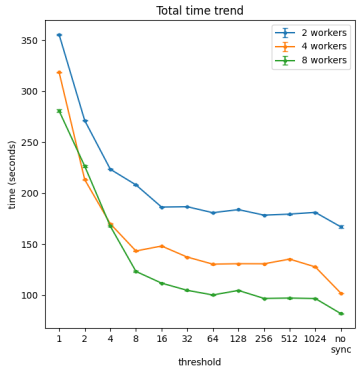
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

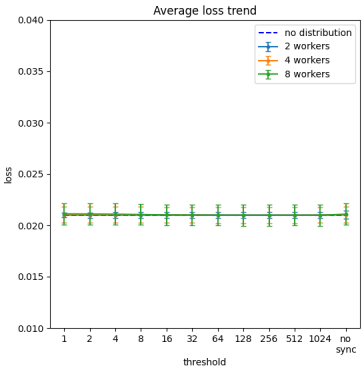


(h) Database write time (seconds)

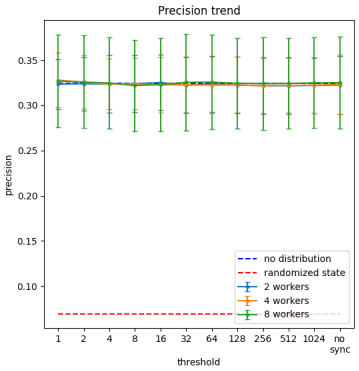


(i) Total execution time (seconds)

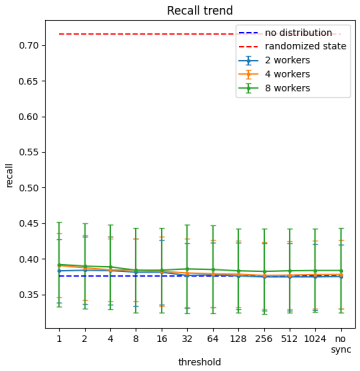
Figure 4.4: Results from distributing Double GRU model with extra layer and skip connection, using Synchronous Bound Sum synchronization



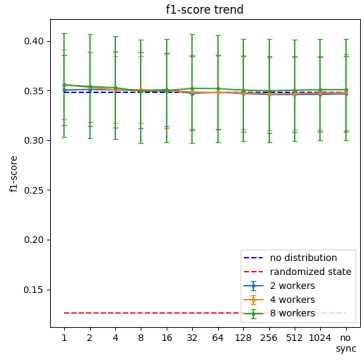
(a) Loss, lower bound not shown due to its higher order of magnitude (lower bound = 0.096494)



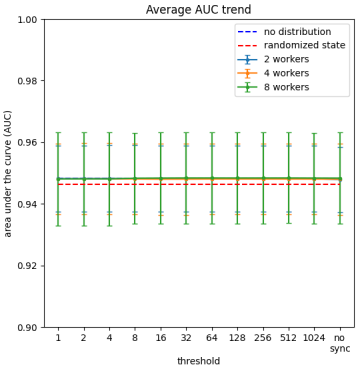
(b) Precision (%)



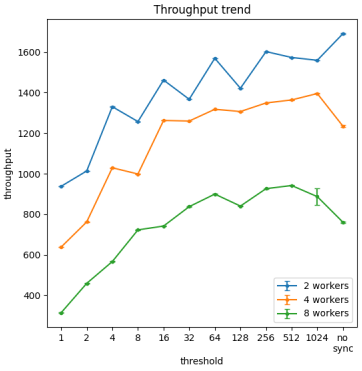
(c) Recall (%)



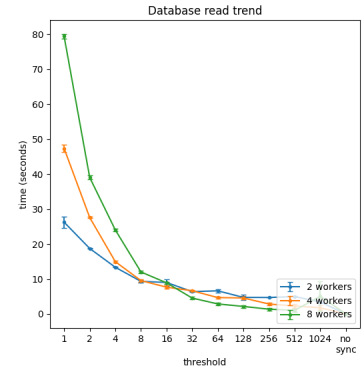
(d) F1-Score (%)



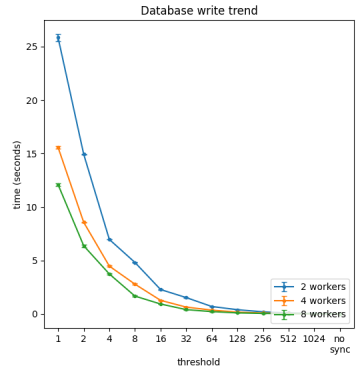
(e) Area Under the Curve (AUC)



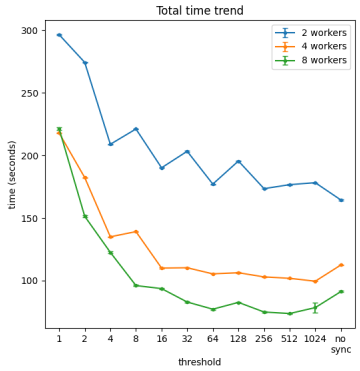
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

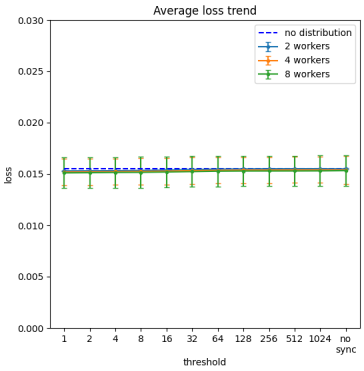


(h) Database write time (seconds)

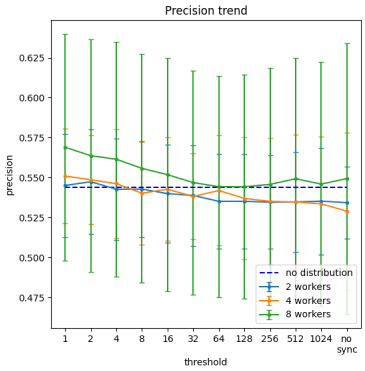


(i) Total execution time (seconds)

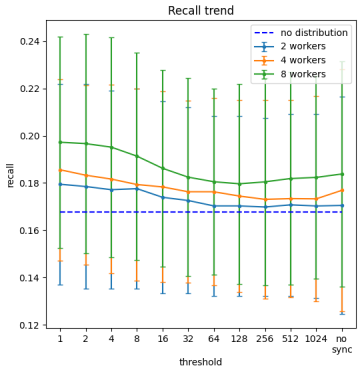
Figure 4.5: Results from distributing Double GRU model, using Synchronous Average synchronization



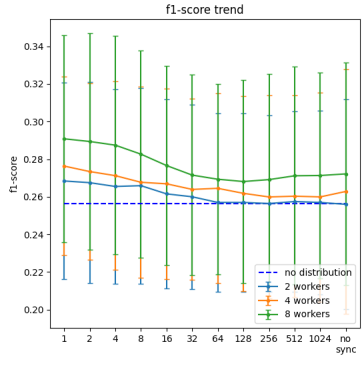
(a) Loss, lower bound not shown due to its higher order of magnitude (lower bound = 0.545953)



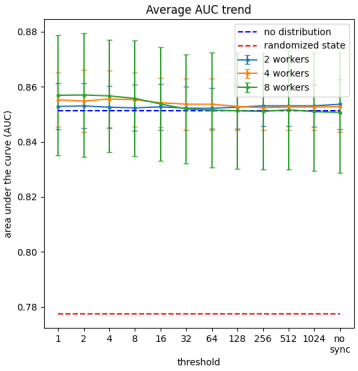
(b) Precision (%), lower bound not shown due to its lower order of magnitude (lower bound = 0.013363)



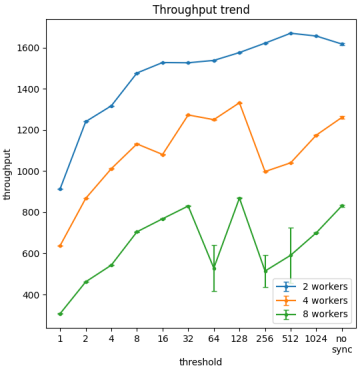
(c) Recall (%), lower bound not shown due to its higher order of magnitude (lower bound = 0.540326)



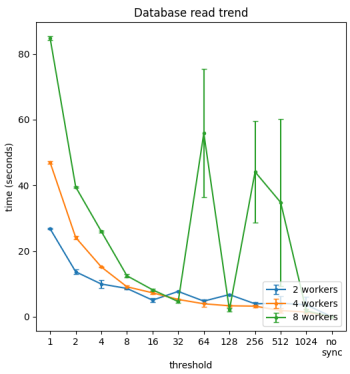
(d) F1-Score (%), lower bound not shown due to its lower order of magnitude (lower bound = 0.026081)



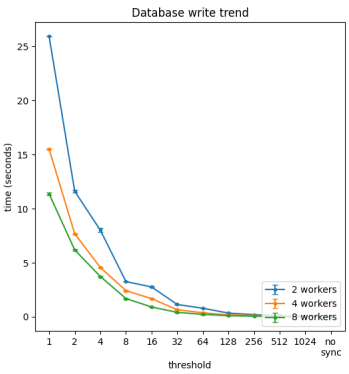
(e) Area Under the Curve (AUC)



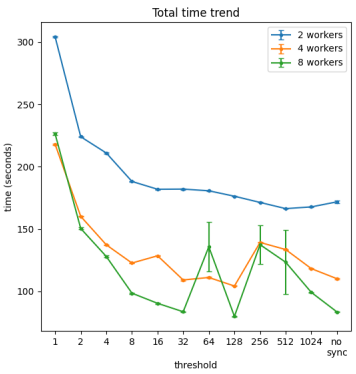
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

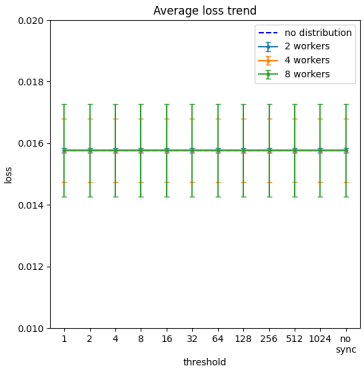


(h) Database write time (seconds)

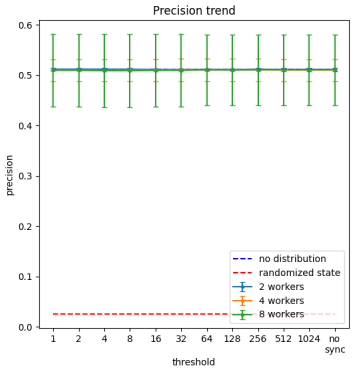


(i) Total execution time (seconds)

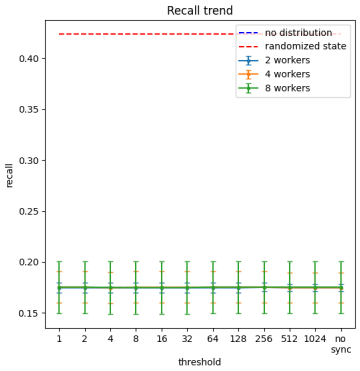
Figure 4.6: Results from distributing Double GRU model with extra layer, using Synchronous Average synchronization



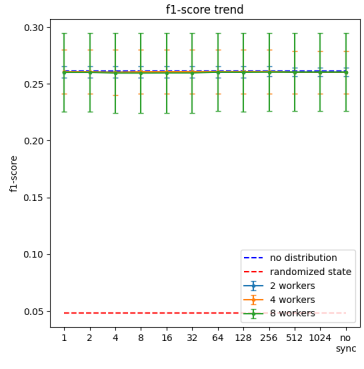
(a) Loss, lower bound not shown due to its higher order of magnitude (lower bound = 0.165568)



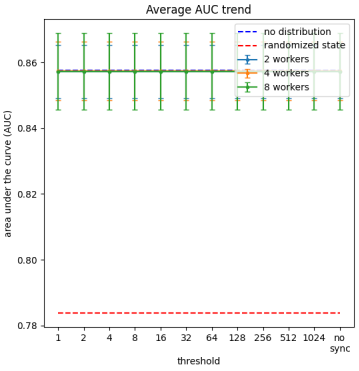
(b) Precision (%)



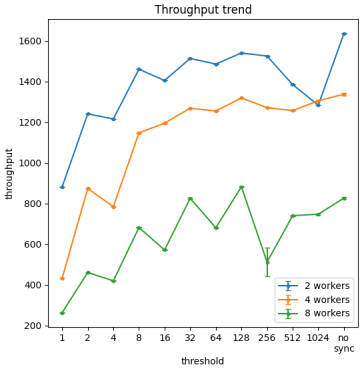
(c) Recall (%)



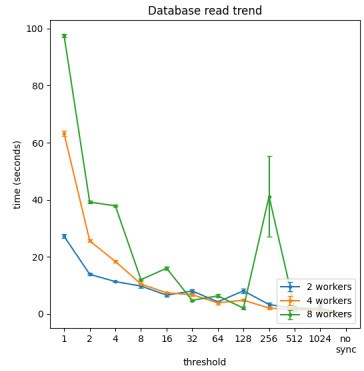
(d) F1-Score (%)



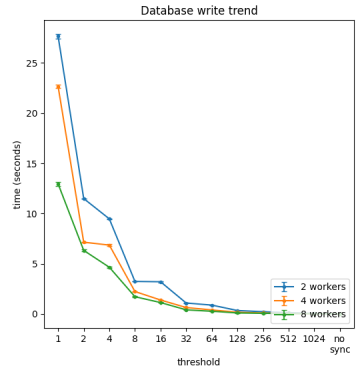
(e) Area Under the Curve (AUC)



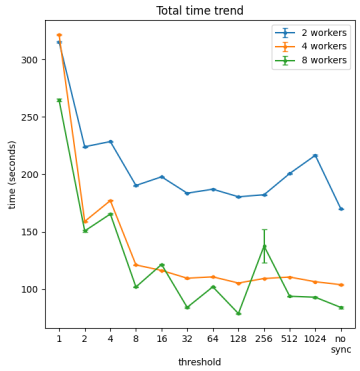
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)

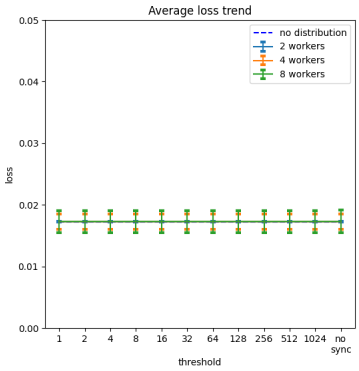


(h) Database write time (seconds)

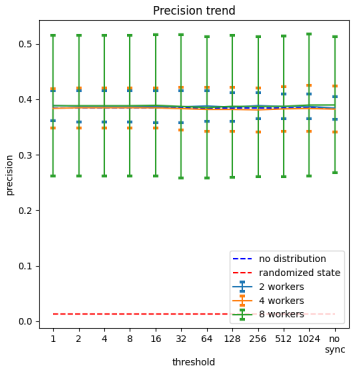


(i) Total execution time (seconds)

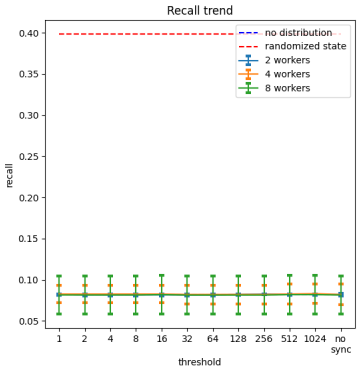
Figure 4.7: Results from distributing Double GRU model with skip connection, using Synchronous Average synchronization



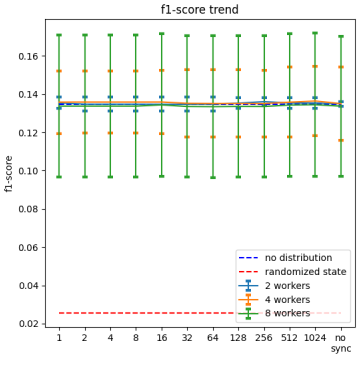
(a) Loss, lower bound not shown due to its higher order of magnitude (lower bound = 0.350612



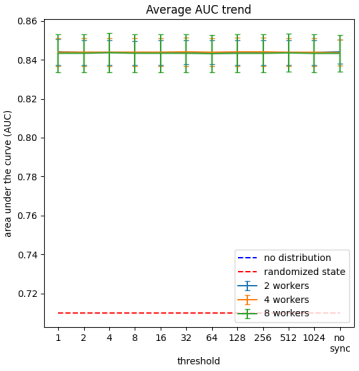
(b) Precision (%)



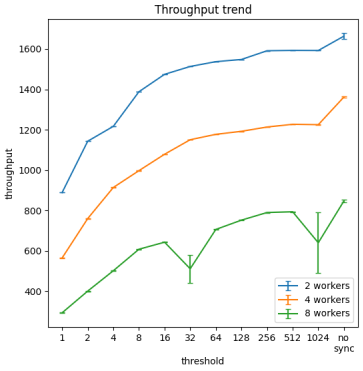
(c) Recall (%)



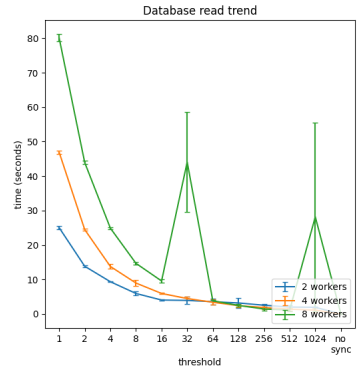
(d) F1-Score (%)



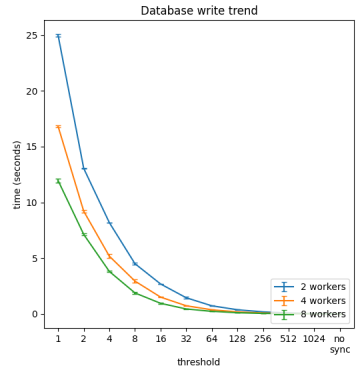
(e) Area Under the Curve (AUC)



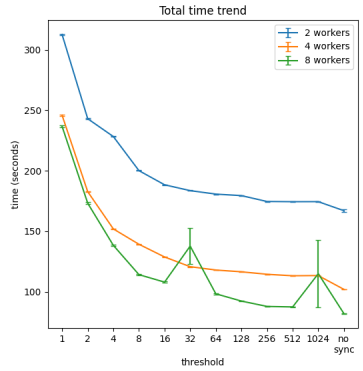
(f) Transaction throughput (transactions per second)



(g) Database read time (seconds)



(h) Database write time (seconds)



(i) Total execution time (seconds)

Figure 4.8: Results from distributing Double GRU model with extra layer and skip connection, using Synchronous Average synchronization

4.4 Impact of alternative synchronization strategies

In this section we will analyze the results obtained from our study in applying synchronization techniques to our emulated distributed production environment. The results obtained correspond to the figures from 4.1 to 4.8. The first 4 figures refer to results from using Bound Sum, whereas the latter 4 are the results from using Average. In each of the 4 figures in each set is the results of the merging function applied to each of the models proposed in this thesis. It was opted to analyze the collected data jointly based on three main factors and their respective impact to the results observed.

1. Synchronization Threshold;
2. Number of Workers;
3. Merging Function

Finally, closing remarks regarding the overall nature of the results obtained, indicate limitations of the current experimental approach, and propose ideas that can be applied and studied for future work to follow.

4.4.1 Impact of synchronization threshold

One can see in all results from this study, that in all models their performance closes in to the results when no synchronization is applied. Not only that, based on table 4.4, these results are very similar to the values when distributing, but without applying any synchronization. Depending on the model being taken into account, for example, applying bound sum on the models with either an extra layer or a skip connection would actually improve overall model performance. Whereas the remaining ones would benefit more from not synchronizing at all. It was theorized synchronizing state, in the scenarios where the model improved in comparison to baseline performance, could be serving as a *regularization factor* for the models, by changing slightly the internal state through bound sum, an thus helping against possible overfitting present in the state. There are some possible means to verify the truthfulness of this theory, one could study and plot the distributions of state before and after synchronizing, compared with the expected distribution normally applied by typical regularization factors.

4.4.2 Impact of number of workers

The impact of the number of workers in overall performance is dependant on which merging function being used. In regards to averaging state, most model's performance did not vary much with the different number of workers in the system, with differences matching at most one hundredth of a percent, with exception to Double GRU with only an extra layer, capping differences between different metrics at most 2%. Now relating to Bound Sum, there is greater difference, albeit this time capping at 5% as seen with the base Double GRU model.

A direct correlation can seen between the degree of standard deviation and the number of workers. As expected, when moving from a single machine to multiple machines, the prediction quality perceived by users can vary depending on which machine is used to serve the user's requests. With that said, it has been recorded for the standard deviation of the different metrics with all models, regardless synchronization strategy, have shown to yield greater deviation when increasing the number of workers, important to not however, that the degree of which the deviation changes is dependant on the model and the merging function in question.

4.4.3 Impact of different merging functions

The results from our study vary greatly in correlation to the merging function used, each showing very different characteristics. In the context of Averaging states, with exception of the 2nd model (Double GRU with extra layer), the results show unaffected performance with increasing thresholds. With mean values nearly identical to the non distributed results. Currently there isn't a proposal that could justify these results, nonetheless, there are possible course of actions to take that can help justify this. For example, plotting the distribution of the shared state before and after synchronization. Simultaneously, standard deviation of the results. Regarding Bound Sum, the results vary greatly depending on the model, further consolidating the idea that each model has learned its own way of generating its shared state and, as a consequence, the effects of adding deltas to its state can become either beneficial (like with the 2nd model and 3rd model) or detrimental (1st model). Therefore, if one had to choose which approach to deploy each of the models in a cluster, one would need to weight the benefits and setbacks each approach yields and make the appropriate decision. With that being said, in a scenario where it was necessary to choose more appropriate merge function for each of the respective modelFs, the decisions would be:

- **Double GRU**: Since its performance decreases overall when using Synchronous Bound Sum, it is trivial to choose Synchronous Average, allowing the model to reap the benefits of parallelizing transactions with no compromise of model performance.
- **Double GRU with extra layer**: With both merging functions, the model's performance increase, however with Average, the model's performance becomes equal to no distribution at higher threshold, wheras with Bound Sum, the model's performance reach peak values of performance for F1-Score in all fields at 512/1024 threshold, with some cost to precision and loss. If in a hypothetical production scenario, where the cost of predicting a false positive input is high enough to force maximizing precision, choosing Average would be the more sensible option. On the other end, if it is possible to leverage precision to gain ability to predict more positives, then choosing Bound Sum would be the best option.
- **Double GRU with skip connection**: This model predictions improve when using Bound Sum at low thresholds, such scenario is useful if we wish to ensure fraudulent transaction are caught. However from a system performance point of view, the choice is not so trivial, as at very low thresholds throughput drops by half. In many production scenarios, performance is a requirement for systems to handle large throughput of incoming transactions. As such, the choice is dependant on the needs of the user.
- **Double GRU with extra layer and skip connection**: Similar as the 1st model, Synchronous Average as a merge function is much more stable in comparison to its Bound Sum counterpart, as a consequence, the most appropriate approach is Synchronous Average.

4.4.4 Final Remarks

Overall, in this study, model quality can vary in ways that are not easy to predict. As possibly each model was trained to generate shared state in different ways. As such, when state is merged and synchronized, the end results can differ a lot. For example, when using bound sum as the merge operation, there are models in whose performance improved when distributed, and others which performed poorly. On the other end, some models using Average merging function have shown indifference in the end results, remaining constant at all thresholds. albeit not in all models. This set of varying results motivate for future work in understanding why such results were obtained in the first place, though careful observation of the

evolution of the shared state as transactions are processed and synchronized, testing in more realistic scenarios and expanding the range of variables like threshold to better capture the landscape of the behavior of such systems.

Chapter 5

Conclusion

This thesis sought to implement a Distributed Neural Network System to study state-of-the-art models, tackling issues that plague development of systems using RNNs and answering questions regarding Feedzai's model proposal, namely:

- Can more profiles be achieved using shared states regarding different aspects of transactions like category?
- How to develop a model for production when the sequence size infinitely large?
- How can shared states be managed in a Distributed System?
- Can such shared states be robust to asynchronous approached?

No literature as ever covered such issues, often because most applications try to work around this issue by isolating state for fixed sequences that are sent to a single machine, removing synchronization altogether. As such this thesis worked on a scenario in which this workaround is not possible, credit card fraud detection, using extremely large sequences whose state can not be computed in a single session.

Despite this, the unpredictable nature of the results seen in chapter 4 show possible indicators of RNN state being very fragile to changes and dependant on order of inputs, such idea is not impossible, as one can look at the field of Natural Language Processing and see the heavy dependence in input order for correct predictions, this being in non–distributed environments. Furthermore, this work doesn't dive in depth on mathematical concepts that might justify the cause behind these results, or even uncover new methods previously unheard of.

In conclusion, this thesis successfully developed new models that expand on the concept of shared states, that more than one type of RNN profiles can be created to capture patterns in sequential data. Consequently, the ML models were deployed over a simulated cluster of machines through multiprocessing, whose internal state was synchronized with proposed synchronization techniques. Finally, the cluster was evaluated to assess the impact it had on model performance.

5.1 Future Work

Nevertheless, there is room for improvement in this work, as well as new ideas to explore this field of study. In the future, some hypothesis could be tested that can be of benefit, for example:

- Using datasets based on real data. Most often than not, datasets made publicly available regarding fraud detection are obtained from two possible sources: generated from simulation algorithms, or obtained from real world data after extensive feature engineering. Enabling the possibility of using real data would make subsequent results stronger and more reliable.
- Hyperparameter tuning to develop models close to the state of the art solutions produced for this dataset.
- A hypothesis for a new approach to this problem is to treat the merging of states as a machine learning challenge. This idea arises from the various results obtained in this thesis. One potential avenue for exploration is that each model, when trained, acquires its unique method for generating its respective state. It might be possible to develop an auxiliary model whose primary task is to learn how to take a set of states as input and produce a single, closely approximate 'true' shared state as output. Such a model could be trained using data generated from shared states when the distributed system runs without synchronization, with the labels representing the corresponding state when executed on a single machine.
- Dive into mathematical properties that generate shared state, with the aim of finding proposals for future methods for merging Shared State.
- Apply experiments in fully distributed environments, where each worker is hosted on a single machine and the key-value store database in a machine separated from all workers.

Bibliography

[1] M. Langer, Z. He, W. Rahayu, and Y. Xue, "Distributed training of deep learning models: A taxonomic perspective," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2802–2818, 2020.

[2] B. Branco, P. Abreu, A. S. Gomes, M. S. Almeida, J. T. Ascensão, and P. Bizarro, "Interleaved sequence rnns for fraud detection," in *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3101–3109, 2020.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[4] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[5] L. A. 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*, vol. 14, pp. 583–598, 2014.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[7] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative study of cnn and rnn for natural language processing," *arXiv preprint arXiv:1702.01923*, 2017.

[8] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[9] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[10] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, pp. 295–306, 2014.

[11] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020.

[12] I. H. Consultants, "The nilson report.," 2019.

[13] J. L. Elman, "Finding structure in time," *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[14] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, no. 4, pp. 681–694, 2020.

[15] openAI, “Chatgpt: Optimizing language models for dialogue,” 2022.

[16] J. W. Lee, “Stock price prediction using reinforcement learning,” in *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No. 01TH8570)*, vol. 1, pp. 690–695, IEEE, 2001.

[17] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[18] S. Burckhardt *et al.*, “Principles of eventual consistency,” *Foundations and Trends® in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, 2014.

[19] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, “More effective distributed ml via a stale synchronous parallel parameter server,” *Advances in neural information processing systems*, vol. 26, 2013.

[20] J. H. Lee, J. Sim, and H. Kim, “Bssync: Processing near memory for machine learning workloads with bounded staleness consistency models,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 241–252, IEEE, 2015.

[21] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, “Automated runtime-aware scheduling for multi-tenant dnn inference on gpu,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 2021.

[22] “Ulb (universit e libre de bruxelles) dataset on big data mining and fraud detection,” 2013.

[23] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[24] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[25] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain.,” *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[26] M. Zinkevich, M. Weimer, L. Li, and A. Smola, “Parallelized stochastic gradient descent,” *Advances in neural information processing systems*, vol. 23, 2010.

[27] W. Yin, K. Kann, M. Yu, and H. Schütze, “Comparative study of cnn and rnn for natural language processing,” *arXiv preprint arXiv:1702.01923*, 2017.

[28] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[29] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 6, no. 02, pp. 107–116, 1998.

[30] S. Kotsiantis, D. Kanellopoulos, P. Pintelas, *et al.*, “Handling imbalanced datasets: A review,” *GESTS international transactions on computer science and engineering*, vol. 30, no. 1, pp. 25–36, 2006.

[31] K. Shenoy, “Simulated credit card transactions dataset generated using sparkov,” 2020.

[32] A. P. Bradley, “The use of the area under the roc curve in the evaluation of machine learning algorithms,” *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[33] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, “Tensorflow-serving: Flexible, high-performance ml serving,” *arXiv preprint arXiv:1712.06139*, 2017.

[34] H. Kim, J. Park, J. Jang, and S. Yoon, “Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility,” *arXiv preprint arXiv:1602.08191*, vol. 3, 2016.

[35] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, “Sparknet: Training deep networks in spark,” *arXiv preprint arXiv:1511.06051*, 2015.

[36] B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent,” *Advances in neural information processing systems*, vol. 24, 2011.

[37] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. Xing, “High-performance distributed ml at scale through parameter server consistency models,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[38] Q. Qin, J. Ren, J. Yu, H. Wang, L. Gao, J. Zheng, Y. Feng, J. Fang, and Z. 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)*, pp. 729–736, 2018.

[39] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression,” *arXiv preprint arXiv:1710.01878*, 2017.

[40] I. C. i. J. L. L. M. P. H. A. Addison Howard, Bernadette Bouchon-Meunier, “Ieee-cis fraud detection,” 2019.