Universidad Autónoma de Madrid

Escuela Politécnica Superior

23|24

Bachelor thesis

Federated Machine Learning based on Variational Inference

Daniel Neo López Martínez

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

Campus Internacional excelencia UAM CSIC+

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor in Computer Engineering

BACHELOR THESIS

Federated Machine Learning based on Variational Inference

Author: Daniel Neo López Martínez
Advisor: Daniel Hernández Lobato

June 2024

Daniel Neo López Martínez
Federated Machine Learning based on Variational Inference

Daniel Neo López Martínez

PRINTED IN SPAIN

To my family, Carlos Canoyra and Carlos Cabezas for keeping me company all along this journey

Modern man thinks he loses something—time—when he does not do things quickly.
Yet he does not know what to do with the time he gains—except kill it.

Erich Fromm

Akcnowledgements

I would like to specially thank my tutor Daniel Hernández Lobato, who has not only taught me the basics of statistical learning, he has also been a great person all around. This year being the last year of my bachelor, it has been complicated with internships, applications and exams, but he has been very understanding, helpful and accommodating. He has never had any problems helping me and has always answered promptly at my emails even if they were sent at the most untimely hours. I could not have wished for a better tutor. For all of that, I thank him.

I would also like to thank the staff at the Escuela Politécnica Superior for all these years of great learning and I hope they keep teaching the future generations with the same enthusiasm and passion they have showed me in my stay here.

From this staff, I would like to specially thank Alejandro Bellogín. He has been a great teacher and tutor for my internship. He has always been available for anything that was needed. He has always showed great interest and showed that he cared, even taking time out of his personal time to assist me multiple times. He has been an exceptional teacher, but an even better person. For all of that I thank him, and I hope that he continues to provide this positive influence to the following generations of students and, in case I follow the academic path, maybe even see each other again in a more professional setting.

I express my gratitude to my family, for always supporting me. They have provided me with a loving and nurturing environment that has made me the person I am today.

And to my friends, who have been there in the good and bad moments. Specially my group of friends from the San Mateo institute and my small group of friends from the Valdeluz institute.

Abstract

Machine learning consitutes an essential part of many fields of study, making it possible to identify, classify and generate data both practically and accurately. However, the computational and functional requirements of the models have greatly increased as the amount and complexity of the data to be processed have augmented. In order to face these challenges, different kinds of training have appeared.

Bayesian Learning offers precise ways to measure the uncertainty of a model, allowing the use of Machine Learning in fields where the cost of misclassification is great.

On the other hand, Federated Learning can train a model in a distributed environment without the need of sharing the data. This allows for the the high computational load of modern models to be distributed across the different users while also keeping the privacy of the training data.

Both types of training could work well together. However, they are not compatible in their basic form due to the inherent incompatibility of privacy and the need of total access to the data. Partitioned Variational Inference was created to unite both types of training.

This thesis consists in the development of a framework for training models with Federated Learning based on Variational Inference, using Partitioned Variational Inference. Two libraries have been created using the Rust programming language. One general library that includes the full functionality for Rust, and another one for Python that just includes the client's functionality.

In the context of developing the framework, a communication protocol for Partitioned Variational Inference has been created. This protocol has been created with the objective of formalising the communication process and allowing for the creation of new independent client or server implementations.

Various experiments have been done using the framework to test its efficiency and proper functioning. These experiments consisted on the training of two different models using two different datasets, one generated artificially and one real, using different configurations.

Keywords

Machine Learning, Federated Learning, Distributed computing, Variational Inference, Rust

Resumen

El aprendizaje automático forma una parte esencial de muchas áreas, permitiendo realizar tareas de identificación, clasificación y generación de manera práctica y precisa. Sin embargo, las necesidades tanto computacionales como funcionales que se requieren de los modelos han aumentado en gran medida según han ido aumentando la cantidad y complejidad de los datos a procesar. Para afrontar estos desafíos han aparecido distintos tipos de entrenamiento.

El aprendizaje Bayesiano provee mecanismos para medir de manera precisa la incertidumbre de un modelo, permitiendo el uso de aprendizaje automático en campos donde la certidumbre es esencial.

Por otra parte, el aprendizaje Federado provee mecanismos para poder entrenar en un entorno distribuido sin la necesidad de compartir los datos. Esto permite la distribución de la carga computacional elevada de los modelos modernos y la conservación de la privacidad de los datos.

Ambos tipos de aprendizaje podrían funcionar muy bien en conjunto, sin embargo, no son compatibles en su forma más básica debido a la incompatibilidad entre la privacidad y la necesidad de acceso total a los datos. La Inferencia Variacional Particionada es un esquema de entrenamiento que surge para compatibilizar ambos tipos de entrenamiento.

Este trabajo consiste en el desarrollo de una librería para la realización de aprendizaje federado usando inferencia variacional basándome en el esquema de Inferencia Variacional Particionada. Se han creado dos librerías desarrolladas en el lenguaje de programación Rust. Una general para Rust que contiene toda la funcionalidad, y otra para Python que solo incluye la del cliente.

En el contexto del desarrollo del proyecto adicionalmente se ha creado un protocolo de comunicación para la Inferencia Variacional Particionada con el fin de formalizar el proceso y permitir la creación de nuevos clientes y servidores que puedan intercomunicarse sin modificaciones.

Se han realizado varios experimentos entrenando modelos utilizando las librerías con el fin de probar la eficacia y correcto funcionamiento de la implementación. Estos experimentos consisten en distintas configuraciones de entrenamiento con dos conjuntos de datos distintos, uno generado artificialmente y otro real, para dos modelos distintos.

Palabras clave

Aprendizaje automático, Aprendizaje federado, Computación distribuida, Inferencia variacional, Rust

Contents

Lists

List of equations

List of Figures

1 Introduction

1.1 Machine Learning and Federated Learning

Machine learning has been the focus of a great amount of research in the last years. Informally speaking, machine learning could be described as the field that studies algorithms whose objective is to, starting from some given data, become able to learn and generalise some identification process or task.

Machine learning encompasses a great number of fields, from pattern recognition in images, to DNA analysis or natural language processing. Machine learning constitutes a very useful tool for these kinds of fields where lots of data processing and pattern generalisation are needed. Current machine learning algorithms are optimised for these kinds of workloads, with fast calculation speeds and great performance processing high amounts of data. As a consequence of this wide area of applications, there has been a substantial development of the field and the term now groups very different types of algorithms.

Bayesian Learning is part of these algorithms. Bayesian learning is a type of machine learning based on the use of Bayes' rule, equation (1.1), and bayesian inference to perform these kinds of tasks.

Given a model, the posterior distribution represents the probability of the model's parameters based on the data observed under the proposed prior distribution. The marginal likelihood acts as a normalising factor so that the posterior is a proper probability distribution. This is further explained in Chapter 2.

$$posterior = \frac{likelihood \cdot prior}{marginal\ likelihood} \tag{1.1}$$

In Bayesian learning, the objective is to obtain a predictive distribution that accurately represents the distribution of the data that we want to predict. Modelling the problem this way offers up some benefits that are quite desireable, like the posibility of reasoning about the nature of the data and its distribution, reasoning about the uncertainty of the model, or improved capabilities when training with low amounts of data.

However, this kind of training entails new kinds of problems, specially regarding the calculation of the posterior distribution. One of the main ones is that the calculation of the true posterior usually requires calculating intractable integrals. The two main current solutions to this problem are, either approximating the integrals by numerical means (like the Markov Chain Monte Carlo methods), or transforming the integral calculation problem into an optimisation one (like the Variational Inference methods, which I will be working with).

Machine learning has opened lots of doors that were previously closed. For example, right now with the rise of different kinds of Neural Networks and Large Language Models, the fields of image generation, audio generation and chatbots have improved their capabilities and quality to levels never seen before. However, this increase in quality has come hand in hand with an increase in performance requirements not only in the training phases but also in the inference phases (inference referring to the process of obtaining an output from an input in a trained model), mainly due to the increase in the number of parameters that these models hold.

As a consequence of this, for some of these modern models normal computers no longer suffice. Some of the solutions to this problem being used right now encompass: the reduction of performance requirements by the use of techniques like quantization of models, the use of computers with better specs or the distribution of the models' workload. This last one is the kind of solution I will be focusing on.

The amount of data being generated increases by the day and, accordingly, the interest placed on using these data for the previously mentioned purposes also increases. However, if we want to learn in a distributed manner, an important problem surges when these data cannot be shared publicly. There are lots of useful data that, for good reasons, are private, like healthcare related data or military information. Usually in a distributed learning environment, these data would be exposed in some way or another when used to train a shared model and, as such, would not be able to be used.

Federated learning tries to mitigate these kind of problems by ensuring that data never leaves its owner. This opens up the possibility of actors with confidential information being able to work towards a shared goal without exposing themselves.

However, silver bullets do not exist, and if we want to learn a shared model without being able to manipulate the data, we will have to make some concessions or significantly complexify the task as we will explore later.

Additionally, this kind of data usually is either scarce or related to high impact fields where failures have great consequences. This is incompatible with the current approach of Deep Learning Neural Networks that provide us with an approximator that usually requires a sizeable amount of data to become precise, and does not usually provide a measure of uncertainty.

Right now there are some frameworks out there that either implement Federated Learning or Varia-

tional Inference learning, but there is no framework that unifies both of these concepts together. Taking all of this into account, I decided to create such a framework to make use of the aforementioned benefits.

1.2 Technical Aspects of Federated Learning and Variational Inference

The main requisites for a Federated Learning framework would be efficiency and heterogeneity. Efficiency is required in almost every form of training due to the computationally expensive nature of the process, but additionally, the framework should also support clients with low resources. Heterogeneity is required due to the possibility of very different clients, as each client is independent from the others.

With this in mind, I chose Rust as the programming language for this project. Rust is a compiled programming language with speeds comparable to C/C++ and no garbage collector. I chose Rust over C/C++ due to its rich type system and ownership model that together guarantee memory and thread safety. I also find Rust to have a better user experience with useful error messages and great tooling.

Regarding heterogeneity, Rust can be compiled to almost every platform, has a great embedded ecosystem and can even be compiled to WebAssembly, but I really wanted to be compatible, so I also designed a protocol for the learning process so that in the future all the possible clients and servers can communicate through this protocol.

However, Rust still has some limitations regarding Probabilistic Learning using Variational Inference, that's why I decided to also offer the client library to Python, which has great scientific tooling, so that a user can get the best of both worlds.

The protocol has been designed with simplicity and optimisation of communication in mind. The clients could have low resources and there will be lots of messages sent between any client and the server. Additionally, there will be potentially lots of clients, so in order to avoid overloading the network or the server, the number and size of the required messages has been reduced to a minimum.

Due to the nature and objectives of Federated Learning, all communications are secure and every client will have to be properly identified to avoid harmful actors.

The project follows the process and workflows detailed in [1], where a process for Federated Learning based on Variational Inference is shown, and adapts them to the previous constraints.

In the end, the objective of this project is to develop an efficient, multiplatform, extensible and easy to use framework that also ensures the safety and confidentiality of the data.

1.3 Chapters' Guide

A brief description of each of the remaining chapters of the document goes as follows:

- 2. Bayesian Machine Learning and Approximate Inference: Introduction to Bayesian Machine Learning in more detail. The main formulas and variational methods used in later chapters are introduced here.
- 3. Partitioned Variational Inference: This is the largest chapter and it contains an explication of the algorithm used to unite Variational Inference and Federated Learning (Partitioned Variational Inference), the design of the system and protocol used to implement it, and an in depth explication of the implementation's details.
- 4. Experiments: The conditions, metrics and details of the experiments used to test the framework. There are two problems where different conditions are tested: a simple problem with the calculation of a mean, and a Bayesian Regression problem.
- 5. Conclusions and Future Work: the results obtained are briefly reasoned about, the future of the ecosystem is discussed and various possible improvements to the framework are explored.

2

Bayesian Machine Learning and Approximate Inference

2.1 Introduction

The extended reason as to why I have chosen Bayesian learning for this task is that I find it a very good fit for the kind of data whose distribution is restricted. These kind of data include as previously mentioned: healthcare related data, military data, personal vehicles geolocation data, etc. There are two main common factors in all of these different types of data aside from the fact that their distribution is restricted, they are either related to an environment where cost of failure is great, scarce or both.

Given Bayes' theorem, where $\boldsymbol{\theta}$ represents the vector of parameters of our model and $\boldsymbol{D}$ represents the dataset, Bayesian learning allows us to address both of these problems by computing a posterior distribution for $\boldsymbol{\theta}$ given $\boldsymbol{D}$:

$$P(\boldsymbol{\theta}|\boldsymbol{D}) = \frac{P(\boldsymbol{D}|\boldsymbol{\theta})P(\boldsymbol{\theta})}{P(\boldsymbol{D})} = \frac{P(\boldsymbol{D}|\boldsymbol{\theta})P(\boldsymbol{\theta})}{\int_{\boldsymbol{\theta}} P(\boldsymbol{D}, \boldsymbol{\theta})\, d\boldsymbol{\theta}} \tag{2.1}$$

Firstly, we can reason about the nature of the results obtained and the uncertainty of the model. In a regular Neural Network, the final result would be a set of weights which given an input, output a value, but usually hold no additional meaning. In a Bayesian model, the output is a distribution that captures the potential values for the model parameters given the observed data, this alone provides a great amount of additional context to the result.

For example, if we obtain a Gaussian with mean μ and standard deviation σ, we can reason about the nature of our data and how it is distributed according to these parameters. This also includes having access to the uncertainty of the model, *the model is capable of knowing what it does not know*. With this information, some tasks such as outlier detection and active learning become much easier.

This last point is essential in safety critical systems where a failure could mean the difference between life and death like healthcare analysis or driving software. It is better for the model to answer to an input, *I am not confident enough to classify this data*, than to confidently give the wrong answer.

To showcase this, let's compare two models trained to drive a car, one trained with a standard Neural

Network and another trained with a Bayesian approach. Let's assume that in the training data there was no data related to blue vehicles. If in the middle of driving the models are faced with a blue truck taking a turn, the first one may associate it with the closest knowledge of big blue things it has encountered like the sky and keep going, while the second one, even if it does not and cannot know what the object is, it knows that it has not seen something like it before and notifies the driver. In both cases the model is unable to solve the problem by itself, but in the second case the model alerts about the problem instead of silently failing. A very similar situation could take place for example when diagnosing cancer.

Secondly, in a Bayesian context, previous knowledge can be included in the training introducing it as the prior of Bayes' rule. This also allows these kinds of models to perform positively with small amounts of data if the prior is chosen correctly, as that bump of previous knowledge can skew the algorithm in the correct way.

However, the choice of prior can be a double edged sword. A good prior can heavily skew the training in the right way, but the opposite is also true, a bad prior can heavily hinder the progress of the training. There are some heuristic and methods to choose the prior, like choosing a weakly informative prior so that it does not overpower the likelihood data, or using some empirical methods for hyperparameter optimization (in this case, the parameters in the prior). The prior is also commonly chosen based on the rest of the model so that the calculations end up simpler. However all of these methods just help choosing a prior, they do not provide an absolute solution.

After making the choice of using Bayesian methods for distributed learning, we still have to choose how we will calculate the posterior distribution given that for the vast majority of useful distributions, the integrals required to calculate $P(\boldsymbol{D})$ will be intractable.

As previously stated, we have two main ways of dealing with this problem, by either solving the integrals numerically with Markov Chain Monte Carlo methods, or by transforming the integral calculation problem into an optimisation one with Variational Inference methods.

2.2 Approximate Inference with Monte Carlo Methods

Monte Carlo methods rely on sampling to solve the aforementioned integral and the predictive posterior.

$$P(\boldsymbol{D}) = \int_{\boldsymbol{\theta}} P(\boldsymbol{D}, \boldsymbol{\theta})\, d\boldsymbol{\theta} = \mathbb{E}_{P(\boldsymbol{\theta})}[P(\boldsymbol{D}|\boldsymbol{\theta})] \approx \frac{1}{N}\sum_{i=1}^{N} P(\boldsymbol{D}, \boldsymbol{\theta_i})\ where\ \boldsymbol{\theta_i} \sim P(\boldsymbol{\theta}) \tag{2.2a}$$

$$P(\hat{y}(x)|\boldsymbol{D}) = \mathbb{E}_{P(\boldsymbol{\theta}|\boldsymbol{D})}[P(\hat{y}(x)|\boldsymbol{\theta})] \approx \frac{1}{N}\sum_{i=1}^{N} P(\hat{y}(x)|\boldsymbol{\theta_i})\ where\ \boldsymbol{\theta_i} \sim P(\boldsymbol{\theta}|\boldsymbol{D}) \tag{2.2b}$$

Both of these Monte Carlo estimates are guaranteed to converge due to the Law of large numbers,

but as the dimensionality of $\boldsymbol{\theta}$ increases, the necessary N for a good estimate rises rapidly. Markov Chain Monte Carlo exchanges sample independence in lieu of better performance and faster convergence. This is done by extracting the samples used to approximate the distributions from a Markov Chain, which produces a sequence of dependent samples. This converges faster as this sequence favours regions with higher probability mass.

However, as stated before, this framework is meant to be used in an heterogeneous environment, probably conformed by a significant amount of low end devices, and this method is computationally expensive even if it is guaranteed to converge to the true posterior.

2.3 Approximate Inference with Variational Methods

That is where Variational Inference comes in. Instead of trying to find the true posterior, this technique seeks to find another distribution that is *close*. This notion of closeness is defined by the Kullback-Leibler divergence between the two distributions, the true posterior and our approximate posterior.

$$D_{KL}(P \,||\, Q) = \int_{-\infty}^{\infty} P(x) \log\left(\frac{P(x)}{Q(x)}\right) dx \tag{2.3}$$

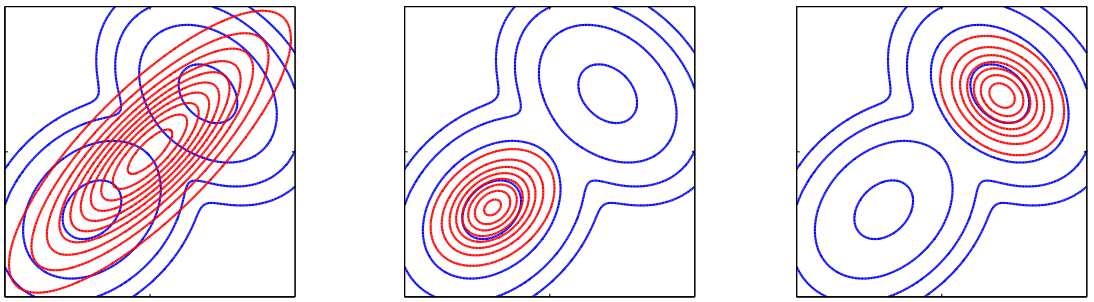
The KL-divergence as a measure is non-negative (as it is a divergence), non-symmetric ($D_{KL}(P \,||\, Q) \neq D_{KL}(Q \,||\, P)$) and its value is zero for two perfectly matching or greater than zero for two non matching distributions.

The objective of Variational Inference is to find the distribution $q^*(\boldsymbol{\theta})$ from a parametric family of distributions Q that minimises the KL divergence with respect to the target distribution $P(\boldsymbol{\theta}|\boldsymbol{D})$, either the forward, equation (2.4a), or reverse, equation (2.4b), kind. The difference between both is that, on the forward kind, the divergence rises if $q(\boldsymbol{\theta})$ does not cover the areas where $P(\boldsymbol{\theta}|\boldsymbol{D})$ is not zero, and on the reverse kind, the divergence rises if there are areas where $q(\boldsymbol{\theta})$ is greater than 0 and it is different from $P(\boldsymbol{\theta}|\boldsymbol{D})$. In practice, this means that while maximising the forward KL divergence tries to cover all non-zero areas of $P(\boldsymbol{\theta}|\boldsymbol{D})$, maximising the reverse KL divergence tries to approximate accurately some areas of $P(\boldsymbol{\theta}|\boldsymbol{D})$ while avoiding inaccurate approximations. Figure (2.1) illustrates the differences between both using a mixture of two Gaussians as an example.

$$q^*(\boldsymbol{\theta}) = \underset{q(\boldsymbol{\theta})\in Q}{argmin}\, D_{KL}(P(\boldsymbol{\theta}|\boldsymbol{D}) \,||\, q(\boldsymbol{\theta})) \tag{2.4a}$$

$$q^*(\boldsymbol{\theta}) = \underset{q(\boldsymbol{\theta})\in Q}{argmin}\, D_{KL}(q(\boldsymbol{\theta}) \,||\, P(\boldsymbol{\theta}|\boldsymbol{D})) \tag{2.4b}$$

However, we cannot calculate either KL Divergence directly as we would need $P(\boldsymbol{\theta}|\boldsymbol{D})$, which is the posterior distribution we are trying to obtain in the first place.



(a) Forward KL divergence $D_{KL}(P \,||\, Q)$ minimum

(b) One local minimum for reverse KL divergence $D_{KL}(Q \,||\, P)$

(c) Another local minimum for reverse KL divergence $D_{KL}(Q \,||\, P)$

Figure 2.1: Comparison of forward Kullback-Leibler divergence and reverse Kullback-Leibler divergence. The blue contours correspond to the objective distribution p given by a mixture of two Gaussians. The red contours correspond to the distribution q found by minimising the corresponding KL divergence. Extracted from [2].

However, we can manipulate the formula for $D_{KL}(q(\boldsymbol{\theta}) \,||\, P(\boldsymbol{\theta}|\boldsymbol{D}))$ (not $D_{KL}(P(\boldsymbol{\theta}|\boldsymbol{D}) \,||\, q(\boldsymbol{\theta}))$) and, even if it is not possible to calculate the divergence directly, we can minimise it by maximising a lower bound $L(q)$, following the deconstruction in equation (2.5a) and equation (2.5b).

The $D_{KL}(q(\boldsymbol{\theta}) \,||\, P(\boldsymbol{\theta}|\boldsymbol{D}))$ is chosen instead of the $D_{KL}(P(\boldsymbol{\theta}|\boldsymbol{D}) \,||\, q(\boldsymbol{\theta}))$ because minimising it is possible as it does not require to calculate $P(\boldsymbol{\theta})$, just $P(\boldsymbol{D}, \boldsymbol{\theta})$, which we **can** calculate by virtue of having access to both the prior and the data.

Let's decompose $P(\boldsymbol{D})$, taking q as the surrogate posterior.

where $L(q)$ is

$$\log(P(\boldsymbol{D})) = L(q) + D_{KL}(q \,||\, P) \tag{2.5a}$$

$$L(q) = \int_{\boldsymbol{\theta}} q(\boldsymbol{\theta}) \log\left(\frac{P(\boldsymbol{D}, \boldsymbol{\theta})}{q(\boldsymbol{\theta})}\right) d\boldsymbol{\theta} \tag{2.5b}$$

It can be seen in equation (2.5a), that given that $\log(P(\boldsymbol{D}))$ is constant, as it depends on the data only, and negative (it is the logarithm of a probability, a number between 0 and 1), and that the KL divergence is always greater or equal than 0, $L(q)$ necessarily is a negative value.

Given that the sum of $L(q)$ and the KL divergence is always constant, as $L(q)$ increases, the KL divergence decreases and, even if we cannot calculate the divergence directly, we can reduce it by solving a maximisation problem on $L(q)$, also known as the ELBO (Evidence Lower Bound). Figure (2.2) illustrates this phenomenon.

There are multiple ways to solve this optimisation problem for the algorithm being implemented. However, the method is not so important as long as we are performing Variational Inference.

For a more detailed explanation of Variational Inference I recommend checking out [3] and [4].

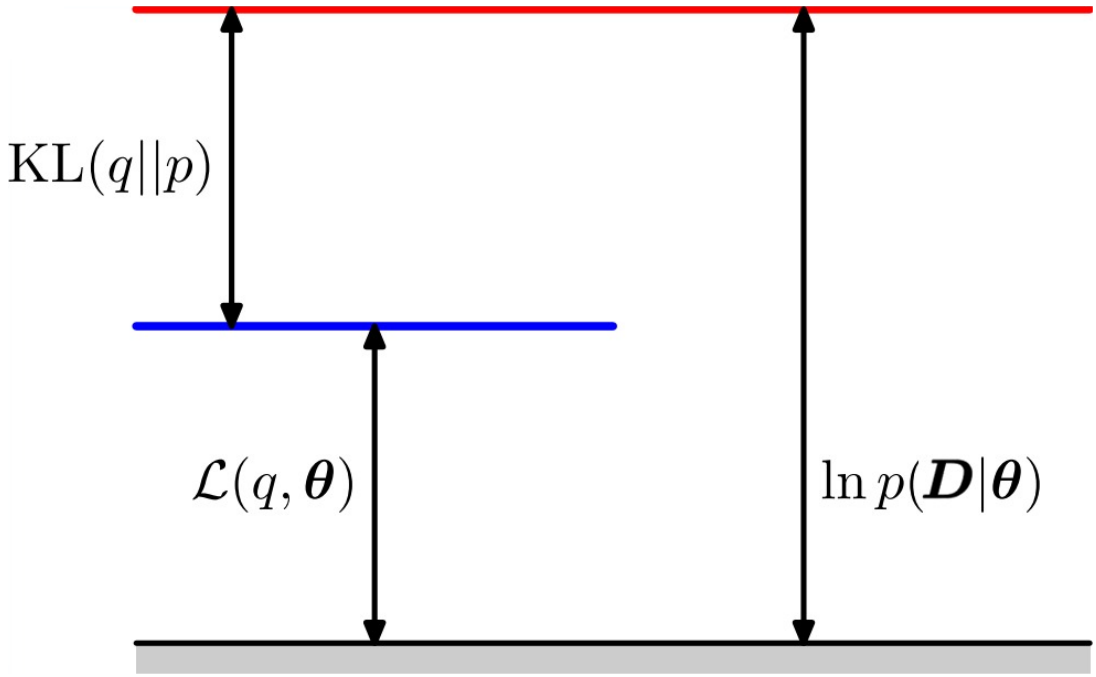


Figure 2.2: Illustration of the decomposition of the log probability of $\boldsymbol{D}$ into the sum of the reverse KL Divergence and $L(q)$ (the Evidence Lower Bound or ELBO). Extracted from [2].

3

Partitioned Variational Inference

In this chapter I will introduce the idea of Partitioned Variational Inference, a framework designed to unite Variational Inference and Federated Learning.

In a Federated setting the task we need to solve is to model a dataset $\boldsymbol{D}$, partitioned into N local datasets $\boldsymbol{d_i}$, one for each of the clients in the system. With a parametric model with a prior $p(\boldsymbol{\theta})$ over the parameters $\boldsymbol{\theta}$ of the model and a likelihood $p(\boldsymbol{D}|\boldsymbol{\theta}) = \prod_{i=1}^{N} p(\boldsymbol{d_i}|\boldsymbol{\theta})$.

However if we tried to perform Variational Inference in a setting like these, we would need to have access to the entire dataset and that is forbidden in Federated Learning.

Partitioned Variational Inference instead decomposes the variational approximation by the product of approximate local factors to avoid the need to have access to the whole dataset.

$$p(\boldsymbol{\theta}|\boldsymbol{D}) = \frac{1}{Z} p(\boldsymbol{\theta}) \prod_{i=1}^{N} p(\boldsymbol{d_i}|\boldsymbol{\theta}) \approx \frac{1}{Z_q} p(\boldsymbol{\theta}) \prod_{i=1}^{N} t_i(\boldsymbol{\theta}) = q(\boldsymbol{\theta}) \tag{3.1}$$

Basically, we are substituting the marginal likelihood Z of the real posterior with a normalisation factor Z_q, and the real likelihood of each client $p(\boldsymbol{d_i}|\boldsymbol{\theta})$ with an approximate likelihood $t_i(\boldsymbol{\theta})$. Algorithm 3.1 shows a basic skeleton of the algorithm.

The skeleton of the algorithm remains the same for any of the three learning modes of Partitioned Variational Inference, each one characterised by the schedule followed when choosing clients for training:

- Sequential: the server only chooses one client to calculate a new likelihood at each iteration.
- Synchronous: all likelihoods are calculated at each iteration, but all likelihoods must have been calculated before making a new iteration.
- Asynchronous: as soon as a client calculates its likelihood, the server updates the approximate posterior and sends it back to the client to calculate another likelihood.

Each one has its benefits and drawbacks. The sequential mode has the property of automatically

obtaining a normalised posterior ($Z_q = 1$) but is generally slower and its very susceptible to the order chosen for training. Synchronous and asynchronous modes are generally faster but usually require of damping in order to try and avoid obtaining an improper posterior.

This damping is done on the calculation of the new local likelihood of each client: $t_k^{(i)}(\boldsymbol{\theta}) \propto \left(\frac{q_k^{(i)}(\boldsymbol{\theta})}{q^{(i-1)}(\boldsymbol{\theta})}\right)^{\rho} t_k^{(i-1)}(\boldsymbol{\theta})$, where $\rho \in (0, 1]$ is the damping factor and i is the iteration number in algorithm (3.1). In [1], a factor of $\rho \propto \frac{1}{N}$ was found to result in stable convergence.

Input: data partition $\{\boldsymbol{d_1}, \ldots, \boldsymbol{d_N}\}$, prior $p(\boldsymbol{\theta})$.

Output: approximate posterior $q(\boldsymbol{\theta})$.

Initialise:

$$t_j^{(0)}(\boldsymbol{\theta}) = 1 \text{ for all } j = 1, 2, \ldots, N.$$
$$q^{(0)}(\boldsymbol{\theta}) = p(\boldsymbol{\theta}).$$

for $i = 1, 2, \ldots$ until stop condition (convergence, maximum number of epochs, ...) **do**

Server: selects the next clients.

$b_i =$ clients chosen for the next round of training

Communicate $q^{(i-1)}(\boldsymbol{\theta})$ to all the chosen clients b_i.

Client: calculation of new local likelihood

for $k \in b_i$ **do**

Calculation of the new local approximate posterior:

$$q_k^{(i)}(\boldsymbol{\theta}) = \underset{q(\boldsymbol{\theta}) \in Q}{argmax} \int q(\boldsymbol{\theta}) \log \frac{q^{(i-1)}(\boldsymbol{\theta}) p(\boldsymbol{d_k}|\boldsymbol{\theta})}{q(\boldsymbol{\theta}) t_k^{(i-1)}(\boldsymbol{\theta})} \mathrm{d}\boldsymbol{\theta}.$$

Calculation of the new approximate likelihood:

$$t_k^{(i)}(\boldsymbol{\theta}) \propto \frac{q_k^{(i)}(\boldsymbol{\theta})}{q^{(i-1)}(\boldsymbol{\theta})} t_k^{(i-1)}(\boldsymbol{\theta})$$

Communicate the difference in likelihoods $\Delta_k^{(i)}(\boldsymbol{\theta}) \propto \frac{t_k^{(i)}(\boldsymbol{\theta})}{t_k^{(i-1)}(\boldsymbol{\theta})}$ back to the server.

end for

Server: Calculation of the new approximate posterior

$$q^{(i)}(\boldsymbol{\theta}) \propto q^{(i-1)}(\boldsymbol{\theta}) \prod_{k \in b_i} \Delta_k^{(i)}(\boldsymbol{\theta}).$$

end for

Algorithm 3.1: Partitioned Variational Inference Skeleton

To calculate the new factors of each client, the procedure is:

1. After receiving the last updated posterior, the last approximate likelihood factor of the client is extracted. The result is used as the new prior for the next variational inference optimisation step. $new\ prior = \frac{q^{(i-1)}(\boldsymbol{\theta})}{t_k^{(i-1)}(\boldsymbol{\theta})}$. This optimisation can be done in multiple ways. The easier one is to choose a model that provides closed form (analytical) solutions to the optimisation step, but other

alternatives are available like the use of optimisers such as Adam or Natural Gradients methods.

2. After calculating the new approximate posterior. The new approximate likelihood is calculated by dividing the new approximate posterior by the new prior calculated in step 1. $new\ likelihood = \frac{q_k^{(i)}(\boldsymbol{\theta})}{q^{(i-1)}(\boldsymbol{\theta})} t_k^{(i-1)}(\boldsymbol{\theta})$.

3. The difference between the old approximate likelihood factor and the new approximate likelihood factor is calculated and sent to the server. $Delta = \frac{t_k^{(i)}(\boldsymbol{\theta})}{t_k^{(i-1)}(\boldsymbol{\theta})}$

Most of these operations are simplified by the selection of model. If the model chosen for both the prior and the posterior belongs to the same Exponential family of distributions, the likelihood approximate factors also belong to the same family, making it so that all distributions involved in the process belong to the same family.

This is due to the fact that the multiplication and the division of Exponential Family distributions' probability density functions is equal to another (unnormalised) Exponential Family distribution density function whose natural parameters are equal to the addition or subtraction respectively of the natural parameters of both operands, and, if both Exponential Family Distributions belong to the same family (eg. Gaussians, Exponential, Bernoulli ...), the resulting distribution also belongs to the same family. That is why if the prior and posterior chosen belong to the same Exponential family distribution, most of the calculations are translated into simple sums and subtractions of natural parameters, which makes them perfect candidates for surrogate posteriors in Partitioned Variational Inference.

3.1 Design

Before any coding was done or any infrastructure was setup, a thorough planning had to be done to ensure no amount of work was wasted and that every requisite was fulfilled.

In this section, the requisites and architecture that have been used as baseline for this project will be explained.

3.1.1 System Architecture

In this architecture, we have three key components:

- The server: presumed to have high resources and the key coordinator of the training. There is only one in the whole architecture.
- The clients: presumed to have low resources and each of them contains different data for the training. There are potentially infinite clients.

- The data: The data is presumed to be unbalanced (one client may have thousands of MB of data while another one may have a few KB). The data must never leave its owner's computer.

The server and the clients could be any process in any computer, but they must meet the following requirements:

- Each client must be able to send packets to the server.
- The server must be able to send packets to each client.

It is important to highlight the fact that each of the clients does not need to be able to communicate with each other, as this is a centralised network and all the communication happens between the leaves and the central server and vice versa.

This however also means that the server represents a single point of failure in our system and will be subjected to great network loads. Additionally, the network itself can become overloaded if a great amount of clients belong to the same network.

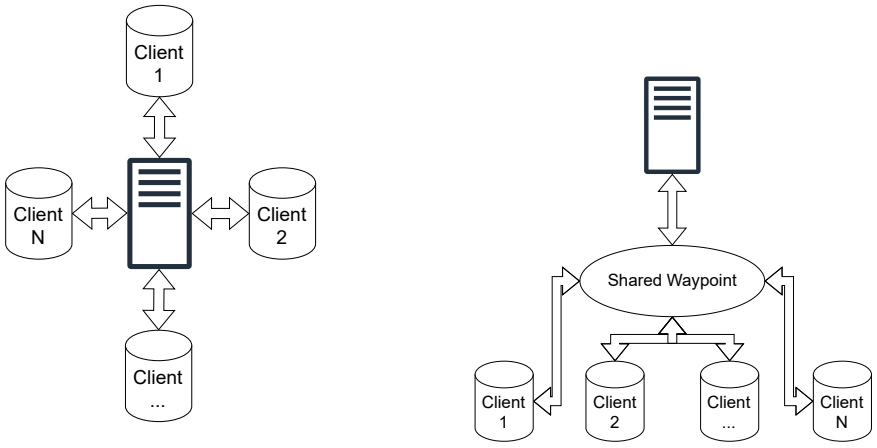
The first problem can be solved with high availability. The protocol contemplates the case of a server going down or transferring its role to another server. So the solution would be to have multiple servers available for training and switching between them when one goes down. In the same vein, a client can also shut down or transfer its role.

This is made possible by the fact that the protocol has been designed with these cases in mind. Each approximate likelihood factor is stored both in the server and in the client, so each of them can signal a transfer, or recover from a fatal shutdown by requesting the likelihood from the partner. The server additionally requires a prior, but this should be either fixed for the execution of the program, or should be able to be retrieved from an external source.

From here on we will assume the structure of just one central server and M clients for all diagrams and explanations, as the high availability does not change the protocol except for the identity transfer or fatal shutdown cases, where the different possible numbers of servers and clients will be addressed.

The second problem is more of a hardware and network architecture problem and it is hard to make any claims about it without further knowledge of the specific network architecture where the framework will be used, but as a rule of thumb, the network should either be as flower shaped as it can be (node in the centre connecting with all its leaves independently), or be able to handle great network loads. Both kinds of architectures are illustrated in Figure (3.1).

This may not seem like a complex problem to solve, but having clients that can potentially transfer ownership and move across the world greatly increases the complexity of the system. It is also a problem that cannot be ignored, being this a distributed architecture. In some learning modes, even if just one branch slows down, it can slow down the whole training, or, if it breaks down, the training itself



(a) Independent connections with each client (flower shaped)

(b) Shared section of the network. If the shared waypoint and the connection from the waypoint to the server are not prepared to handle great network loads it could lead to problems, as they constitute the bottleneck of the system.

Figure 3.1: Some examples of network architectures that could arise in the circumstances of training in a Partitioned Variational Inference setting

may need to be cancelled.

3.1.2 Protocol

The communication between client and server must follow some kind of specification so that they can understand each other and collaborate.

Given the heterogeneous nature of both clients and servers in our architecture, any kind of communication based on language specific serialisation, OS or architecture had been discarded. Instead, I chose to leave the serialisation schema up to the implementors of the protocol, and only specify the nature of the protocol itself. However, even if the serialisation format is not specified, the messages are required to have its length at the first 32 bits (big endian).

I chose to create a message based protocol, as I found that it fit quite well with the Partitioned Variational Inference algorithm. The Partitioned Variational Inference algorithm consists mainly on waiting, receiving a message with a new posterior and sending back a new local approximation, which adjusts very well with the aforementioned message based approach. There are additional considerations during the Partitioned Variational Inference algorithm that should also be addressed by the protocol as I will show, but they do not conflict with this schema of communication.

The protocol differentiates between Client messages and Server messages, and only starts once

the TLS connection has been established. The communication protocol is built upon TLS, as it is meant to be used with critical data. The data itself never leaves its host, but I found it necessary to encrypt the data obtained from the training and the communication itself. From an implementation point of view, TLS also provides a quite convenient way of identifying clients and servers based on their respective certificates. Figure (3.2) illustrates the protocol's workflow.

The protocol requires that an implementor at least implements the following messages, but is open to the addition of new types of messages:

Client Messages

- JoinCluster: used by the Client to initially join the cluster for the training. Requires two fields:
  1. data size: specifies the size of its data in number of elements (does not need to be accurate, but it is supposed to be used as a way to normalise loss and measure convergence). Type: u64 [1].
- ReJoinCluster: used by the Client to rejoin the cluster after some Client required disconnection. It does not require any field.
- UpdatedLikelihood: used by the Client to return the updated likelihood after one round of training. Requires three fields:
  1. new likelihood: the new approximate likelihood. Type: Distribution args [2].
  2. delta: ratio between previous and new likelihoods. Type: Distribution args.
  3. loss: local loss in this round of training. Type: f64.
- ReturnLastLikelihood: used by the Client to return the last updated likelihood, meant to mainly be used by the server after a disconnection to restore its previous state. Requires three fields:
  1. likelihood: last calculated local likelihood. Type: Distribution args.
- EarlyLeaveCluster: used by the Client to disconnect from the training earlier than expected. It has two optional fields:
  1. reason: reason for the early disconnect. Type: String.
  2. expected absence: expected time for the disconnection. If not present, the absence is assumed to be permanent. Type: Duration expressed as a struct with both seconds and nanoseconds whose sum represents the duration of the absence.
- FinalLeaveTraining: used by the Client to disconnect from the training after reaching the end of the training. Requires one field:

[1] Numerical types are formatted as character followed by number of bits. The characters represent [u]nsigned integer, s[i]gned integer and [f]loat

[2] Varies based on the distribution being trained, but should contain all the information to fully recreate the distribution. For example, for a Gaussian Distribution, it could both be the natural parameters or the mean and standard deviation, but just a mean would not suffice.

1. available for future training: boolean representing the availability of the Client for future training. It represents only an estimation and does not represent any type of binding responsibility. Type: boolean.

- EndOfConnectionAcknowledgement: used by the Client to acknowledge the disconnection from the server after receiving a server message signalling an early stop. It does not require any field.
- Error: used by the Client to signal an error in the communication, like a malformed message or a message not expected. It has one optional field:

  1. reason: should contain a brief reason for the error. Type: String.

Server Messages

- AcceptedIntoCluster: used by the Server to accept the request from a Client to join the cluster for the training. It has one optional field:

  1. expected start time: specifies the expected start of the training. Type: UTC DateTime

- ReAcceptanceIntoCluster: used by the Server to accept a rejoin into the cluster after some Client required disconnection. To avoid Client impersonation, the TLS certificate should also be checked with the previous one to authenticate the Client. It only requires one field:

  1. last likelihood: last local likelihood returned by the Client previously in the cluster. Type: Distribution args.

- RejectionFromCluster: used by the Server to reject the request from a Client to join the cluster for training. It has two fields:

  1. reason: it is optional and should contain a brief reason for the rejection. Type: String.
  2. fixable: true if the rejection can be fixed by, for example, changing the certificate or reducing the data size, or false if the rejection is permanent. Type: boolean.

- SelectedForTraining: used by the Server to signal to a Client that it has been chosen for the next round of training. It contains two fields:

  1. current posterior: current posterior. Type: Distribution args.
  2. damping factor: optional argument signaling the damping factor to be used when calculating the newly updated approximated likelihood. Type: f64

- EarlyCloseOfConnection: used by the Server to disconnect from the training earlier than expected. It has two optional fields:

  1. reason: reason for the early disconnect. Type: String.

2. return in: time for the clients to join again for the training. If not present, the absence is assumed to be permanent. Type: Duration expressed as a struct with both seconds and nanoseconds whose sum represents the duration of the absence.

- EndOfTraining: used by the Server to signal the end of the training. It has two fields:
  1. final posterior: final posterior obtained as a result of the training. Type: Distribution args.
  2. future training: optional estimated date for the next training. Type: UTC DateTime.
- EndOfConnectionAcknowledgement: used by the Server to acknowledge the disconnection from the Client after receiving a message signalling an early leave. It does not require any field.
- Error: used by the Server to signal an error in the communication, like a malformed message or a message not expected. It has one optional field:
  1. reason: should contain a brief reason for the error. Type: String.

3.1.3 Software Architecture

The framework is inherently divided into four different sections:

- Shared code between client and server: the creation of the TLS connection, the protocol implementation, the serialisation of the messages and the probability distributions related code.
- Server related code: the handling of all the different connections, the different training modes, communication with the client, updating the posterior and coordinating the Partitioned Variational Inference algorithm.
- Client related code: handling of connection to the server, handling of the training cycle, training operations (like substraction of likelihood factors) and communication with the server.
- Python interfacing: all the code related to the interoperation between Rust and Python.

I decided to divide this code in two main components, a library in Rust which contains all the core logic and code needed for a Partitioned Variational Inference training in Rust, and a library, also written in Rust but meant to be used from Python, to interface with the aforementioned library.

The library itself is structured in four different modules: server, client, distributions and communication (this last one handles both the protocol, the TLS communication and the messages part of the serialisation). Figure (3.3) contains a diagram showing the general structure of the main library.

Due to the nature of the training, the framework is meant to be used asynchronously and it is designed as such. However, in the Python interfacing, I chose to adapt the interface and make it

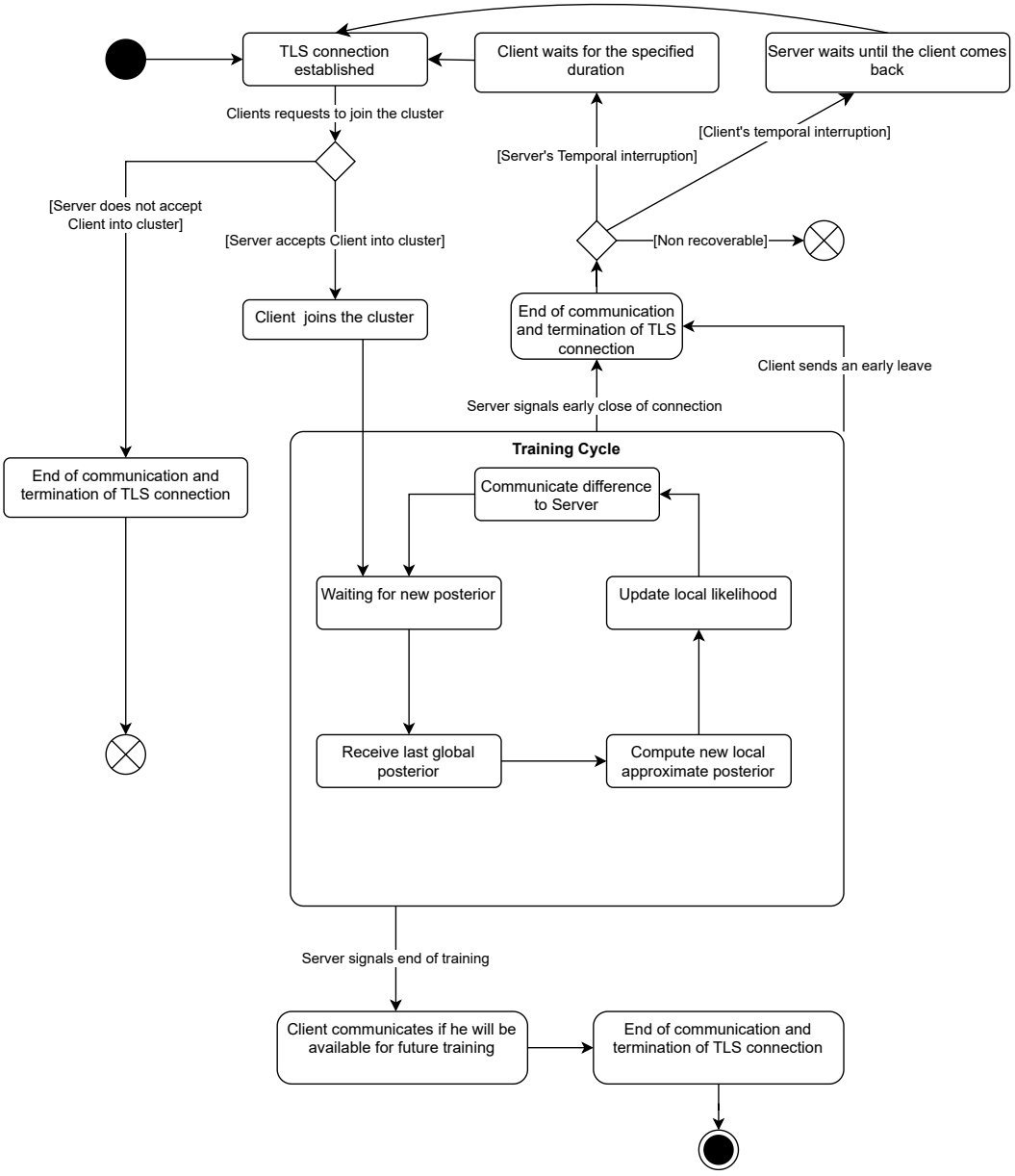


Figure 3.2: Flow Diagram depicting the workflow of the Partitioned Variational Inference protocol

synchronous as I found it much more usable and, compared to the server, the ease of use obtained by making the client synchronous greatly outweighs the performance cost in the client.

Regarding the communication module, it is itself divided in two modules: one for the TLS handling, and one for the communication protocol and messages.

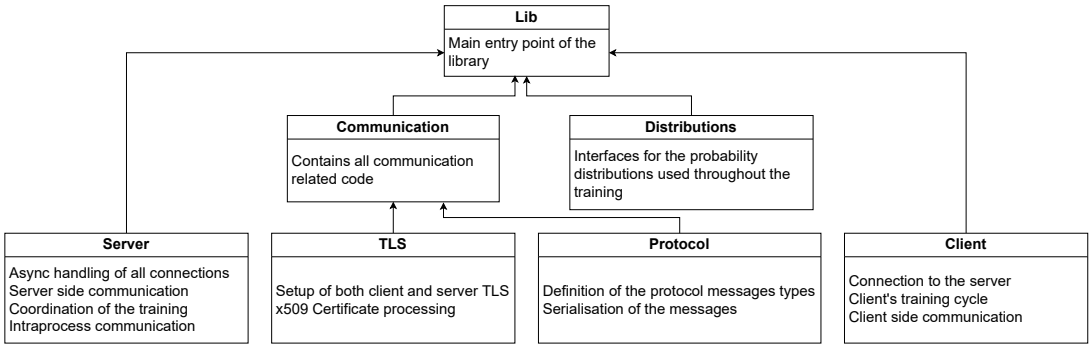


Figure 3.3: Diagram of the modules of the main library

I decided to implement this architecture as an actor based system where each of the components communicate with each other by the use of well defined messages, think of them like Erlang, Akka or Orleans actors. However, the goal of this system is to facilitate the development of the project and thus, some purity may be scrapped in some places in favour of ease of use or ease of development.

Even if the channels used for communication will be different, as there will be communication between components in the same machine and communication between components in different machines, this actor model greatly simplifies working with the distribution of tasks.

3.2 Implementation

In this section, the details regarding the final implementation will be displayed. It will be divided in sections regarding each part of the framework so that it is easier to navigate.

I will explain the relevant crates (that is the name libraries are given in the Rust ecosystem) in their respective subsections but there is one crate that is essential to the whole framework and thus I will give a brief explanation here.

In Rust, asynchronous programming is part of the standard library. However, it is runtime agnostic, so if we want to run asynchronous code we will need to choose a runtime. Designing the library to be runtime agnostic is also posible, but it brings quite a few problems and greatly increases the complexity of the system as the async ecosystem is still quite young. I would also have to conform with the minimum set of shared capabilities between all runtimes, so I instead chose to work with Tokio's runtime as I found it to be quite feature complete, simple to work with and it has a great number of

associated crates that provide many different functionalities.

The main features of Tokio that I will be using aside of being a runtime are:

- Spawning tasks: Tokio allows through a very simple interface to spawn tasks (also called green threads or fibers in other languages). There are two main ways to spawn tasks, in a non blocking way with the simple spawn function, or for long running tasks or tasks that require blocking, spawn blocking. This last option sends the task to a different designated thread where it is acceptable to block.
- Creating channels: Tokio provides a mpsc (multiple producer single consumer) channel type which is used throughout the library for communication of the different actors.
- select! macro: Tokio also provides a macro to simultaneously wait on different Futures (the return value of asynchronous functions, also called Tasks or Promises in other languages) and work on the first one to resolve. This is used mainly to consume messages from different sources but will be explained in more detail in each specific subsection.
- The integration with relevant crates: there are multiple crates that I needed to use like Rustls for TLS, or Serde for serialisation, for which Tokio already provides interfaces integrated with the runtime.
- TCP: Tokio comes with TCP structures designed to work well with its runtime and, given that I will be working with TLS at a TCP level, this reduces some of the inherent complexity of working with sockets at a low level.

This section will generally refer to the general Rust crate unless indicated otherwise, as the Python crate is a wrapper around the general crate. Even then, the implementation of this wrapper is far from trivial and will be addressed in the last subsection of this section.

3.2.1 Network Communication

Encryption

The encryption of the data sent between the Server and the different clients is based on TLS. The Rustls crate has been used to handle the connection and the verification of the certificates.

The x509-parser crate is also used to provide additional information about the certificates if the user so desires, like common name, country, company, etc.

My framework offers simple functions for setup for both the client and the server in the respective, setup_client_tls and setup_server_tls functions. These functions accept the path to the different required files with the keys and certificates and return a TCP connection wrapped by a TLS stream

provided by the tokio_rustls crate. Specifically, in the server case, it returns a TLSAcceptor struct which will return a TLS stream when provided with a TCP stream after each client connects.

This stream is an abstraction over the socket, providing the same interface as the underlying TCP stream and automatically encrypting and decrypting the data sent through it.

It will work with any valid TLS setup given that the certificates are signed by any of the trusted CAs passed to the functions. However, the intended way of using this part of the library, is to have a specific CA for the training generate certificates for all the members involved and only use trusted certificates signed by that CA, to avoid authentication attacks related to other CAs.

Serialisation

The serialisation is a big part of the framework as, this being a distributed framework, all the messages must be serialised in one way or another from the server to the clients and vice versa.

For this task, I used the Serde framework, which by the use of macros, derives some traits (also known as interfaces in most other languages) to allow automatic serialisation and deserialisation. Then other libraries use this traits to serialise and deserialise in various formats.

In our case, I am using the Tokio framework as the async runtime, so I also used a crate to automatically serialise and deserialise from the TCP connection called tokio_serde. In addition to the tokio_utils crate, both crates provide a Framed struct which wraps the underlying stream of data and acts upon it to read and write the data according to a Codec. The LengthDelimitedCodec codec is used to prepend the length of each message, and depending on the serialisation format chosen, a Json, Cbor, Bincode or MessagePack codec is used to serialise the messages.

Even if some of the formats are self describing, I found it best to always prepend the length to avoid any ambiguities. In addition, given the high performant nature of the requirements of the training and the desire to have the lowest message size, most binary formats will be given preference, which usually are not self describing. Another advantage of having access to the message's length is that, given that our communication is over TCP, we can easily know what to expect when reading from the socket and read the exact amount needed.

The communication pipeline then follows the following pattern:

Serialisation

1. Transformation of the message into the required serialisation format.
2. The length is calculated and it is prepended to the message.
3. The whole message, length included, is encrypted using TLS.

Deserialisation

1. The message is decrypted using TLS.
2. The length is read from the starting 32 bits.
3. The message is reconstructed according to the serialisation format.

Protocol

The protocol implementation can be divided in two parts: the messages implementation and the handling of the messages themselves.

The messages are implemented using enumerations. In Rust enumerations are what is commonly called Algebraic Data Types in most functional programming languages. This type of enums are very powerful and are one of the primary reasons I chose Rust to implement Partitioned Variational Inference. They do not only allow for a very powerful kind of pattern matching, they also allow to represent complex data with great expressiveness.

Rust allows different kind of enum variants, but for the protocol messages I wanted to be explicit. Hence, I represented the client messages and server messages with two struct enums, ClientMsg and ServerMsg respectively. Code (3.1) contains the source code of both types.

Each enum has a variant for each of the different messages of the protocol and has fields with the names specified in the protocol, so that when the Serde macros for serialisation and deserialisation are applied they produce the right output. The optional arguments are represented by the Option enum wrapping the type (an optional type, very common in functional programming languages). Both enums are generic regarding the Distribution Args for the likelihood and posterior fields, imposing the restriction that they implement a trait for creating a valid distribution (more on that on the learning section).

Generics in Rust work by monomorphizing. These means that for each instance of the generic type, the compiler generates code for it. For example, if I had a Vec<T>, representing a vector of Ts, if I used a Vec<u32> and a Vec<bool>, the code of Vec would be modified to generate both a Vec<u32> and a Vec<bool> implementation.

This approach has both advantages and disadvantages, it is faster than a dynamic dispatch approach because there is no indirection with a vtable (even though this approach is also possible in Rust), but it also means that the binary size increases. Normally the trade off is favourable and this approach is great, but it can also become very complex when introducing lifetimes and deeply nested types.

I will not delve deep on what lifetimes are in Rust, but they are essentially the time that a reference will be alive, because in the Rust borrow model, once a variable stops being used, its memory is freed. Lifetimes exist to avoid use after free errors and are one of the greatest achievements of Rust in my

Code 3.1: Both enums used for the message passing between the server and the clients for protocol based communication.

```
#[derive(Serialize, Deserialize, Debug, Clone)]
pub enum ClientMsg<T>
where T: TrainableModel,
{
    JoinCluster {data_size: u64},
    ReJoinCluster,
    UpdatedLikelihood {
        new_likelihood: T::LikelihoodFactor,
        delta: T::Delta,
        loss: f64,
    },
    ReturnLastLikelihood {
        likelihood: T::LikelihoodFactor,
    },
    EarlyLeaveCluster {
        reason: Option<String>,
        expected_absence: Option<Duration>,
    },
    FinalLeaveTraining {
        available_for_future_training: bool,
    },
    Error {error: Option<String>},
    EndOfConnectionAcknowledgement,
}
#[derive(Serialize, Deserialize, Debug, Clone)]
pub enum ServerMsg<T>
where T: TrainableModel,
{
    AcceptedIntoCluster {
        expected_start_time: Option<DateTime<Utc>>,
    },
    ReAcceptanceIntoCluster {
        last_likelihood: T::LikelihoodFactor,
    },
    RejectionFromCluster {
        reason: Option<String>,
        fixable: bool,
    },
    SelectedForTraining {
        current_posterior: T::ParametricPosterior,
        damping_factor: Option<f64>,
    },
    EarlyCloseOfConnection {
        reason: Option<String>,
        return_in: Option<Duration>,
    },
    EndOfTraining {
        final_posterior: T::ParametricPosterior,
        future_training: Option<DateTime<Utc>>,
    },
    Error {error: Option<String>},
    EndOfConnectionAcknowledgement,
}
```

opinion in comparison to C or C++.

This kind of generics make it very easy for users of the framework to implement their own different data structures representing different distributions and use them. They just need to implement the necessary traits and the compiler will do the rest.

The handling of the messages themselves is relegated to various instances of pattern matching. Both life cycles of the protocol are represented internally as state machines, and the processing is done with pattern matching on both the state and the message (or in some cases in the client, the state is omitted as it is implied, like when joining the server) and checking the combination of both. Then, if the combination is defined, the message is processed and the state changes accordingly to the state machine. If it is not or there is an error with the message itself, an error message is returned. In the implementation of the protocol, the workflow of role transfer has not been implemented as its handling requires vastly different solutions depending on the user of the library.

3.2.2 Workflow

In this section, I will describe how the usual workflow is implemented in the framework. For a code example of a normal workflow from a user perspective, refer to the examples found in the code repository in Appendix (A).

Client

The client workflow starts by a call to setup_client_tls, then with that stream an struct is created to hold the communication stream, the current likelihood and the current state.

The client then sends the corresponding ClientMsg to join the cluster through the join_cluster function. If accepted, then the state is modified and the next step is to call the wait_for_posterior function.

This starts a cycle where the client waits for the next posterior, obtains the next approximate likelihood, sends the updated data using the send_updated_likelihood function and then waits again.

This continues until either the client needs to leave early, the server needs to leave early or the training is finished.

As specified before, all the handling of the messages and the state machine associated is done by pattern matching.

The client workflow does not have much complexity due to the fact of the crate practically acting as a state machine and communication channel. The main complexity of the client rests on the learning.

Server

On the other hand, the server workflow is much more complex as it has to handle all the different connections with each client, the coordination of all the tasks, training and input of the server.

The server user's workflow starts by calling setup_server_tls and ends by calling the create_server function. However, even if the interface for the user is simple, there is a lot of work behind the scenes.

The workflow is managed by the following actors:

- Input handler: actor spawned in a blocking task for handling the input and output of the program.
- Coordinator: actor responsible for handling the events sent by the input actor, sending output events and handling the general state of the training. This actor decides when to start the training, when to disconnect, which schedule to follow, etc.
- Connection handler: actor responsible of creating new TCP connections and wrapping them in a TLS stream for each client. Then, for each client spawns an actor to handle that specific clients communication.
- Client actor: there is one for each client. They handle the communication with each client directly through the TLS streams. The communication with the coordinator is done through the same channel for all clients, so they must also send the client id. The client id in this case is an index into a vector that the coordinator assigns. The real identification is in that vector, where the TLS certificate is stored and serves as a unique identifier for each client. The same is not true for the inverse case, the coordinator communicates with each client actor directly.

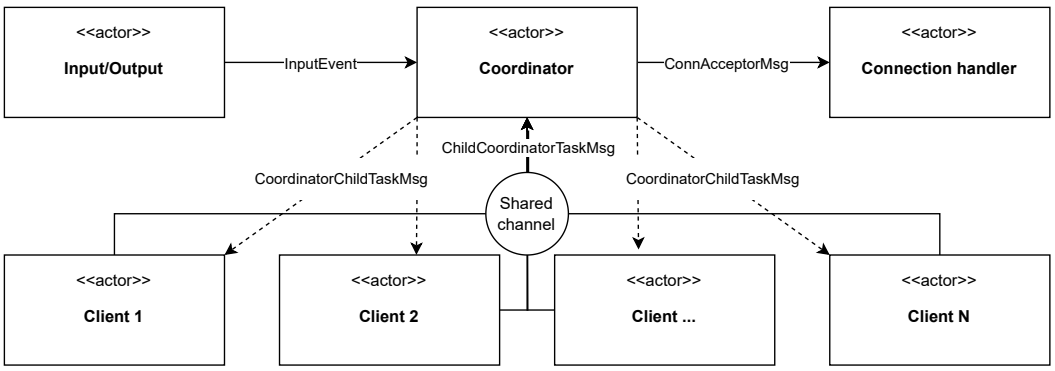


Figure 3.4: Diagram showing the layout of the actors in the system and the types used as messages between them. The communication channels between the coordinator and the clients are dotted to further reinforce their different cardinality and to avoid any confusions over the overlapping channels.

The connection handler does not send any messages himself, he just receives messages from the coordinator and, when it is required, accept connections and spawn new client actors, which will communicate on their own.

Inside the create_server function, the first three actors are created and then the task awaits until the shutdown of these actors.

All actors communicate with each other using Tokio's unbounded mpsc channels. These channels are generic in regards to the information being sent. So for each actor pair combination that has communication (for example, the input actor does not communicate with each client actor), their respective messages are represented with an enum type and the channels are created for those types. Figure (3.4) contains a diagram showing the general schema of communication inside the application and the types used as messages.

All actors make use of Tokio's select! macro to await their respective channels (and TLS streams in the case of the client actors) at the same time, and process the messages as soon as they arrive. Code (3.2) contains an example of the use of this macro.

Rust's pattern matching is exhaustive, so all the different message possibilities are guaranteed to be handled. The select! macro can cause some problems if it is used with functions that are not cancellation safe (imagine reading half of a message and then a whole message arrives from another channel), but luckily both the mpsc channels and the TLS streams used are cancellation safe.

Code 3.2: Simplified Coordinator event loop's code, handling consuming from the multiple channels by using the tokio::select macro.

```
loop {
        tokio::select! {
            input_event = inp_coord_receiver.recv() => {
                /// Handle Input Event
            },
            client_event = client_coord_receiver.recv() => {
                /// Handle Client's Message
            }
        }
}
```

The main workflow goes as follows:

1. All the three main actors are initialised after calling create_server.
2. The connection handler accepts connections and spawns client actors.
3. The client actors check if the user is able to join the cluster or not by sending a message to the coordinator, and if they can, send a message to the client and await for training.
4. The coordinator decides that training starts due to different reasons like: maximum number of clients has been reached, the time desired for training has arrived or the training has been ordered to start from standard input. He sends a message to the connection handler to stop accepting connections.

5. The coordinator initialises the training related information (like the damping factor) and starts the following cycle.
   5.1. The coordinator, following a schedule based on the Training Mode, sends messages to the clients actors choosing them for training and awaits for messages.
   5.2. The client actor receives the message and then sends a protocol message to the Client and awaits for messages.
   5.3. The client actor receives the updated likelihood via protocol from the Client, sends a message to the coordinator and awaits.
   5.4. The coordinator receives the updated likelihood, updates the posterior and restarts the cycle.
6. This cycle ends when either an unexpected situation arises like an early leave or when the training goals have been reached (maximum number of epochs, convergence, etc.).
7. The coordinator sends end of training messages to the client actors and awaits.
8. The client actors send the protocol messages signalling the end of training and await.
9. The client actors receive the protocol messages and send them to the coordinator, then shutdown.
10. The coordinator processes these messages, sends signals for shutdown to the input and connection actors, and then shutdowns, returning the final posterior achieved.
11. The input and connection actors shutdown.
12. The function returns the final posterior obtained.

All the other protocol messages are handled in the client actor, within the state machine. They are not of great interest, they simply fullfill the protocol specification explained in the design and if any state message combination is not valid or the messages themselves are wrong, they send an error.

3.2.3 Learning

The learning follows the Partitioned Variational Inference learning algorithm described at the start of the section. The operations required are generalised to multiple distributions by the use of the methods defined on the TrainableDistribution trait, so the handling of the oddities and specifics of the different distributions is done by the user. This trait contains all the necessary operations specified in the algorithm, so if one implements it for an user created distribution, the algorithm can also be implemented for that distribution.

Nevertheless, the framework handles some distributions right off the bat, and offers a ExponentialFamilyDistribution trait that handles the addition and substraction of likelihoods by the use of natural parameters. The user only has to define the bijection between the distribution and the natural parameters. Code (3.3) contains the definition of both traits.

Code 3.3: Code for both the TrainableDistribution and ExponentialFamilyDistribution traits

```
pub trait ExponentialFamilyDistribution:
    Send + Serialize + DeserializeOwned + Clone + Unpin
{
    type NaturalArgs: Add<Output = Self::NaturalArgs>
        + Sum
        + Sub<Output = Self::NaturalArgs>
        + Mul<f64, Output = Self::NaturalArgs>;

    fn from_nat_params(args: Self::NaturalArgs) -> Self;

    fn nat_params(&self) -> Self::NaturalArgs;
}

pub trait TrainableModel {
    type ParametricPosterior: Send + Serialize + Unpin + DeserializeOwned + Clone;

    type LikelihoodFactor: Send + Serialize + Unpin + DeserializeOwned + Clone;

    type ShapeWithoutLikelihood;

    type Delta: Send + Serialize + Unpin + DeserializeOwned + Clone;

    fn update_posterior(
        curr: Self::ParametricPosterior,
        deltas: impl IntoIterator<Item = Self::Delta>,
    ) -> Self::ParametricPosterior;

    fn extract_likelihood_factor(
        curr: Self::ParametricPosterior,
        factor: Option<Self::LikelihoodFactor>,
    ) -> Self::ShapeWithoutLikelihood;

    fn calculate_delta(
        past_likelihood: Option<Self::LikelihoodFactor>,
        new_likelihood: Self::LikelihoodFactor,
    ) -> Self::Delta;

    fn calculate_new_likelihood(
        approximated_posterior: Self::ParametricPosterior,
        past_posterior: Self::ParametricPosterior,
        past_likelihood: Option<Self::LikelihoodFactor>,
        damping: Option<f64>,
    ) -> Self::LikelihoodFactor;
}
```

Client

The client's learning process is simple, same as before with the workflow. This is due to the fact of the computationally complex part of the learning outsourced to the client (the minimisation of the loss). However, the client part of the framework still has to manage part of the calculations. Specifically, the client needs to handle the subtraction and addition of likelihoods (subtraction and addition as in the component aspect, the real mathematical operations are the product and the division). In the synchronous and asynchronous learning modes, the client library also handles the damping of the factors.

Aside from that, the main advantage of the framework is that it comfortably handles the logic behind the algorithm and, thanks to the state machine implementation, ensures that the system does not enter any inconsistent state.

Server

The server learning process is quite similar to the client's one. Once a new likelihood is obtained, by using the methods of the various distribution traits, the server extracts and adds the likelihoods obtained.

Here, the state machine implementation is quite more relevant, as the server has to handle multiple clients and it is easier to enter in an inconsistent state, specially when taking into account the different learning modes.

The more interesting part of the learning aspect of the server are its different learning modes. To avoid having to model different client actors and coordinators, the training algorithm was abstracted to a method of the TrainingHandler enum. This enum stores in each of its variants different data to keep track of the schedule:

- Sequential: a vector with client ids and an increasing index that circles around when reaching the end.
- Synchronous: a number indicating the remaining clients training for the round and a damping factor.
- Asynchronous: a damping factor.

The coordinator calls the handle method of its TrainingHandler every time it receives a new likelihood. This enum then pattern matches on itself, and according to the training mode, accesses its schedule data and applies its schedule.

3.2.4 Python Interfacing

Mixing a strongly typed compiled language such as Rust with Python brings along large amounts of complexity due to the vast difference between these two worlds. Thankfully, as it is common to use libraries from compiled languages such as C or C++ in Python for faster computations, Rust is not an exception and there exist crates to facilitate this interoperation.

Concretely, I have used Pyo3. Pyo3 is a great crate that handles lots of different needs for interoperation between Python and Rust. In my case, my needs are quite simple as I only require calling Rust from Python and wrapping my own library. However, the possibilities are quite more extensive, being able to call Python from Rust, use async, manipulate the GIL, consume lambdas and many more.

This is great and certainly reduced the amount of work I had to put in, but there was one big problem that I could not avoid. This problem was the generalisation of my types. Python is a scripting language, so there is no way to know ahead of time with which types my generic types are going to be called and generate the necessary code. This was other of the consequences of monomorphization instead of dynamic dispatch, like Python does.

There were two solutions to this problem, generating the necessary code myself for the generic types that I wanted to use or abstracting the types away and losing type insurance. I was in this situation in the first place because I wanted my types to guide my implementation and protect me from errors, so I chose the first one. Instead of hard coding all the different possibilities, which would have taken a long time, I created a macro that when provided with a type and a name, it would create all the associated code necessary for that type. This fixed the problem, but the only inconvenient is that currently is hard to add new user defined distributions only from Python without having to step into Rust.

The next problem consisted in the separation of concerns between the general crate and the Python crate. The Python crate's functions, structs and methods that wanted to step into the Python world needed to be annotated with some Pyo3 macros that generated the necessary code for that. However, I did not want to annotate the functions in the general crate with these macros, why should a Rust only user of the framework have to bear the costs of the Python interoperability?

There were once again two options, either copy the code and annotate it in the Python crate, which seemed very prone to errors and repeated a lot of code, or find a way to wrap the code. I chose this last option because I already had to wrap the automatically generated code from before with this new traits, so now I would simply have to generate some more wrapper structs and enums and apply the macros on them. This adds a layer of indirection that sometimes clutters a little bit the interface, but I found it acceptable for the benefits obtained of full and easy compatibility with the general crate.

4

Experiments

To test the efficacy of this framework, a series of experiments was carried out on two datasets. The infrastructure of the experiments consisted of a network with two physical computers. The first one was a Linux desktop that hosted the server and part of the clients, and the other one was a Windows laptop that hosted only clients. This distribution was chosen due to the personal material restriction of only two computers, and the number of total clients at each experiment was chosen so that, even if the potential clients work as they have low resources, it can be used as a test of a real world scenario. The different Operative Systems were chosen to test that intercommunication between Windows and Linux worked correctly.

Each client was spawned as a python process using the Python interface of the framework and the Pyro library to perform the optimisation steps. The server was be spawned as a rust binary using the general Rust crate of the framework. The code of both the server and the clients used in these experiments and the results obtained can be found at the general code repository in Appendix (A). All the certificates used by the clients and the server have been signed by a Certificate Authority explicitly created for these experiments.

For each experiment, each of the three training modes for Partitioned Variational Inference was tested. Regarding the optimisation step at each client, to generate a new approximate posterior 1000 gradient steps were taken and for each one 10 samples were taken to approximate the gradient.

The main metric being analysed in these experiments is the evolution of the total loss, as it represents the communication efficiency of the training (if it converges faster with less messages it is more efficient) and also shows possible fluctuations during training. The SVI (Stochastic Variational Inference) steps correspond to each clients' full optimisation step, not one of the 1000 ones they do internally. For each updated likelihood that a clients sends back to the server, an SVI step is taken.

Metrics like temporal efficiency, will also be taken into account but not from a numeric standpoint, they will be analysed from a relative point of view. This is due to the fact that with the limited resources, the speeds are massively volatile with so many clients running on one same machine.

4.1 Proof of Concept

In this experiment, we will consider a generated dataset consisting on 10 000 samples of unidimensional data generated from a single Gaussian with mean 5 and standard deviation 1. These data will be split among 10 clients, 5 on each computer. The goal of the training is to find the closest Gaussian that accurately depicts the distribution of the mean used to generate this data, starting from a standard Normal prior. This is illustrated in Figure(4.1).

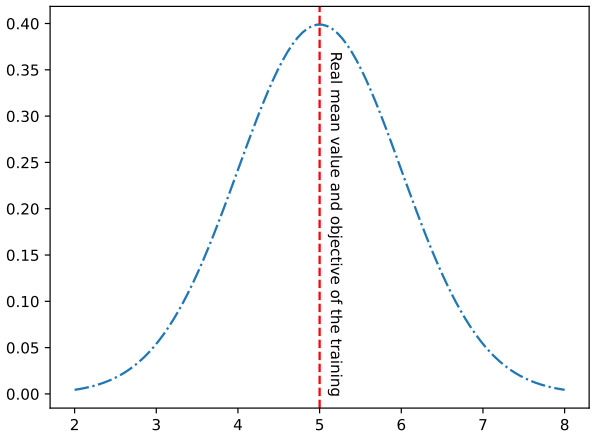


Figure 4.1: Illustration of the objective of the Proof of Concept training. The generated data is distributed according to the possibility density functioned represented by the blue line. The red dashed line represents the objective of the training, the mean of the Gaussian used to generate the data.

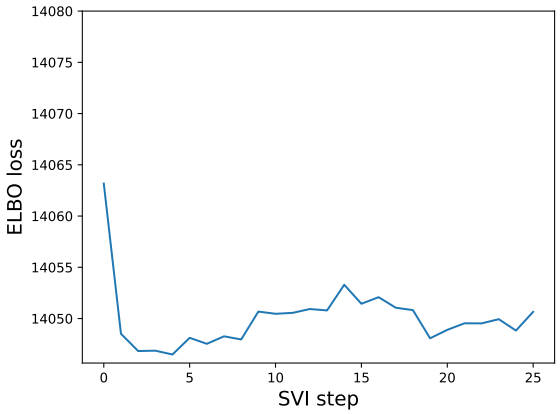
This experiment should be very easy to solve for our algorithm. However, the main objective of this experiment is to check that the implementation works correctly, even across different Operative Systems. It will also serve to observe some characteristics that vary between the setups that could carry on to other more complex scenarios. Figure (4.2) contains the graphs showing the evolution of the loss on the different kinds of training and Table (4.1) contains the final values obtained for each type of training.

	Mean	**Variance**
Sequential	5.019	2.443e-6
Synchronous	5.038	3.312e-4
Asynchronous	4.967	4.480e-4

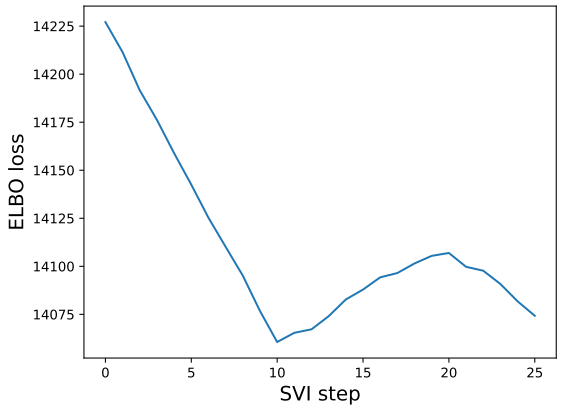
Table 4.1: Parameters of the final Gaussian posterior for the Proof of Concept experiment.

4.2 Bayesian Regression

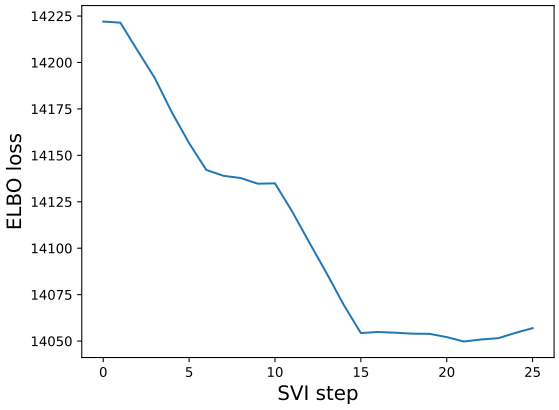
In this experiment, we will consider a dataset that contains geographic data about 234 countries, their real GDP per capita for the year 2000 and whether or not the given nation is in Africa, among others.



(a) Sequential training mode.



(b) Synchronous training mode



(c) Asynchronous training mode

Figure 4.2: Evolution of global loss by epochs in the Proof of Concept experiment. It is important to note that given the simplicity of the problem, even if the graph seems rough and that it lacks convergence, if one takes note of the dimensions of the y axis, it is practically not changing at all. The problem is so simple that it is almost solved from the initial prior.

These seemingly specific data are relevant as these data were procured from [5], where it was noted that terrain ruggedness is correlated with bad economic performance outside of Africa, but that it has the contrary effect for African countries.

The data were distributed among 3 clients on the Linux computer due to the low amount of data, and to test the circumstances of a low amount of clients with important data that needs to be private.

With these data, Bayesian linear regression will be performed with a predictor $\beta \boldsymbol{X} + \alpha$, where β and α are the regression coefficients. To achieve that, the distribution of the latent variables has been modelled as a Gaussian with diagonal covariance, which assumes there is no correlation among the latent variables. This is commonly known as a mean-field approximation. A standard diagonal Gaussian has been used as the prior.

In this form of regression, our objective will be to find the arguments α, β_1, β_2 and β_3 so that the mean calculated as in equation(4.1) offers the best general fit.

$$mean = \alpha + \beta_1 \cdot is\,african + \beta_2 \cdot ruggedness + \beta_3 \cdot is\,african \cdot ruggedness \tag{4.1}$$

This experiment was done with mainly three testing objectives: using a real dataset, training in an environment with low data per client and using more complex multidimensional distributions. Figure (4.3) contains graphs showing the evolution of the loss on the different kinds of training and Table (4.2) contains the final values obtained for each type of training.

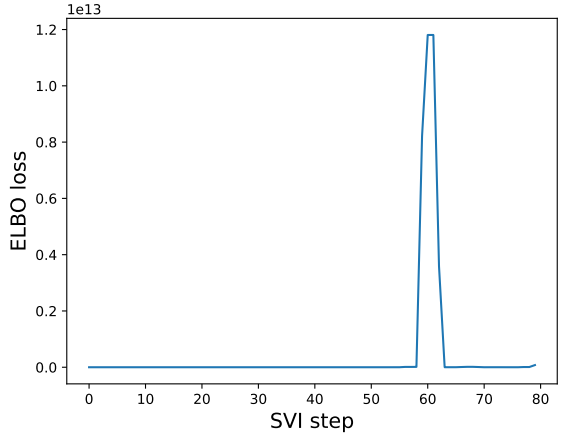
	α **Mean**	α **Scale**	β_1 **Mean**	β_1 **Scale**	β_2 **Mean**	β_2 **Scale**	β_3 **Mean**	β_3 **Scale**
Sequential	9.693	8.358e-05	-3.431	1.644e-4	-0.425	3.961e-05	0.906	4.974e-05
Synchronous	9.236	2.658e-6	-1.944	3.806e-6	-0.211	5.411e-7	0.388	2.283e-6
Asynchronous	9.236	1.388e-6	-1.968	5.135e-6	-0.208	3.402e-7	0.407	1.527e-6

Table 4.2: Parameters of the final Multivariate Normal posterior for the Bayesian Regression experiment.

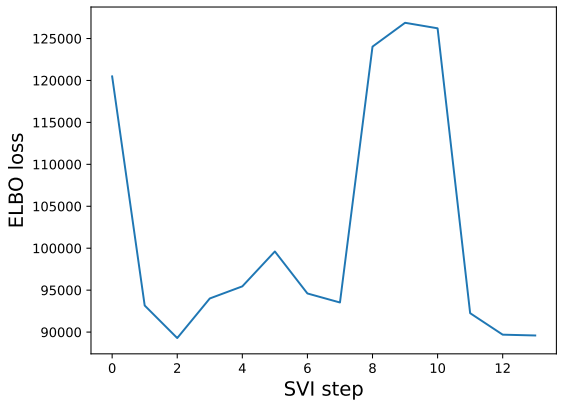
4.3 Experiments' Results

The main conclusions from the results obtained are:

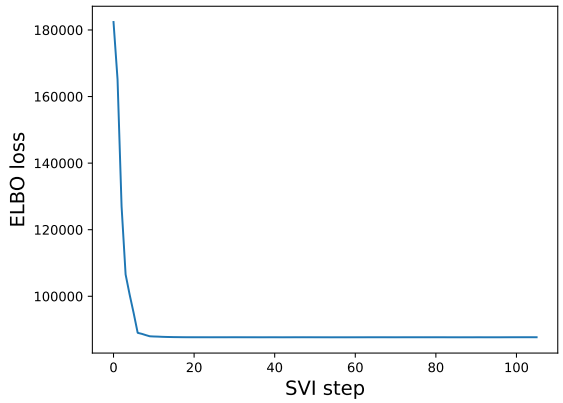
- Temporal Efficiency: Sequential mode is several times slower than their Synchronous and Asynchronous counterparts. In the Bayesian Regression training, the Sequential mode could take 40 or 50 minutes to train while the others would be done in 15 minutes. This makes sense due to the fact that the other modes can parallelise work while Sequential cannot. Between Synchronous and Asynchronous mode, Asynchronous is generally a bit faster to progress, but takes longer to



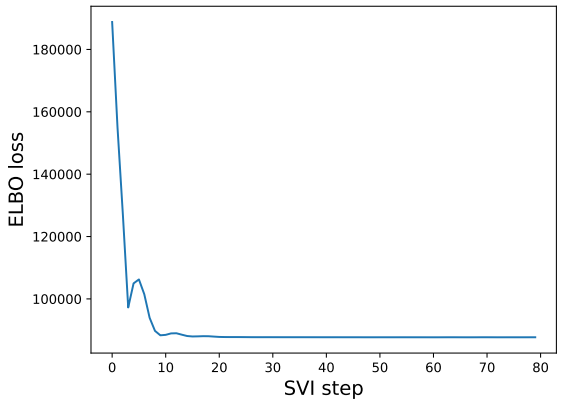
(a) Sequential training mode, without zooming.



(b) Sequential training mode, zooming on the zone of minimum loss, at the first iterations. This phenomenon is explained more in depth in Section (4.3)

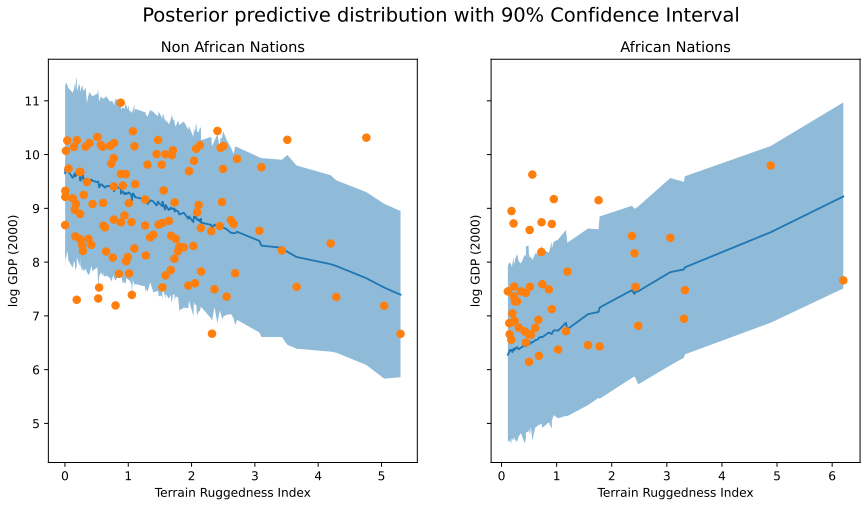


(c) Synchronous training mode.

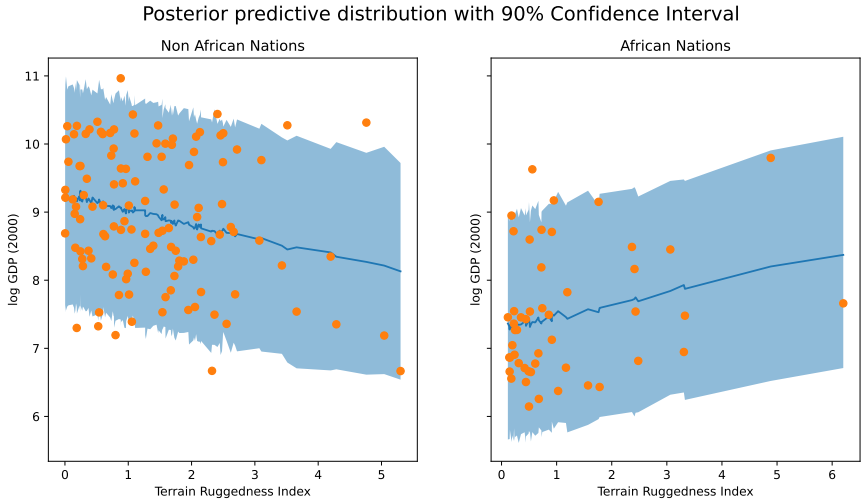


(d) Asynchronous training mode.

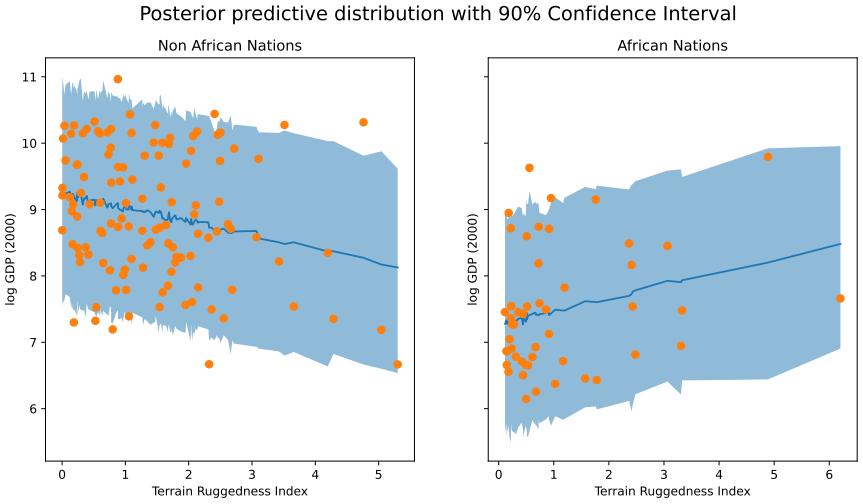
Figure 4.3: Evolution of global loss by epochs in the Bayesian Regression experiment.



(a) Sequential training mode.



(b) Synchronous training mode.



(c) Asynchronous training mode.

Figure 4.4: 90% Confidence Interval from the posterior obtained in the Bayesian Regression experiment alongside the original data.

converge so it is not as clear a decision, greatly depends.

- Message Efficiency: Sequential mode wins in Message Efficiency due to no work being wasted, each optimisation step has the full knowledge of the previous one. It is followed by Sequential mode which although the clients do not have the full knowledge of the likelihoods of the other clients on the same iteration, they have the ones of the last one. Asynchronous is the worst because due to the chaotic nature of the order of messages, lots of times there are serious fluctuations due to desynchronised likelihoods that severely slow down the training process. It is also important to note that the selection of the damping factor was shown to be quite relevant, with values too small or too big completely hindering the training. The same can be said for the number of samples for a SVI step and the number of steps at each client's optimisation.
- Handling of special cases: the observant reader can see that I have mentioned that the Sequential mode is the best at message efficiency but for example in Subfigure (4.3(a)), there is a great fluctuation. This is not due to the mode of training but due to the models chosen. Gaussians cannot have negative variances so some transformation has to be done when one is generated due to the algorithm to be able to make the optimisation step. Usually this transformation severely slows the training due to a sudden change in the factors. This happens on the Sequential mode mainly due to the fact of being so precise that it reaches values very close to zero faster, which are very susceptible to becoming negative. However, this is a trait of the Gaussian family and the implementation used, other distributions or implementations may not suffer the same problems.
- Precision of posterior distributions: even if the modes differed on the different kind of efficiencies, all modes were able to obtain precise results, with more or less the same precision.

5 Conclusions and Future Work

5.1 Conclusions

Bayesian Learning and Federated Learning are two great assets in the toolbox of a Machine Learning specialist working in the modern landscape of data generation, processing and analysis, where privacy, security and certainty are in great demand.

The benefits that each one provides complement each other quite well and could be used in tandem to increase the quality of models in fields like healthcare, where advances are directly translated to major material results, like early detection of diseases or enhancement of medical tools.

However, some of the requisites of these kinds of training are incompatible and, are not able to be used together. Partitioned Variational Inference offers a solution to that, modifying the standard Variational Inference algorithm so that it can be used in a Federated setting.

This opens up the gates to a new set of training possibilities, where lots of different actors with private or confidential data, but a common shared goal, can work together towards that goal in a Bayesian setting, where uncertainty can be accurately measured for data whose misanalysis could result in grave consequences.

The design of the system created to implement Partitioned Variational Inference took into account the different characteristics of the network architecture (with the heterogeneous and low performant nature of the clients) and the usefulness of an extensible network protocol tailor made for it. The modularisation of, not only the code of the system, but also of the different libraries and interfaces, greatly contributed to the developer experience and useability of the framework.

Rust proved to be a great language for this kind of performant, multiplatform, safe and extensible code. Throughout the whole development of this framework not once was a memory-related error encountered, even in the development of a complex asynchronous multithreaded server implementation with encrypted serialisation and communication.

The actor model and message based communication chosen to implement the algorithm's logic also proved to be a great fit for the problem. It also opened up the possibility of easily implementing the

most error prone parts of the algorithm as an state machine to avoid any unwanted states caused by an unexpected series of events.

The ecosystem of Rust and Python's interoperability was found to work correctly, and to offer useable choices, albeit a bit underdeveloped. This made the experience when working on the interfaces complex, not only due to some rough edges on the interoperability libraries, but also due to the inherent differences in methodologies and type systems of Rust and Python.

However, with some work to iron out the problems and create a Python interface, some Machine Learning libraries available in Python were able to be used with the framework to be able to work with complex problems.

The result obtained was an efficient, multiplatform, extensible and easy to use framework that makes possible training a Bayesian model using Variational Inference in a Federated setting.

The framework and algorithm were proved to be able to be used in a real setting with the experiments realised. These experiments not only proved that the framework and algorithm were able to achieve their goals, they also showed some characteristics of the different Partitioned Variational Inference modes. Providing valuable information about which one to use depending on the objective desired and the restrictions present.

With the development of this framework, and many possible others that implement these kind of training, the landscape of modern Machine Learning could greatly change and bring forth a new wave of innovations.

After developing this framework, I have come to know the current situation of the Probabilistic and Federated Learning ecosystems, specially in Python, and I found them to be quite underdeveloped in regards to documentation, accesibility and flexibility.

They are quite powerful, but are highly specialised and not developer friendly. In the coming years, I hope that more attention and care is put into these fields as I think they have great potential and would bring countless new possibilities to the Machine Learning ecosystem.

My framework was but my contribution to this ideal, but as years come, I expect that many such libraries will appear and contribute to the development of the field.

5.2 Future Work

The current framework could be expanded in a variety of ways, but the most beneficial in my opinion regarding this research would be:

- Asynchronous capabilities in the Python module: it was decided to create a synchronous interface

with the objective of offering a simple way to interface with the system. However, further work could be done in regards of thinking how to offer a simple, yet effective interface with asynchronous programming in mind.

- Exploring different network architectures: the main paper this implementation was based in [1], assumed the structure of a central server and N clients, but indicated that a peer 2 peer implementation could be possible. I agree with that statement, as long as the prior is known to all and the different likelihoods are distributed across the network, it could be possible to learn collaboratively without sharing the data. Exploring this kind of architecture could offer a possible collaborative learning algorithm that offers privacy and accurate uncertainty measures.

- Introducing part of the learning and optimisation parts of the algorithm inside the framework: the scientific calculation part of the Rust ecosystem is still not fully developed, but as it becomes more mature, an implementation of the optimisation algorithm could be added to the framework. This would not only benefit from the speed and security increases of using Rust, it would also mean that the framework would be self contained, without having to rely in any external factors in order to work.

- Further enhancing the general infrastructure of the project: different upgrades like a CI/CD pipeline, further logging and testing could be added. Even introducing remote traces could be very helpful for future users of the framework and greatly increase the debugging capabilities of the framework.

- Exploring different learning schedules or algorithms to reduce the negative effects of indiscriminate sequential, synchronous or asynchronous modes: maybe with further research a way can be found that groups most of the beneficial aspects of these different kinds of learning without most of the detrimental aspects.

- Offering interfaces to other languages: instead of supporting just Python, different libraries for the framework could be created for other languages. This would greatly expand the potential user base and would greatly align with the design philosophy of the framework based on clients' heterogeneity.

- Testing more complex models: due to the material constraints, experiments with large amounts of data or with models that require great amount of computational resources were not able to be made. However, in the future some interesting models like Variational Autoencoders or Bayesian Neural Networks could serve as a basis for more experiments.

Bibliography

[1] M. Ashman, T. D. Bui, C. V. Nguyen, S. Markou, A. Weller, S. Swaroop, and R. E. Turner, "Partitioned variational inference: A framework for probabilistic federated learning," 2022.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] C. Zhang, J. Butepage, H. Kjellstrom, and S. Mandt, "Advances in variational inference," 2017.

[4] M. Wainwright and M. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends in Machine Learning*, vol. 1, pp. 1–305, 01 2008.

[5] N. Nunn and D. Puga, "Ruggedness: The Blessing of Bad Geography in Africa," *The Review of Economics and Statistics*, vol. 94, pp. 20–36, February 2012.

[6] X. Liu, T. Shi, C. Xie, Q. Li, K. Hu, H. Kim, X. Xu, T.-A. Vu-Le, Z. Huang, A. Nourian, B. Li, and D. Song, "Unifed: All-in-one federated learning platform to unify open-source frameworks," 2023.

[7] C. Zhang, J. Butepage, H. Kjellstrom, and S. Mandt, "Advances in variational inference," 2017.

[8] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," 2016.

[9] A. Ganguly and S. W. F. Earp, "An introduction to variational inference," 2021.

[10] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278 – 2324, 12 1998.

[11] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," 2023.

Appendix

A

Code repository

For further detail of the code and procedures followed, the code used for the development of the framework and the one this thesis is based upon can be found in the following GitHub repository (Repo).

In the repository the following elements can be found:

- The source code of both the general Rust crate and the Python interface.
- The code used for the distributions in the experiments.
- Examples of both a server and a client.
- The certificates used in the experiments.
- The raw data obtained in the experiments.


UÁM
Universidad Autónoma
de Madrid