Phase-Type Distributions
Simulation and Estimation with Machine Learning

Simon Antoine Savine, s204798

Denmark Technical University, Bachelor Thesis, December 2024

Abstract

Phase-Type (PH) distributions are a versatile family of one-dimensional probability distributions, which includes familiar distributions such as the exponential or the Erlang distribution. Since a mixture of PH is itself a PH, it also includes mixtures of exponentials or mixtures of Erlangs, or mixtures of both. PH distributions offer a common representation for all these distributions, and many more. In fact, PH distributions are dense in the set of positive-valued scalar distributions. This means that any such distribution is approximated to arbitrary precision by a PH distribution, in the same sense that polynomials approximate all smooth functions by Taylor expansion. PH distributions are also intimately linked to Markov chains, which model many phenomena in natural or social sciences, such as the sequence of states of an excited atom, or the credit rating of a company.

In return for expressiveness, the analytic manipulation of PH distributions is nontrivial, simulation is computationally heavy, and estimation is very hard. This thesis focuses on the simulation and the estimation of PH distributions. There is a fair amount of existing literature on estimation of PH distributions, generally by approximation of the maximum likelihood estimator (MLE) with the expectation-maximization (EM) algorithm. We take a different and relatively new approach, where we attempt to estimate PH parameters from samples with end-to-end machine learning (ML). The ML approach could eventually provide a unified estimation framework, applicable to arbitrary distributions, even when the computation of the MLE is hard or untractable. This thesis does not reach that far, but it does provide a very promising proof of concept.

Estimation by ML is a rather recent and very different approach to the estimation problem in Statistics. To the best of our knowledge, it has never been applied to PH distributions.

We develop a special artificial neural network (ANN) architecture, based on so-called Deep Sets, for end-to-end estimation of parameters from samples of varying size, and evaluate performance by comparison with established methods based on MLE. Our numerical results are limited to some special PH distributions for now: exponential, Erlang and mixtures of exponentials. Performance is solid, roughly on par with MLE, so the new field of ML estimation appears very promising, and deserving further research to improve ANN performance and extend to more general PH distributions.

Contents

Introduction

This thesis is arranged in two parts. The first part introduces PH distributions and continuous-time Markov Chains (CTMC). It is mainly theoretical and gathers material from various textbooks and articles, such as [2], [3], [4], [6] or [8]. This is by no means a comprehensive exploration of CTMC or PH distributions and their properties, but rather a report of a Bachelor student's journey to understand those things and report them in a consistent and (hopefully) digestible manner, thereby facilitating the understanding of others.

Another purpose of the first part is the introduction of all the moving pieces: mathematical bases, definitions, properties and theorems, used in the second part about estimation in general, and with Machine Learning (EML[1]) in particular. Simulation is a critical cog in the EML machinery, and we cover it comprehensively in its own chapter. The properties of CTMC and PH covered in the first part are those most useful for simulation and estimation. In particular, we introduce some special kinds of PH distributions: exponential, Erlang and mixture of exponentials, which are the main subjects of the experiments in the second part.

Phase-type (PH) distributions are specified by a dimension n, probability vector P_0 of size n and intensity matrix Q of shape n by n, and defined by the probability density function (pdf):

$$f(x) = \alpha_n^T e^{Qx} Q P_0$$

where α_n is the vector of size n with 1 in its nth entry and zero everywhere else. This just means that the pdf is given by the nth entry of the vector $e^{Qx}QP_0$.

We will refine this sketch of definition and specify the necessary constraints on P_0 and Q. For now, we see that this definition deserves some explanation: what does it mean? What is this e^{Qx} that looks like the exponential of a matrix? Where does this pdf come from? And how exactly does it represent such a vast set of familiar and less familiar distributions?

We will attempt to answer all these questions in the next few chapters, starting with the necessary mathematical bases, such as matrix exponentials. We will also introduce Markov chains, because they are intimately related to PH distributions. Every PH distribution has an associated Markov chain (but, as we will see, not every Markov chain defines a valid PH distribution). The Markov chain representation greatly facilitates theoretical and practical manipulation of PH distributions. For instance, we are going to see that an Erlang distribution is specified by an integer shape k and an intensity λ, and defined by the pdf:

$$f(x) = \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}$$

It is not immediately apparent that this pdf is a special case of the PH pdf, nor is it evident to find the PH parameters P_0 and Q such that the PH coincides with the Erlang. We will see that this exercise becomes almost trivial when reasoning about the associated Markov chains.

Similarly, there is (to the best of our knowledge) no easy means of simulating PH distributions, so the most common solution is to simulate a sample path for the associated Markov chain.

The second part is much more exploratory and covers the relatively new idea of end-to-end estimation by Machine Learning, first introduced (to the best of our knowledge) in the 2023 paper [7]. In a stark departure from traditional approaches to estimation, mainly focused on maximum likelihood estimators and related algorithms, such as expectation-maximization (EM), the authors proposed to train artificial neural networks (ANN) to directly learn an estimation function from samples of independent realizations of some random variable, to the estimated parameters of the variable's distribution. The ANN is trained with labels that are true generative parameters, and inputs of random samples drawn for the distribution with these parameters.

What is remarkable is that (as we will see) the ANN achieves a performance (measured, for example, as the mean-squared error or MSE on out-of-sample data) similar to MLE, without any analytic work on such or such distribution. The ANN doesn't know anything about the distribution, its pdf or its MLE. It only 'sees' samples labeled by generative parameters, and manages to 're-derive' a function with estimation performance similar to MLE.

[1] We say EML for estimation by Machine Learning, and MLE for maximum likelihood estimator

It takes hard work to achieve good performance, in particular, overcome bias. This thesis merely constitutes a proof of concept, albeit a very promising one. We applied EML systematically on some families of PH distributions: exponential, Erlang and mixtures of exponentials (as well as Gaussian distributions, which are not PH distributions, for a first proof of concept). We tried to leave things unchanged when working with different distributions, but, for now, the training loop still must be fine-tuned on a distribution by distribution basis.

The main achievement of this thesis is the development of a special ANN architecture for estimation, inspired by the Deep Sets introduced in [9]. As a result, our neural estimators are permutation-invariant, and also, accept samples of arbitrary size, something very difficult to achieve with neural networks. Our estimators also behave as expected when the sample size grows, with unchanged bias and standard error decreasing to zero.

There is however much room for improvement: many details remain to be fine-tuned for performance, the inclusion of new distributions needs to be automated, and some questions remain unanswered, such as: why does the ML model struggle so much with small rate parameters for estimation of the gamma and Erlang distributions? It is hoped that the promising results may inspire further research in this exciting new topic.

The Python code is available on the github repo `https://github.com/simonsavine/phasetype/` and also in the body of the document where helpful. The notebooks were developed on Google colab. The EML notebooks perform heavy computations, best executed on the A100 GPU. 40GB of CPU and GPU RAM are required.

Part I

Markov chains and PH distributions

Chapter 1

Mathematical bases

This chapter introduces some mathematical prerequisites for the study of phase-type (PH) distributions: matrix exponentials and Markov chains.

Matrix exponentials are mathematical constructs, explicitly included in the expression for the probability density function (pdf) of Phase-Type distributions. PH distributions are also the probability distributions of the absorption times of continuous-time Markov chains (CTMC). CTMCs are introduced in the next chapter as an extension of (discrete-time) Markov chains.

We therefore introduce these two prerequisites: matrix exponentials and Markov chains, in the first chapter, in order to facilitate reading the rest of the document.

1.1 Matrix Exponentials

A Phase-Type distribution with n states is a scalar, continuous probability distribution over non-negative real numbers, parameterized by a vector of probabilities $P_0 \in \mathbb{R}^n$ and a matrix of intensities $Q \in \mathbb{R}^{n \times n}$. A PH distribution is specified by the pdf:

$$f(t) = \alpha_n^T e^{Qt} Q P_0$$

where $\alpha_n = (0, 0, ..., 0, 1)$. The next few chapters will clarify why the pdf has this form and where it comes from.

For now, we focus on the term e^{Qt}. Since Q is a (square) matrix and t is a number, Qt is a matrix of the same shape as Q. e^{Qt} is therefore the exponential of a matrix. Matrix exponentials of the form e^A have a precise meaning in mathematics, which is is not the same as the matrix of the exponentials of A's coefficients. Here, we formally define matrix exponentials and introduce some of their properties, used in the rest of the document.

1.1.1 Definition

For a real variable x, one definition of the exponential e^x is:

$$e^x = \sum_{m=0}^{\infty} \frac{x^m}{m!}$$

Matrix exponentials are defined in a similar manner.

Definition (Matrix exponential)**.** *For a square matrix A, the matrix exponential e^A is defined by the expression:*

$$e^A = \sum_{m=0}^{\infty} \frac{A^m}{m!}$$

where A^m means matrix product of A by itself, m times.

Note that this definition only makes sense for square matrices, and includes the scalar definition for square matrices of size 1. In addition, it immediately follows that e^A commutes with A: $Ae^A = e^A A$.

Matrix exponentials inherit many properties of scalar exponentials. For example, $lim_{n\to\infty}(I_n + A/n)^n = e^A$. On the other hand, as usual with matrices, one must be wary of commutation: $e^A e^B = e^{A+B}$ only if A and B commute!

Matrix exponentials also differentiate similarly to scalar exponentials, as we will see, after we reviewed the special, but important, case of a diagonalizable matrix.

1.1.2 Exponential of a diagonalizable matrix

The definition of a matrix exponential as an infinite sum is abstract and not particularly helpful for the actual computation of the exponential of a given matrix. In practice, the exponential is generally correctly approximated by the few first terms of the sum, due to extremely fast growth of the factorial: $20! \approx 2e + 18!$ But one must still compute multiple expensive matrix products, and estimate the number of terms necessary for a required accuracy.

However, when A is diagonalizable, its exponential can be computed in a much simpler and more efficient manner. Recall that a square n by n matrix A is diagonalizable if there exists a diagonal matrix Λ (which entries are the eigenvalues of A) and a square invertible matrix V (which columns are the eigenvectors of A), both of shape n by n, such that:

$$A = V\Lambda V^{-1}$$

In this case, the following propositions provides an efficient computation.

Proposition (Exponential of a diagonalizable matrix)**.** *If $A = V\Lambda V^{-1}$ is diagonalizable, then:*

$$e^A = Ve^{\Lambda}V^{-1} = V(e^{\lambda_i})V^{-1}$$

where (e^{λ_i}) is the diagonal matrix of eigenvalues exponentials.

Proof. Note that:

$$\begin{aligned} A^2 &= \left(V\Lambda V^{-1}\right)^2 \\ &= V\Lambda \underbrace{V^{-1} \cdot V}_{\text{Identity matrix}} \Lambda V^{-1} \\ &= V\Lambda\Lambda V^{-1} = V\Lambda^2 V^{-1} \end{aligned}$$

and, by induction: $A^m = V\Lambda^m V^{-1}$.

Substitute in the definition:

$$e^A = \sum_{m=0}^{\infty} \frac{A^m}{m!} = \sum_{m=0}^{\infty} \frac{V\Lambda^m V^{-1}}{m!} = V \sum_{m=0}^{\infty} \frac{\Lambda^m}{m!} V^{-1} = Ve^{\Lambda}V^{-1}$$

Furthermore, since Λ is diagonal, $\Lambda^m = (\lambda_i^m)$. Hence:

$$e^{\Lambda} = \sum_{m=0}^{\infty} \frac{\Lambda^m}{m!} = \sum_{m=0}^{\infty} \frac{(\lambda_i^m)}{m!} = (\sum_{m=0}^{\infty} \frac{\lambda_i^m}{m!}) = (e^{\lambda_i})$$

□

The computation of matrix exponentials, and therefore, CTMC and PH distributions, is much easier and more efficient with diagonalizable matrices. In practice, we use scipy's function expm(), which is correct in all cases, but less efficient than a dedicated computation for diagonalizable matrices. This function is tested in the notebook https://github.com/simonsavine/phasetype/tree/main/matrix_exponential.ipynb.

1.1.3 Derivative of matrix exponentials

We don't need to introduce all the properties of matrix exponentials to understand CTMC and PH distributions. But we will need to differentiate expressions of the form e^{Ax} (where x is a number), or find the solution of the initial value problem (IVP) $v'(x) = Av(x)$ when v is a vector function of x in dimension n and $v(0)$ is given.

In the scalar case, we know that $de^{ax}/dx = ae^{ax}$ and that the solution of the IVP $f'(x) = ax$ is $f(x) = f(0)e^{ax}$.

The following proposition shows that those results carry over naturally to matrix exponentials.

Proposition (Derivative of Matrix Exponential)**.**

$$\frac{de^{Ax}}{dx} = Ae^{Ax} = e^{Ax}A$$

Proof.

$$\begin{aligned}
\frac{de^{Ax}}{dx} &= \frac{d}{dx} \sum_{m=0}^{\infty} \frac{(Ax)^m}{m!} \\
&= \frac{dI}{dx} + \frac{d}{dx} \sum_{m=1}^{\infty} \frac{(Ax)^m}{m!} \\
&= \frac{d}{dx} \sum_{m=1}^{\infty} \frac{A^m}{m!} x^m \\
&= \sum_{m=1}^{\infty} \frac{A^m}{m!} m x^{m-1} \\
&= \sum_{m=1}^{\infty} \frac{A^m x^{m-1}}{(m-1)!} \\
&= A \sum_{m=1}^{\infty} \frac{A^{m-1} x^{m-1}}{(m-1)!} \\
&= A \sum_{m=0}^{\infty} \frac{A^m x^m}{m!} \\
&= Ae^{Ax}
\end{aligned}$$

□

This is particularly helpful due to the following corollary:

Corollary. *The solution of the vector differential equation:*

$$\frac{dv(x)}{dx} = Av(x)$$

is:

$$v(x) = v(0)e^{Ax}$$

1.1.4 Smooth functions of matrices

As a closing comment on matrix exponentials, all of the above generalizes to smooth functions other than exponentials. If f is a smooth scalar function and A is a square matrix, we can define and manipulate expressions of the type $f(A)$ following what we did for exponentials. In particular, we define $f(A)$ by its Mclaurin series:

$$f(A) = \sum_{m=0}^{\infty} \frac{f^{(m)}(0)A^m}{m!}$$

If $A = V\Lambda V^{-1}$ is diagonalizable, then:

$$f(A) = Vf(\Lambda)V^{-1} = V[f(\lambda_i)]V^{-1}$$

Further,

$$\frac{df(Ax)}{dx} = Af'(Ax) = f'(Ax)A$$

1.2 Discrete Time Markov Chains

1.2.1 Introduction

Phase-Type distributions are distributions of the absorption time of continuous-time Markov chains. Before we introduce these notions in the next chapters, we introduce the simpler and more familiar discrete-time Markov chains here.

A Markov chain is a particular case of a state-space process, which models a system that randomly transitions between a finite number n of states. For example, an excited atom transitions between quantized energy states before eventually reaching its ground state, where it remains thereafter in the absence of external interference. As another example, the credit rating of a firm consists of a grade among a dozen possibilities, AAA (the highest credit rating), AA+, AA, AA-, A+, A, A-, BBB+, BBB, etc. all the way down to C (the lowest credit rating for a firm in activity) and D when the firm files for bankruptcy. As the company conducts business, its credit rating transitions between those grades at regular intervals, when rating companies publish their current verdict. When a firm defaults, it transitions to state D, where it remains forever. The ground state of the atom, or the default state of a firm, are called absorbing states.

State-space processes provide a mathematical formalism for the study of such systems. They are well studied due to the large number of applications in many fields.

Definition (Discrete time, discrete state-space process)**.** *A discrete time, discrete state-space process* (X_i) *is a process observed on a set of discrete times* T_i *for* $i = 1, 2...$ *in one of* n *possible states. We denote* $X_i = j$ *to mean that the process is in state* j *at time* T_i*.*

A state-space process is called Markov chain when it is memory-less, modeling the idea that "the future only depends on the present, not the past". In other terms, the probabilities of future states only depend on the current state, irrespective of how we got there. More formally:

Definition (Markov chain)**.** *The state process* (X_i) *is a Markov chain if the state probabilities for time* T_{i+1} *conditional to the state history up to time* T_i *only depend on the state at time* T_i*:*

$$Pr(X_{i+1} = j|X_i, X_{i-1}, ...) = Pr(X_{i+1} = j|X_i)$$

All those transition probabilities $r_{jk}^{(i)} = Pr(X_{i+1} = j|X_i = k)$ can be collected in a n by n matrix $R_i = (r_{jk}^{(i)})$ that completely specifies the behavior of the Markov chain between times T_i and T_{i+1}. In principle, transition probabilities may be different on different time steps i. When those transition probabilities are fixed: $R_i = R$ for all i, the Markov chain is called time-homogeneous.

Definition (Time-homogeneous Markov chain)**.** *A Markov chain is time-homogeneous if its transition probabilities* $r_{jk}^{(i)} = Pr(X_{i+1} = j|X_i = k)$ *do not depend on time* i*:* $r_{jk}^{(i)} = r_{jk}$ *for all* i*. The matrix* $R = (r_{jk})$ *is called the transition matrix of the Markov chain.*

Time-homogeneous Markov chains are the most common types of Markov chains, and the only kind considered in this document. All Markov chains are implicitly homogeneous in all that follows.

1.2.2 State probabilities

A Markov chain is entirely specified by its transition matrix R, and the n-dimensional vector P_0 that specifies the distribution of the initial state at time T_0.

When the chain starts in some definite state k, $P_0 = \alpha_k$, a vector with kth entry 1 and zeroes everywhere else.

More generally, we denote P_i the n-dimensional vector of state probabilities at time T_i. State probabilities are governed by following theorem.

Theorem (State probabilities of Markov chains)**.** *The state probabilities of a Markov chain with transition matrix* R *satisfy the equation:*

$$P_{i+1} = RP_i$$

And by induction, for all time steps $i_2 \geq i_1$*:*

$$P_{i_2} = R^{i_2 - i_1} P_{i_1}$$

In particular:

$$P_i = R^i P_0$$

Proof. By the law of total probabilities:

$$Pr(X_{i+1} = j) = \sum_{k=1}^{n} Pr(X_{i+1} = j | X_i = k) Pr(X_i = k)$$
$$P_{i+1,j} = \sum_{k=1}^{n} r_{jk} P_{i,k}$$
$$P_{i+1} = RP_i$$

□

1.2.3 Stochastic matrices

The following corollary of the previous theorem immediately provides a simulation algorithm for sampling the paths of a Markov chain.

Corollary. *The state probabilities at time T_{i2} conditional to being in the definite state k at time $T_{i1} \leq T_{i2}$ are:*

$$Pr(X_{i_2} | X_{i_1} = k) = R^{i_2 - i_1} \alpha_k$$

which is the kth column of $R^{i_2 - i_1}$. In particular:

$$Pr(X_{i+1} | X_i = k) = R\alpha_k$$

which is the kth column of R.

This last equation shows that the kth column of R contains the probabilities of jumping into states $1, 2, ..., n$ from state k over one time step, including the probability of staying in state k in the kth entry.

The off-diagonal terms $r_{j \neq k}$ of R are probabilities of jumping from state k to state j over one time step. Its diagonal terms R_{kk} are probabilities of staying in state k.

It follows that the entries of R are non-negative, and its columns must sum to 1. A square matrix with those properties is called stochastic matrix. Stochastic matrices have interesting properties. For example, if R is stochastic, R^p is also stochastic, since it is the transition matrix over p time steps.

Another useful property is that the eigenvalues of stochastic matrices are always real. But stochastic matrices are not always diagonalizable. When the transition matrix of a Markov chain is diagonalizable, its state probabilities are computed much more efficiently.

Remark (Diagonalizable case)**.** *If $R = V\Lambda V^{-1}$ is diagonalizable, then:*

$$R^i = V\Lambda^i V^{-1} = V(\lambda_j^i)V^{-1}$$

and:

$$P_{i_2} = V(\lambda_j^{i_2 - i_1})V^{-1} P_{i_1}$$

in particular:

$$P_i = V(\lambda_j^i)V^{-1} P_0$$

and if the Markov chain starts in the definite state k at time T_0:

$$P_i = V(\lambda_j^i)V^{-1}\alpha_k$$

In practice, we use numpy's optimized matrix_power() function.

1.2.4 Simulation of Markov chains.

The simulation of a Markov chain is performed with the immediate algorithm: initialize the initial state $k = X_0$, then iteratively draw the next state $k = X_{i+1}$ with conditional probabilities $Pr(X_{i+1}|X_i = k)$ in the kth column of R. The python code is available online, in the notebook markov_chain_simulation_discrete_time.ipynb of our github repo: https://github.com/simonsavine/phasetype/tree/main.

Figure 1.1 displays some numerical results. We simulated a Markov chain with 5 states and a fixed (diagonalizable) transition matrix R, starting in the first state at time T_0. We display theoretical state probabilities on time step 50, computed with equation $P_i = R^i \alpha_k$ (with $i = 50$ and $k = 1$), along with the empirical state frequencies observed on simulated sample paths. We see that empirical frequencies converge towards theoretical probabilities, as expected, when the number of sample paths grows.

Markov chains have many other interesting properties, particularly in terms of their asymptotic behavior when time step i grows to infinity. We do not investigate Markov chains in discrete time any further in this document. Our main purpose is the simulation and estimation of PH distributions, which relate to continuous-time Markov chains, and we only introduced the discrete-time variant to facilitate the introduction of their continuous-time extension.

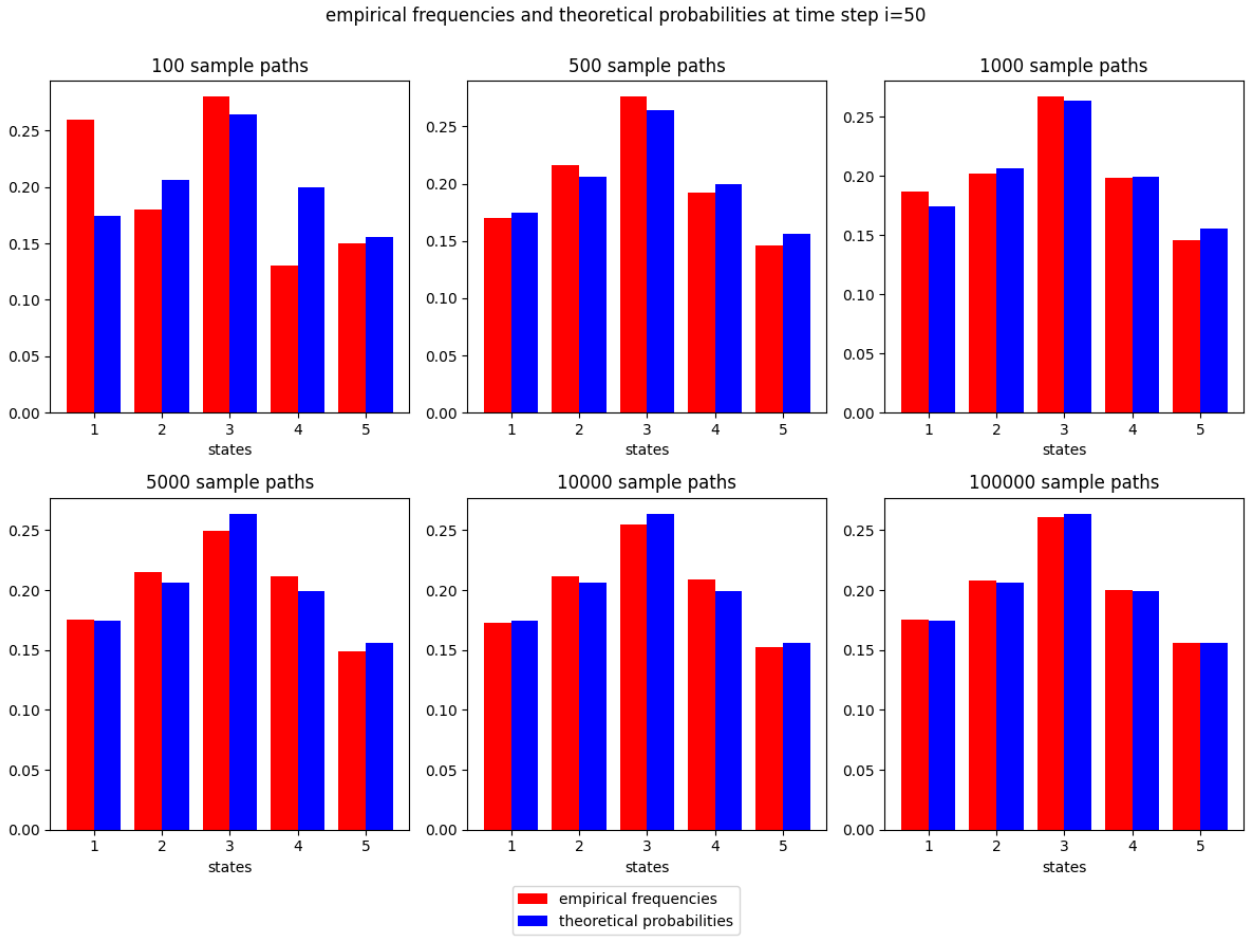

Figure 1.1: Numerical results for time step $i = 50$ from the simulation of a Markov chain with initial state $X_0 = 1$ and fixed transition matrix R. Theoretical state probabilities are $P_i = R^i \alpha_k$ (with $i = 50$ and $k = 1$). Empirical state frequencies are estimated on simulated sample paths. For information: $R = \begin{pmatrix} 0.284 & 0.103 & 0.167 & 0.0719 & 0.287 \\ 0.020 & 0.329 & 0.176 & 0.238 & 0.259 \\ 0.280 & 0.131 & 0.415 & 0.242 & 0.192 \\ 0.336 & 0.158 & 0.116 & 0.274 & 0.146 \\ 0.077 & 0.277 & 0.124 & 0.172 & 0.115 \end{pmatrix}$.

Chapter 2

Continuous time Markov chains and PH distributions

2.1 Continuous time Markov chains

2.1.1 Introduction and definition

We are primarily concerned with continuous-time Markov chains (CTMC), an extension of (discrete-time) Markov chains, where the state process X is not observed on discrete times $T_0, T_1, T_2, ...$ but continuously over a non-negative real timeline.

To build intuition, consider a discrete-time Markov chain with transition matrix R, observed on $m+1$ time steps over a time interval (T_a, T_b), such that $T_0 = T_a$, $T_{i+1} = T_i + h$ where h is the time interval: $h = (T_b - T_a)/m$, and $T_m = T_b$.

If the initial state at time T_a is k, we know that:

$$P_{T_b} = R^m \alpha_k$$

Further, recall that the off-diagonal entries $r_{j \neq k}$ of R are probabilities of jumping states over one time step. It is natural to scale them with the length h of a time step:

$$r_{j \neq k} = q_{jk} h$$

where q_{jk} does not depend on h or m. The columns of R must sum to 1. Hence, its diagonal entries r_{kk} must satisfy:

$$r_{kk} = 1 - \sum_{j \neq k} r_{jk} = 1 - \sum_{j \neq k} q_{jk} h$$

An example may help visualize what is going on. Consider a 3 by 3 transition matrix. We have:

$$
\begin{aligned}
R &= \begin{bmatrix} 1-(q_{21}+q_{31})h & q_{12}h & q_{13}h \\ q_{21}h & 1-(q_{12}+q_{32})h & q_{23}h \\ q_{31}h & q_{32}h & 1-(q_{13}+q_{23})h \end{bmatrix} \\
&= I_3 + \begin{bmatrix} -(q_{21}+q_{31}) & q_{12} & q_{13} \\ q_{21} & -(q_{12}+q_{32}) & q_{23} \\ q_{31} & q_{32} & -(q_{13}+q_{23}) \end{bmatrix} h \\
&= I_3 + Qh
\end{aligned}
$$

where I_3 is the identity matrix in dimension 3. The result:

$$R = I + Qh \iff Q = \frac{R-I}{h}$$

carries over to general n, where I is the identity matrix in dimension n, and Q is a square matrix independent of h or m, with columns summing to 0.

It follows from $r_{j\neq k} = q_{jk}h$ that the off-diagonal entries $q_{j\neq k}$ of Q are probabilities *per time unit*, also called *intensities*, of jumping from state k to state j. These intensities are non-negative.

Further, the diagonal entries of Q are $q_{kk} = (r_{kk}-1)/h$, or in other terms, $-q_{kk} = (1-r_{kk})/h$. Since r_{kk} is the probability of staying put in state k, $-q_{kk}$ is the intensity of of jumping out of state k (into whatever state other than k). Diagonal entries q_{kk} are therefore negative (or zero).

Q is called intensity matrix.

Furthermore, substituting $R = I + Qh$ into $P_{T_b} = R^m\alpha_k$, we get:

$$P_{T_b} = R^m\alpha_k = (I+Qh)^m\alpha_k = (I + Q\frac{T_b - T_a}{m})^m\alpha_k$$

Recall, once again, that Q does not depend on m. It follows that:

$$lim_{m\to\infty}P_{T_b} = lim_{m\to\infty}(I + Q\frac{T_b-T_a}{m}h)^m\alpha_k = e^{Q(T_b-T_a)}\alpha_k$$

As the number m of time steps between T_a and T_b grows to infinity, the discrete-time Markov chain converges to a continuous-time Markov chain on the interval (T_a, T_b) and the state probabilities at T_b, conditional to $X_{T_a} = k$, converge to $e^{Q(T_b-T_a)}\alpha_k$, the kth column of $e^{Q(T_b-T_a)}$.

This intuitive construction leads to the following formal definition.

Definition (Continuous-time Markov chain (CTMC))**.** *A continuous-time (homogeneous) Markov chain is a process (X_t) observed in one of n possible states at any (non-negative real) time t, with conditional state probabilities:*

$$Pr(X_{t_2}|X_{t_1} = k) = e^{Q(t_2-t_1)}\alpha_k$$

irrespective of the state path up to t_1, for all $t_2 \geq t_1$.

$Q = (q_{jk})$ is called the intensity matrix. Its columns sum to 0. Its off-diagonal entries $q_{j\neq k}$ are the intensities of jumping from state k to state j. Its diagonal entries q_{kk} are negative or zero, $-q_{kk}$ is the intensity of jumping out of state k.

As before, we denote P_t the n-dimensional vector of state probabilities at time t. In particular, P_0 contains the distribution of the initial state, which may or may not be definite. In case the process starts in a definite initial state k, $P_0 = \alpha_k$. P_0 and Q entirely specify a CTMC.

It follows from the definition of a CTMC and the law of total probabilities that the state probabilities verify:

$$P_{t_2} = e^{Q(t_2 - t_1)} P_{t_1}$$

In particular:

$$P_t = e^{Qt} P_0$$

As with discrete-time Markov chains, we don't investigate the many other properties of CTMC. This introduction is sufficient for our purpose: design a simulation algorithm for CTMC, and introduce PH distributions.

2.1.2 Holding times and exponential distributions

Distribution of holding times

One property of CTMCs that will prove very useful is the distribution of its holding times, defined as the time it spends in a given state, or more formally:

Definition (Holding time)**.** *Consider a CTMC with intensity matrix Q, in state k on a given time t. We call holding time and denote τ_k the amount of time spent in state k before jumping out of it:*

$$\tau_k = min\{s | X_{t+s} \neq k | X_t = k\}$$

It so happens that we can derive the distribution of holding times.

Proposition (Distribution of holding time)**.** *The cumulative probability density of the holding time τ_k is given by the expression:*

$$F_{\tau_k}(t) = Pr(\tau_k < t) = 1 - e^{q_{kk}t}$$

It follows that the pdf of τ_k is given by:

$$f_{\tau_k}(t) = F'_{\tau_k}(t) = -q_{kk} e^{q_{kk}t}$$

Sketch of proof. *Recall that the diagonal terms diagonal terms q_{kk} of Q are intensities (probabilities per time) of jumping out of state k. Hence:*

$$\begin{aligned}
Pr(\tau_k > t + dt) &= Pr(\tau_k > t + dt \text{ and } \tau_k > t) && \text{because } \tau_k > t + dt \rightarrow \tau_k > t \\
&= Pr(\tau_k > t + dt | \tau_k > t) Pr(\tau_k > t) \\
&= Pr(X_{t+dt} = k | X_t = k) Pr(\tau_k > t) \\
&= [1 - Pr(X_{t+dt} \neq k | X_{t+dt} = k | X_t = k)] Pr(\tau_k > t) \\
&= (1 + q_{kk} dt) Pr(\tau_k > t)
\end{aligned}$$

Since $Pr(X_{t+dt} \neq k | X_{t+dt} = k | X_t = k)]$ is the probability of jumping out within time dt, which we saw was equal to $-q_{kk}dt$, the intensity of jumping out of state k.

$$\frac{Pr(\tau_k > t + dt) - Pr(\tau_k > t)}{dt} = q_{kk} Pr(\tau_k > t)$$

$$\frac{\partial Pr(\tau_k > t)}{\partial t} = q_{kk} Pr(\tau_k > t)$$

Besides, $Pr(\tau_k > 0) = 1$*, hence the function* $g(t) = Pr(\tau_k > t)$ *satisfies the differential equation* $g'(t) = q_{kk} g(t)$ *with initial value* $g(0) = 1$*. The solution is:*

$$g(t) = e^{q_{kk} t} = Pr(\tau_k > t)$$

Hence:

$$Pr(\tau_k \leq t) = 1 - e^{q_{kk} t}$$

Exponential distribution

The distribution of the holding time is the well-known exponential distribution.

Definition (Exponential distribution)**.** *An exponential distribution with rate* λ *is a probability distribution over non-negative real numbers, with CPD:*

$$F(x) = 1 - e^{-\lambda x}$$

and pdf:

$$f(x) = \lambda e^{-\lambda x}$$

The expectation of a variable X with exponential distribution with rate λ is:

$$EX = \frac{1}{\lambda}$$

and its variance is:

$$EX = \frac{1}{\lambda^2}$$

Finally, one useful property for simulation is the inverse CPD:

$$y = F(x) \iff y = 1 - e^{-\lambda x} \iff e^{-\lambda x} = 1 - y \iff x = -\frac{\log(1 - y)}{\lambda} = F^{-1}(y)$$

2.1.3 Absorbing states

Recall the motivating examples in the introduction to Markov chains: an atom transitioning between energy levels until it reaches its ground state, and a firm transitioning between credit ratings for as long as it remains in business. In both cases, we have a special state where the system stops transitioning: the ground energy state and the bankruptcy state. The system cannot escape those states: once reached, the system remains in this state forever. The probability of transitioning into a different state is zero, the probability to remain in this state is one. Those states are called absorbing states.

Absorbing states appear in many applications, and it is often useful to measure how long it takes the system to reach the absorbing state. The time elapsed between $t = 0$ and when the system first hits the absorbing state, called absorption time, is a scalar, continuous, non-negative random variable, which realization depends on the realized state path. The distribution of the absorption time is so useful in many applications that scientists have developed and formalized a family of distributions, called phase-type (PH) distributions, for this purpose. A PH distribution, like its underlying CTMC, is parameterized by a vector of probabilities P_0 and an intensity matrix Q, and encodes the distribution of the absorption time of the CTMC.

The following definition formalises the notion of an absorbing state.

Definition (Absorbing state)**.** *The state k of a CTMC with intensity matrix Q is absorbing if the kth column of Q is 0.*

Recall that diagonal elements of Q are intensities to jump out of states, and off-diagonal elements $q_{j\neq k}$ are intensities of jumping from state k to state j. When column k of Q is 0, the process cannot jump out of state k and it cannot reach any other state. This formal definition therefore adequately captures the notion of an absorbing state.

In the rest of the document, we only work with CTMCs with exactly one absorbing state. Further, by convention, we assign it number n, the last state, so that the last column of Q is 0.

2.2 PH distributions

We formalize the definition of the absorption time:

Definition (Absorption time)**.** *Consider a CTMC (X_t) with one absorbing state n. Its absorption time, denoted τ, is the time when the CTMC first reaches the absorbing state:*

$$\tau = min\{t|X_t = n\}$$

As mentioned earlier, the probability distribution of the absorption time τ is of primary interest in this document, and for many other applications. The following theorem expresses its cumulative probability distribution (CPD) and probability density function (pdf).

Theorem (Distribution of absorption time)**.** *Consider a CTMC (X_t) with n states, initial probabilities $P0$ and intensity Q such that the nth state is absorbing, i.e. the last column of Q is zero.*

The CPD of its absorption time τ is:

$$F_\tau(t) = \alpha_n^T e^{Qt} P_0$$

where $\alpha_n^T = (0, 0, ..., 0, 1)$ and its probability density function (pdf) is:

$$f_\tau(t) = \alpha_n^T e^{Qt} Q P_0$$

Proof. The probability of reaching absorbing state before t is the probability of being in the absorbing state at t, which is the nth entry of P_t:

$$F_\tau(t) = Pr(\tau < t) = Pr(X_t = n) = \alpha_n^T P_t = \alpha_n^T e^{Qt} P_0$$

The pdf is obtained by differentiation:

$$f_\tau(t) = \frac{dF_\tau(t)}{dt} = \alpha_n^T e^{Qt} Q P_0$$

□

Conversely, a distribution with a pdf of this form, parametrized by a vector of probability P_0 and a matrix Q with adequate properties (probabilities are non-negative and sum to 1, off-diagonal intensities are non-negative and the columns of Q sum to 0, the last column of Q is 0) is called a phase-type distribution and denoted $PH(P_0, Q)$.

As we see, any PH distribution has one associated CTMC, specified with the same initial probabilities and transition intensities. The PH distribution is the distribution of the absorption time of the CTMC. Conversely, for a CTMC to have an associated PH, it must have an *attainable* absorbing state, as we see next.

Chapter 3

Special PH distributions

While PH distributions are versatile, special cases such as the ones below will prove to be particularly important. These cases are much simpler to work with, and will serve as building blocks for better understanding PH distributions as a whole.

All we need to reason about all those special PH distributions is to remember that the exit time from state k is an exponential distribution with rate $-q_{kk}$:

$$\tau_k \sim Exp(-q_{kk})$$

3.1 Exponential distribution

Let us begin with a simple 2-state CTMC starting in state 1, with absorbing state 2: $P_0 = \alpha_1$ and:

$$Q = \begin{bmatrix} -\lambda & 0 \\ \lambda & 0 \end{bmatrix}$$

The system is absorbed when it exits state 1. Hence, the absorption time τ coincides with the holding time τ_1, and therefore follows an exponential distribution with rate λ.

Hence, the $PH(P0, Q)$ is simply an exponential distribution. Note that this also proves, by coincidence of the pdfs, that:

$$\alpha_2^T e^{\begin{bmatrix} -\lambda & 0 \\ \lambda & 0 \end{bmatrix} t} \begin{bmatrix} -\lambda & 0 \\ \lambda & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \lambda e^{-\lambda t}$$

3.2 Invalid PH

Before moving on to other special PH distributions, it is worth mentioning that some PH distributions do not make sense. Besides the obvious case without an absorbing state (when the last column of Q is nonzero), we have to consider cases where the absorbing state exists but is not attainable, as the following 4-state system:

$$P_0 = \begin{bmatrix} p_1 & p_2 & 0 & 0 \end{bmatrix}^T \quad \text{and} \quad Q = \begin{bmatrix} -\lambda_1 & \lambda_2 & 0 & 0 \\ \lambda_1 & -\lambda_2 & 0 & 0 \\ 0 & 0 & -\lambda_3 & 0 \\ 0 & 0 & \lambda_3 & 0 \end{bmatrix}$$

Reasoning about the associated CTMC, it may only start in state 1 or 2. From state 1, it can only jump to state 2, and from state 2, it can only jump back to state 1. Hence, the system is forever stuck in states 1 and 2, the state 3, the only state from which it could jump into the absorbing state, is unreachable; therefore, the absorbing state is also unreachable: $\tau = \infty$ with probability 1.

It follows that the PH CPD and pdf are uniformly zero.

Finally, if the last entry of P_0 is nonzero, the CTMC can jump into the absorbing state immediately, so that $Pr(\tau = 0) = \alpha_n^T P_0 > 0$. It follows that the pdf is infinite at 0, in other terms, the pdf is the sum of a density that integrates to $1 - \alpha_n^T P_0$, and a Dirac mass at 0 with weight $\alpha_n^T P_0$.

In order to avoid dealing with this special case, we only consider PH with zero probability to start in the absorbing state, such that $P_0 = [p_1, p_2, ..., 0]^T$

3.3 Erlang distribution

Let us now consider more complex CTMC, where $P_0 = \alpha_1$ and Q is of the form:

$$Q = \begin{bmatrix} -\lambda & 0 & 0 & 0 \\ \lambda & -\lambda & 0 & 0 \\ 0 & \lambda & -\lambda & 0 \\ 0 & 0 & \lambda & 0 \end{bmatrix}$$

(this is a four-state system for illustration, we consider the general case in some dimension n).

The system starts in state 1, and from state i, can only move forward to state $i + 1$, with fixed exit rate λ, until it reaches the absorbing state n. It follows that the absorption time is the sum of the holding times:

$$\tau = \sum_{k=1}^{n-1} \tau_k$$

where we have seen that the holding times $\tau_1, \tau_2, ..., \tau_{n-1}$ are independent and identically distributed with exponential distribution with the same rate λ. The sum of k independent exponentially distributed variables with same rate λ is known as the Erlang distribution, with shape k and rate λ. This is a well-studied distribution. Its pdf is:

$$f(t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{(k-1)!}$$

This demonstrates that the Erlang distribution can be represented as a special PH with P_0 and Q of the form above. In particular, it proves that (with this particular form of P_0 and Q):

$$\alpha_n^T e^{Qt} Q P_0 = \frac{\lambda^{n-1} t^{n-2} e^{-\lambda t}}{(n-2)!}$$

which is a nontrivial analytic exercise. But it is immediately apparent when reasoning about associated CTMCs.

3.4 Gamma distribution

The gamma distribution generalizes the Erlang distribution by allowing the shape parameter k to be any positive real number, not necessarily an integer.

Recall the Erlang pdf:

$$f(t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{(k-1)!}$$

The gamma pdf is obtained by replacing the factorial by its continuous extension, called the gamma function:

$$f(t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{\Gamma(k)} \quad t \geq 0$$

where

$$\Gamma(z) = \int_0^\infty t^z e^{-t}\, dt$$

is the gamma function. It coincides with the factorial function on integer values: $\Gamma(k) = (k-1)!$, and 'interpolates' them for all positive real numbers.

The gamma distribution therefore extends the Erlang distribution, in the sense that the two coincide for integer shapes, but the gamma function remains a valid distribution for shapes such as 0.1 or 2.5. It is particularly interesting with 'small' shapes between 0 and 1. For our purpose, it is not so much a topic of interest of its own, since, strictly speaking, it is not a PH distribution. But it makes it easier to work with Erlang distributions, in the context of estimation, because it is often easier to work with real parameters than integer ones.

3.5 Mixture of exponential distributions

Let us switch gears and consider CTMCs of the form:

$$P_0 = [p_1, p_2, p_3, 0]^T$$

and

$$Q = \begin{bmatrix} -\lambda_1 & 0 & 0 & 0 \\ 0 & -\lambda_2 & 0 & 0 \\ 0 & 0 & -\lambda_3 & 0 \\ \lambda_1 & \lambda_2 & \lambda_3 & 0 \end{bmatrix}$$

The system starts in any one of the $n-1$ non-absorbing states, with probabilities $p_1, p_2, ..., p_{n-1}$, and, from any of those states, it can only jump into the absorbing state, with intensities $\lambda_1, \lambda_2, ..., \lambda_{n-1}$. It follows that the absorption time could be distributed as $Exp(\lambda_1)$ with probability p_1, or $Exp(\lambda_2)$ with probability p_2, and so on.

This distribution is called a *mixture of* $n-1$ *exponential distributions* with distinct rates λ_i. More generally, we call mixture of m distributions d_i with pdfs f_i and probabilities p_i, the distribution of a variable drawn as follows:

1. Draw the hidden state $k \in [1, m]$ with probabilities $p_1, ..., p_m$

2. Draw the variable from the corresponding distribution d_k

By the law of total probabilities, its pdf is simply given by:

$$f_{mix}(t) = \sum_{k=1}^{m} p_k f_k(t)$$

In particular, the pdf of a mixture of exponential distributions is of the form:

$$f_{mix}(t) = \sum_{k=1}^{m} p_k \lambda_k e^{-\lambda_k t}$$

Mixtures of distributions are very useful in Physics, Machine Learning, and many other applications. In the second part about estimation, we provide an example from Physics, where the decay time of a heterogeneous radioactive gas follows a mixture of exponential distributions.

Once again, we easily derived, by reasoning about associated CTMCs, that PH distributions encompass mixtures of exponentials. The following result naturally follows by coincidence of pdf, although it is not easy to demonstrate by analytic means: for general P_0 (with 0 in the nth entry) and Q of the shape above:

$$\alpha_n^T e^{Qt} Q P_0 = \sum_{k=1}^{n-1} p_k \lambda_k e^{-\lambda_k t}$$

3.6 Mixtures of PH

The ability of PH distributions to represent mixtures is not limited to mixtures of exponential distributions. Mixtures of (say) Erlang distributions, or mixtures of distributions including both exponential and Elrang distributions can also be represented as PH distributions. In fact, any mixture of PH distributions can be represented as one PH distribution.

To see that, consider two PH distributions of respective dimensions n_1 and n_2, and respective parameters $(P_0^{(1)}, Q^{(1)})$ and $(P_0^{(2)}, Q^{(2)})$. Denote $\widetilde{P_0}$ the probability vector P_0 deprived of its last entry 0. This is a vector of size $n-1$. Similarly, denote $\widetilde{Q}$ the intensity matrix deprived of its last column (all 0) and last row (with intensities of jumping into absorption). Finally, denote q_n the last row of the intensity matrix Q.

We want to represent the mixture of the two PH distributions, with probabilities p_1 and p_2. We can achieve this with the PH distribution of dimension $n-1$, with parameters of figure 3.1:

With probability p_1, the system starts in one of the $n_1 - 1$ non-absorbing states of the first PH, with conditional probabilities $\widetilde{P_0^{(1)}}$. From there, it behaves exactly like the first CTMC until absorption, in particular, it cannot jump into the regime of the second CTMC. With probability p_2, the system starts in one of the $n_2 - 1$ non-absorbing states of the second PH, with conditional probabilities $\widetilde{P_0^{(2)}}$. From there, it behaves exactly like the second CTMC until absorption, in particular, it cannot jump into the regime of the first CTMC.

One of the numerical results of the next chapter relies on this construction to simulate a bimodal mixture of Erlang distributions.

We successfully constructed a PH that mixes the two PH distributions. This construction easily generalizes to mixtures of an arbitrary number of PH distributions, by induction. It follows that any mixture of PH is a PH.

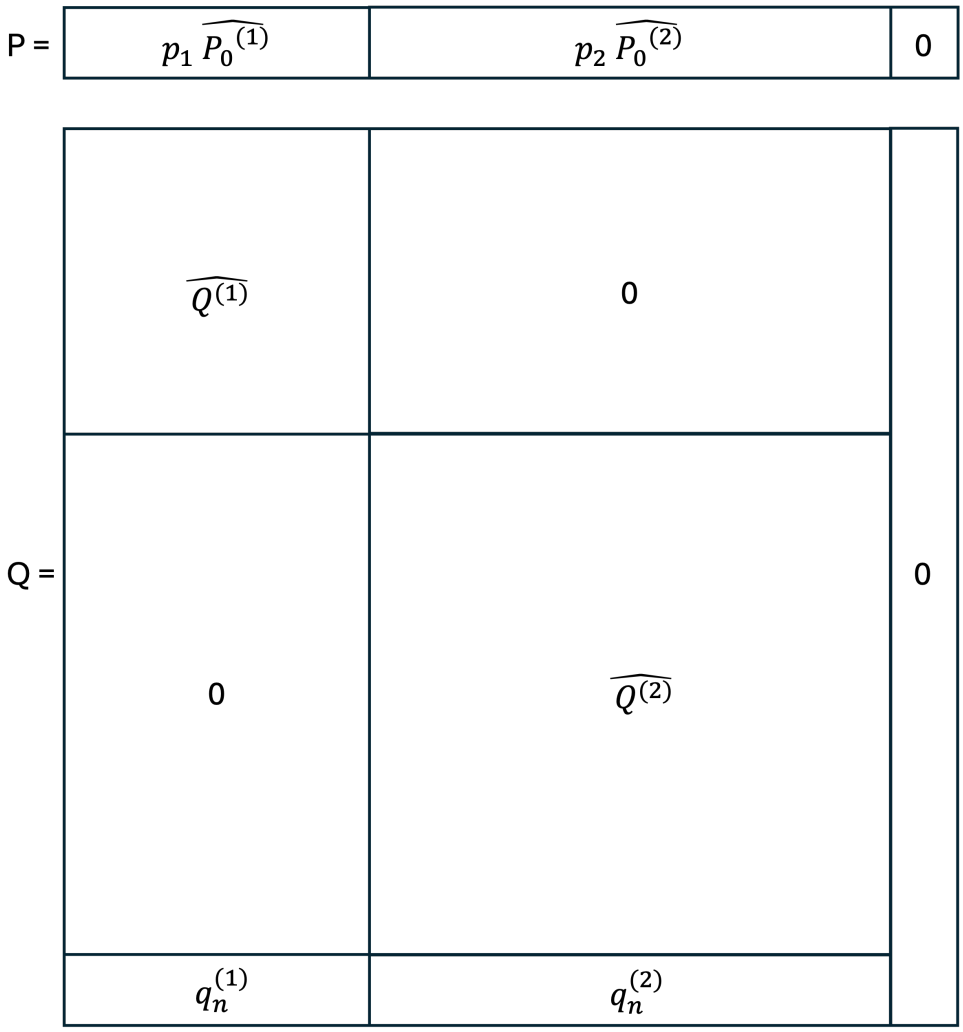

Figure 3.1: PH representation of a mixture of 2 PH

3.7 Density property of PH distributions

What precedes is only a glimpse at the impressive expressiveness of PH distributions, and the power of reasoning about associated CTMCs. The following theorem provides the full measure of this expressiveness, essentially stating that *any* distribution can be represented as a PH.

Theorem (Density of PH distributions)**.** *The set of phase-type distributions is dense in the field of all positive-valued distributions, that is, it can be used to approximate any positive-valued distribution to arbitrary precision.*

This is similar to the ability of polynomials to approximate any smooth function to arbitrary precision, by Taylor expansion.

This concludes our theoretical review of CTMC and PH distributions. The next chapters are concerned with simulation and estimation.

Chapter 4

Sampling of PH distributions and simulation of CTMC paths

The sampling of PH distributions is one of the stated goals of this thesis. The second goal is estimation: given a sample of m independent realizations of a PH-distributed variable, for example, repeated measurements of the time it takes an atom to reach its ground state, estimate the parameters P_0 and Q. In particular, we are going to investigate a relatively novel approach to estimation, by way of Machine Learning (ML). The main idea is to train a ML model on a *simulated* dataset, by contrast to the usual ML practice of training ML models with datasets picked in the real world.

Therefore, sampling is not only a goal in itself, but also, a prerequisite to estimation by ML. In this section, we introduce sampling algorithms for random variables in general, and PH distributions in particular, exhibit an algorithm, and review some results of its implementation in Python.

4.1 Sampling random variables

Sampling from a distribution F_X means to obtain multiple independent measurements of the random variable X. When this is performed by a computer program, we call it simulation. For example, all programming languages offer some way to draw random numbers uniformly in $(0, 1)$. In Python, we can call numpy.random.uniform().

The concrete implementation of these uniform samplers is a discipline of its own, out of the scope of this document. What is more interesting for us is that those uniform samples provide a basis for sampling from arbitrary distributions. The following theorem provides an "army knife" for sampling.

Theorem (Sampling from arbitrary distributions)**.** *If U is drawn from a uniform distribution over $(0, 1)$, then $X = F_X^{-1}(U)$ is a random variable with CPD F_X.*

Proof. Denote $Y = F_X^{-1}(U)$. Then:

$$F_Y(y) = Pr(Y \leq y) = Pr[F_X(Y) \leq F_X(y)]$$

since F_X is increasing

$$F_Y(y) = Pr[F_X(F_X^{-1}(U)) \leq F_X(y)] = Pr[U \leq F_X(y)] = F_U[F_X(y)] = F_X(y)$$

since $F_U(u) = u$.

Therefore, $F_Y = F_X$ □

This theorem provides a direct algorithm for sampling from any distribution whose CPD is easily inverted: draw a uniform sample and feed it to the inverse CPD to obtain a sample of the desired distribution. For example, we can easily sample the exponential distribution, using the inverse CPD calculated earlier.

In practice, there exists a function in Python for drawing from exponential distributions: the function exponential() from numpy.random. This implementation parameterizes exponential distributions, not with the rate λ, but its inverse $1/\lambda$, called "scale". Like all numpy functions, it is efficient and vectorized, in the sense that it can simulate many samples at the same time, using SIMD capability of the CPU.

The general method may or may not be tractable, depending on the CPD. For example, the CPD of a Gaussian distribution has no analytic expression, and neither does its inverse. The general PH distribution does have an analytic CPD, but it is not analytically invertible. For those distributions where the general method is not easily applicable, other options include trying to compute accurate approximations of the inverse CPD, analytically or numerically, or design specific algorithms for this particular distribution. For example, the Box-Muller algorithm efficiently simulates Gaussian variables. For PH distributions, a natural and common method is to simulate the underlying CTMC and measure its absorption time.

4.2 Sampling CTMC

4.2.1 Distribution of holding times and next states

Therefore, sampling PH reduces to the simulation of CTMC sample paths. Earlier in this document, we designed and implemented in Python an algorithm for the simulation of *discrete-time* Markov chains. This was essentially trivial: iterate over time steps, drawing the next state from the probability distribution conditional to the present state k, given by the kth column of the transition matrix R.

This is much less immediate in continuous time, where there is no time steps, and the process can jump any time. We follow the steps described in [8]. The main idea is to iterate over *jump* times, when the process changes state. We have already seen that holding times follow exponential distributions, and that those distributions are easily sampled. What remains is to pick the next state, where the process jumps when it leaves its former state.

Proposition (Distribution of next state)**.** *Consider a CTMC with intensity matrix Q, entering state k at time t, and leaving it at time $t + \tau_k$. The distribution of the next state after leaving state k is:*

$$Pr(X_{t+\tau_k} = j) = \frac{q_{jk}}{-q_{kk}} \text{ for } j \neq k \text{ and } Pr(X_{t+\tau_k} = k) = 0$$

Sketch of proof. *Conditionally to leaving state k, the probability of jumping into state k is obviously* 0*.*

Probabilities of jumping into other states j, conditionally to leaving state k, are:

$$\begin{aligned}
Pr(X_t = j | X_{t-dt} = k \text{ and } X_t \neq k) &= \frac{Pr(X_t = j \text{ and } X_{t-dt} = k \text{ and } X_t \neq k)}{Pr(X_{t-dt} = k \text{ and } X_t \neq k)} \\
&= \frac{Pr(X_t = j \text{ and } X_{t-dt} = k)}{Pr(X_{t-dt} = k \text{ and } X_t \neq k)} \\
&= \frac{Pr(X_t = j | X_{t-dt} = k) Pr(X_{t-dt} = k)}{Pr(X_t \neq k | X_{t-dt} = k) Pr(X_{t-dt} = k)} \\
&= \frac{Pr(X_t = j | X_{t-dt} = k)}{Pr(X_t \neq k | X_{t-dt} = k)} \\
&= \frac{q_{jk}}{\sum_{l \neq k} q_{lk}} \\
&= \frac{q_{jk}}{-q_{kk}}
\end{aligned}$$

4.2.2 Simulation algorithm

Now we know how to simulate holding times and next states, we can state the full simulation algorithm.

Algorithm (Simulation of CTMC)**.** *A CTMC with initial probabilities P_0, intensity matrix Q and one absorbing state n is simulated as follows:*

1. *Draw initial state k from distribution P_0. Memorize the resulting state $X_0 = k$ at time $t = 0$.*
2. *Repeat until absorbing state $k = n$:*
   (a) *Draw the holding time τ_k from exponential distribution with rate $\lambda = -q_{kk}$. Update current time $t = t + \tau_k$.*
   (b) *Draw the next state j from the distribution $P_j = q_{jk}/-q_{kk}$. Update current state to $k = j$ at time t.*

For example, we work through one sample path for a CTMC with 3 states, the third one being the absorbing state.

First, we sample the initial state from the probability distribution $P_0 = (p_1, p_2, p_3)$. Say, we draw state 2. The simulation starts with state 2 at time 0.

Next, we draw the holding time in state 2 from the exponential distribution with rate $\lambda = -q_{22}$. Say, we draw $23sec$. The process jumps out of state 2 at time $t = 0 + 23 = 23$. We also draw the next state, which may be 1 or 3 but not 2, since we jumped out of state 2. Probabilities are given by:

$$Pr(X_{23} = 1) = q_{12}/-q_{22}$$
$$Pr(X_{23} = 3) = q_{13}/-q_{22}$$

Say, we draw state 1. This means the process spent $23sec$ in state 2 and jumped into state 1 at time $t = 23$.

We repeat, drawing the holding time in state 1 from the exponential distribution with rate $\lambda = -q_{11}$. Say, we draw $7sec$. The process jumps out of state 1 at time $t = 23 + 7 = 30$. We draw the next state with probabilities given by:

$$Pr(X_{30} = 2) = q_{21}/-q_{11}$$
$$Pr(X_{30} = 3) = q_{31}/-q_{11}$$

Say, we draw 3, the absorbing state, thereby stopping the simulation. The resulting path is:

$$[(0, 2), (23, 1), (30, 3)]$$

In particular, the absorption time, (implicitly) drawn from the phase-type distribution with parameters P_0 and Q, is 30. We can repeat and independently sample m paths to obtain a sample of size m for this phase-type distribution.

4.2.3 Numerical results

The simulation algorithm is implemented with Python code given in appendix, and also available online in the notebook markov_chain_simulation_continuous_time.ipynb of our GitHub repo.

In this notebook, we developed a CTMC class initialized with a probability vector P_0 and intensity matrix Q, checking that the dimensions are consistent and that P_0 and Q have the required properties. We also check for an absorbing state by ensuring that the last column of Q is 0.

Our CTMC class implements the methods for an analytic computation of state probabilities, both unconditional and conditional to being in state k at some prior time T, and the CPD and pdf of the absorption time, which define the $PH(P_0, Q)$ distribution.

Finally, the CTMC class provides methods for the simulation of one or multiple sample paths, with the algorithm above.

We tested the implementation with fixed P_0 and Q in dimension $n = 4$, as seen on figure 4.1, along with one example of a sample path.

```
[208] # pick P0 and Q

      n = 4
      P0 = make_random_probability_vector(n)
      Q = make_random_intensity_matrix(n)
```

```
[209] P0
```

```
array([0.32408362, 0.51063714, 0.13841181, 0.02686743])
```

```
[210] Q
```

```
array([[-1.36541951,  2.80742967,  0.32159156,  0.        ],
       [ 0.12102277, -3.50583472,  0.31075694,  0.        ],
       [ 0.09255567,  0.33950404, -1.65552051,  0.        ],
       [ 1.15184108,  0.35890102,  1.023172  ,  0.        ]])
```

```
[240] # sample a path
      ctmc = CTMC(P0, Q)
      ctmc.sample_path()
```

```
array([[0.        , 0.27627849, 0.60422601, 0.69601773, 1.06523361],
       [1.        , 2.        , 1.        , 0.        , 3.        ]])
```

Figure 4.1: One sample CTMC path. The process started in state 2 (showing 1, indexing starts at 0 in Python) at time 0, jumped into state 3 (showing 2) at time 0.28, state 2 at time 0.60, state 1 at time 0.70, and finally jumped into the absorbing state 4 (showing 3) at time 1.07. The absorption time was therefore 1.07 for this particular run.

Next, we compared the empirical state frequencies with theoretical probabilities, like we did for discrete-time simulations. Keeping parameters P_0 and Q fixed at the values of figure 4.1, we simulated sample paths and

compared the empirical frequencies of the $n = 4$ states at time $t = 1$ with the theoretical probabilities computed with the formula $P_t = e^{Qt}P_0$. The result is displayed in figure 4.2, where we see that empirical frequencies converge nicely towards theoretical probabilities as the number of sample paths grows. Note that the simulation is not based on the matrix exponential formula, but on simple conditional probabilities of holding times and next states. Yet, the empirical and theoretical distributions appear entirely consistent.

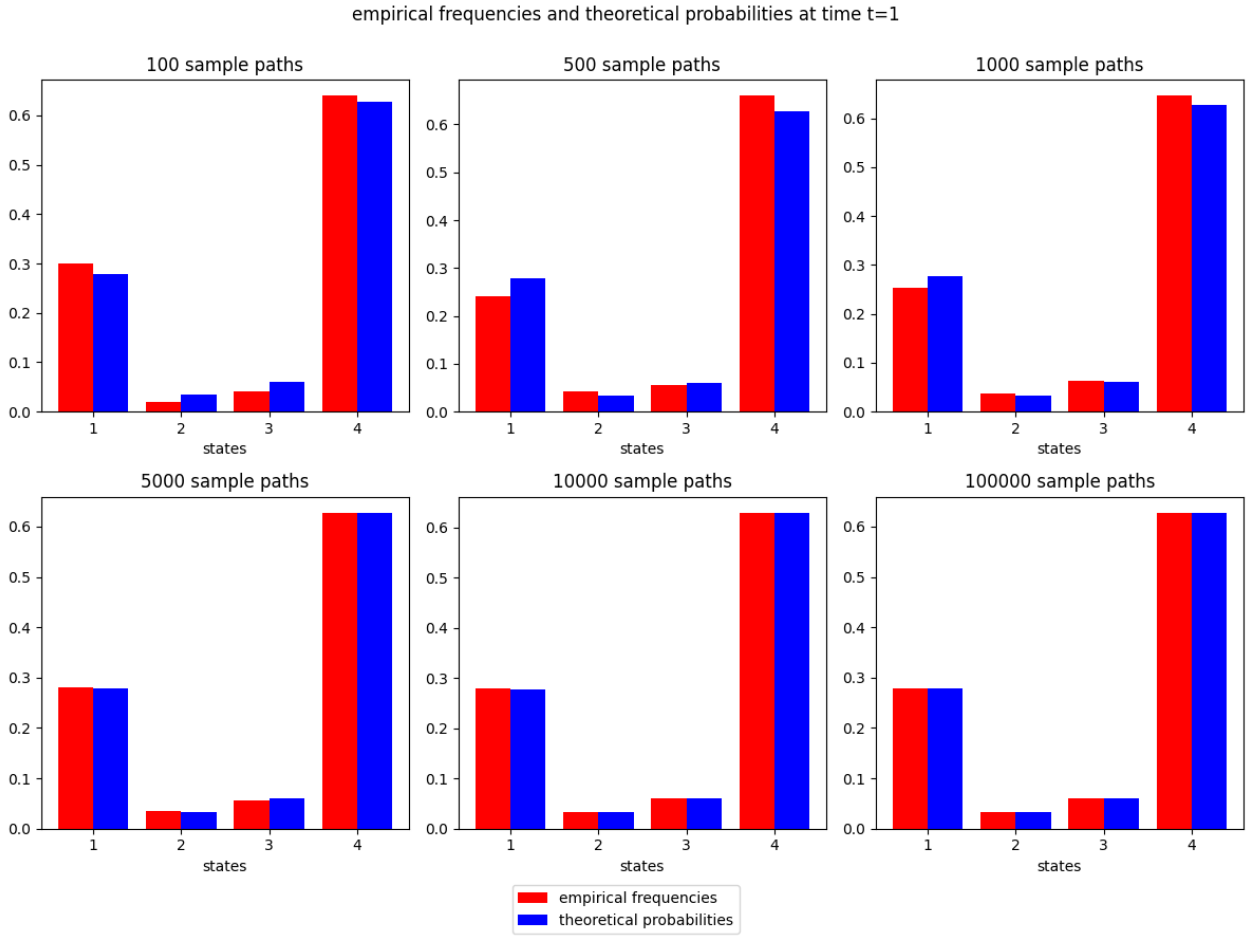

Figure 4.2: Numerical results for time $t = 1$ from the simulation of a 4-state CTMC with fixed parameters P_0 and Q given in figure 4.1. Theoretical state probabilities are $P_t = e^{Qt}P_0$ (with $t = 1$). Empirical state frequencies are estimated on simulated sample paths.

Next, from the same sets of sample paths with fixed P_0 and Q, we compared the empirical distribution of the simulated absorption times with the theoretical $PH(P_0, Q)$ pdf $f_\tau(t) = \alpha_n^T e^{Qt} Q P_0$. The result is given in figure 4.3, where we also see the convergence of the empirical density towards the theoretical PH pdf.

Finally, we repeated the experiment with the more interesting mixture of Erlang(2, 0.5) and Erlang(15, 1), with probabilities 1/2 and 1/2. This distribution happens to be bimodal, as exhibited in the notebook. We used the construction of the previous chapter to represent it as a CTMC, and simulate with the CTMC class in the notebook. The result is displayed on figure 4.4.

We checked that the theoretical PH pdf exactly coincides with the average of the two Erlang pdfs, and also checked the convergence of empirical densities towards this pdf, as the number of simulated paths grows. The result is shown on figure 4.5.

At this point, we have a concrete method for sampling PH distributions with arbitrary parameters. We can

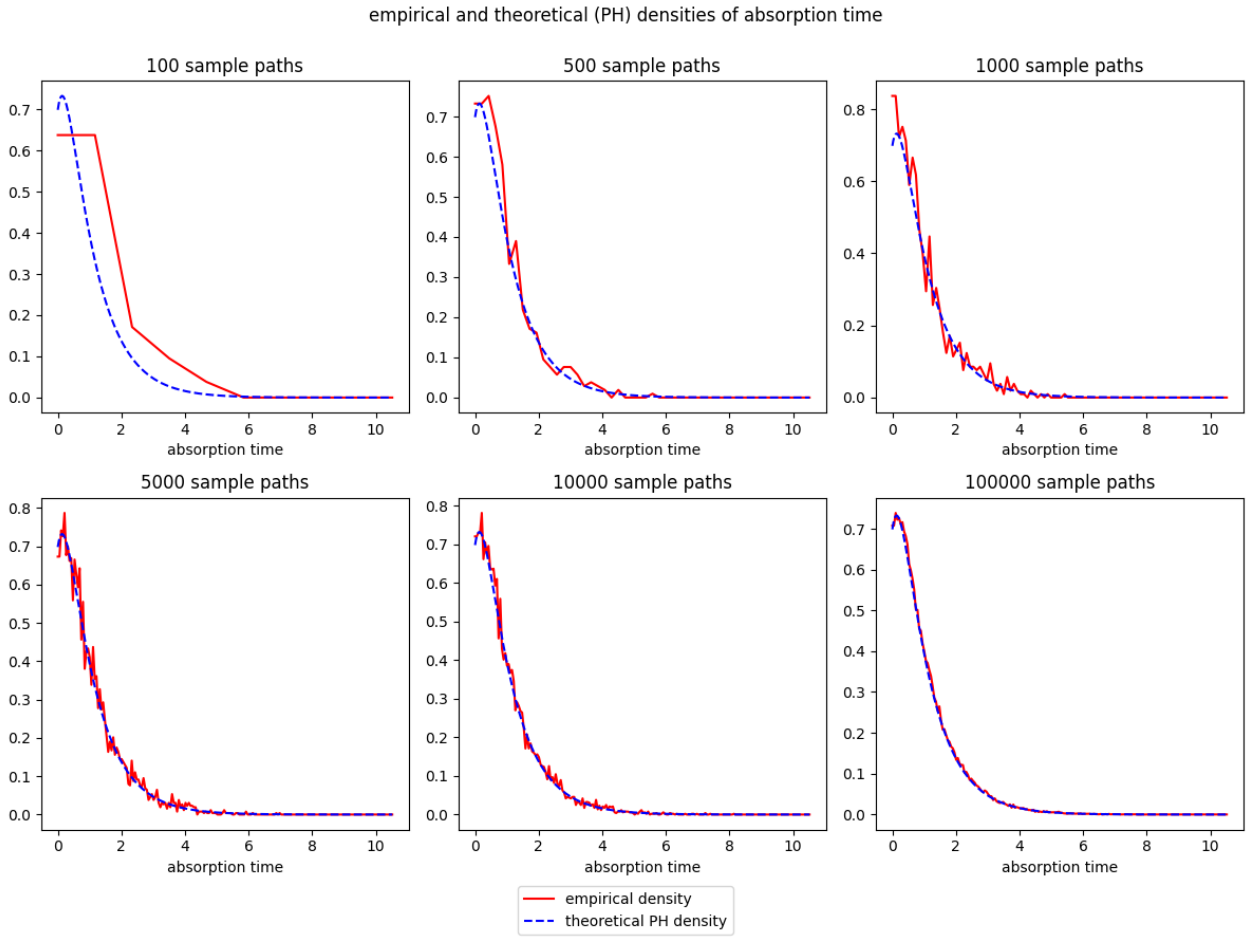

Figure 4.3: Numerical results from the simulation of sample paths for a 4-state CTMC with fixed parameters P_0 and Q given in figure 4.1. Theoretical PH density is $f_\tau(t) = \alpha_n^T e^{Qt} Q P_0$. Empirical density is obtained by counting frequencies of sampled absorption times in uniform time intervals, normalized by interval length. We used 200 intervals for larger samples, and the number of sample paths /10 for smaller samples.

use it as a building block for training Machine Learning models to estimate PH parameters from samples, on simulated datasets. This is the main topic for the rest of the document.

```
P0
[0.5 0.  0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
Q
[[-0.5  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.5 -0.5  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0.   0. ]
 [ 0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.  -1.   0. ]
 [ 0.   0.5  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   1.   0. ]]
```

Figure 4.4: PH representation of a 50/50 mixture of Erlang(2, 0.5) and Erlang(15, 1)

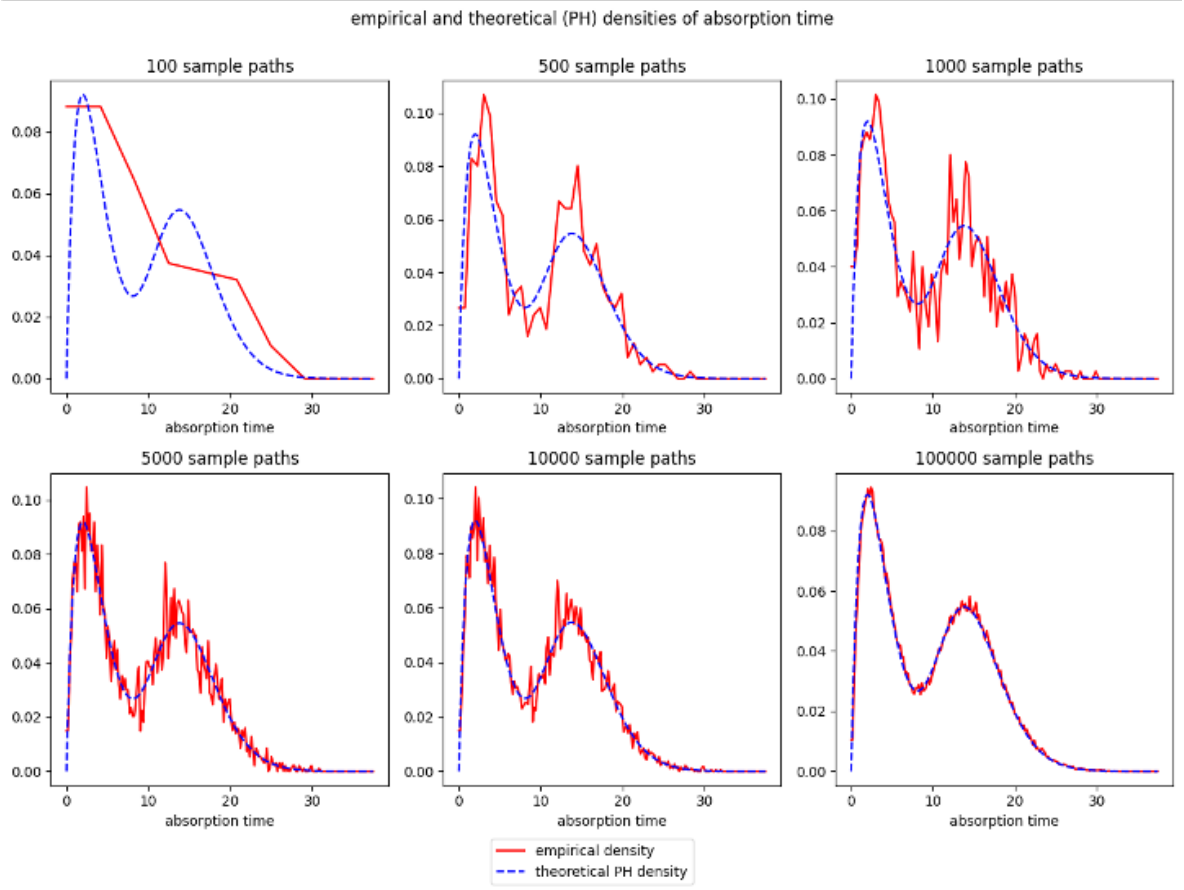

Figure 4.5: Repeat the experience of figure 4.3 with bimodal 50/50 mixture of Erlang(2, 0.5) and Erlang(15, 1), represented as a PH with the parameters of figure 4.4

Part II

Estimation with or without Machine Learning

Chapter 5

Statistical estimation

The rest of the document focuses on the estimation problem in general, and the estimation of PH distributions in particular. Estimation is a central problem in Statistics: given a sample drawn from some distribution, infer an estimate of its parameters. For example, to measure the bias on a coin, we may throw it independently a thousand times, and call X_i the binary variable worth 1 if throw number i is heads, 0 otherwise. The variables X_i are independently and identically distributed Bernoulli variables with parameter μ, the true probability of heads. The $1,000$-dimensional vector $X = (X_1, X_2, ..., X_{1000})$ is our sample. By observing the sample, we can infer an estimate $\widehat{\mu}$ of μ. This estimate has to be a deterministic function of the sample: $\widehat{\mu} = g(X)$. In this case, the obvious estimate is the observed frequency of heads, in other terms, the sample mean: $\widehat{\mu} = \overline{X}$. Since the sample X is random, the estimate $\widehat{\mu}$ is also a random variable. We can compute its expectation. In this case, trivially, $E[\widehat{\mu}] = \mu$. An estimator with this property is called unbiased. We can also compute its variance. In this case, $Var[\widehat{\mu}] = \mu(1-\mu)/1000 \approx \widehat{\mu}(1-\widehat{\mu})/1000$. We see that the variance goes to zero when the sample size grows, and with a finite sample, its standard deviation is inversely proportional to the square root of the sample size. The standard deviation of the estimator is also called standard error. In addition, the Central Limit Theorem states that the distribution of $\widehat{\mu}$ converges to a Gaussian distribution, which allows to compute confidence intervals and perform statistical tests.

The *estimation function* g is how we use the sample to estimate parameters. In the trivial example, $g(x) = \overline{x}$. In general, the construction of this estimation function is a nontrivial matter, and the main topic of the rest of this document.

As a (slightly) less trivial example, the decay time τ of a radioactive isotope follows an exponential distribution with rate λ depending on the isotope. Suppose we have n independent measurements $\tau_1, \tau_2, ..., \tau_n$ of individual decay times. We can use this sample to estimate λ, and thereby identify the isotope. As we are going to see, a good estimate of the rate of the exponential distribution is $\widehat{\lambda} = g(\tau) = 1/\overline{\tau}$, the inverse sample mean.

Suppose now that we observe a heterogeneous set of 3 different isotopes, with proportions p_1, p_2, p_3 and decay rates $\lambda_1, \lambda_2, \lambda_3$. As before, we collect a sample of n decay time measurements $\tau_1, \tau_2, ..., \tau_n$, but we don't know which isotope was measured in each one the n measurements. τ_1 could be a realization of an exponential distribution with rate λ_1, with probability p_1, or it could be λ_2 with probability p_2, or λ_3 with probability p_3. This is a mixture of 3 exponential distributions, parameterized by 6 parameters: the 3 probabilities, and the 3 corresponding rates. Estimation, in this case, consists in finding a function $\widehat{p_1}, \widehat{p_2}, \widehat{p_3}, \widehat{\lambda_1}, \widehat{\lambda_2}, \widehat{\lambda_3} = g(\tau_1, \tau_2, ..., \tau_n)$ so we can estimate those parameters from the sample. The estimation of a mixture of exponentials is a nontrivial problem, which we investigate later in this document.

As a final motivating example, suppose we repeatedly excite an atom and measure the time τ it takes to revert to its ground state, thereby building a sample $\tau_1, \tau_2, ..., \tau_n$. We have seen that the distribution of τ is a PH distribution with parameters initial probabilities P_0 and intensity matrix Q. With 3 transient states (4 states in total including the absorbing ground state), this is 4 probabilities and 12 intensities, or 16 parameters (actually less, since probabilities sum to 1 and the columns of Q sum to 0). The estimation of

these parameters from the sample is a highly nontrivial problem, and the main topic of the 172 pages of [3]. This is also the ultimate goal of our thesis, albeit with a very different approach. Sadly, it was not achieved for lack of time.

Let us formalize these notions first.

Definition (Sample)**.** *Given a distribution $D(\theta)$ parametrized with a vector of p parameters $\theta = (\theta_1, \theta_2, ..., \theta_p)$, a sample $X = (X_1, X_2, ..., X_n)$ is a collection of n independent and identically distributed (iid) random variables, all from distribution D.*

Definition (Estimator)**.** *With the notations of the previous definition, an estimation function is a function g from $\mathbb{R}^n$ into $\mathbb{R}^p$. An estimator $\widehat{\theta}$ for the parameters θ is obtained by evaluation $\widehat{\theta} = g(X)$ of g with a sample X.*

By this definition, any function with the right source and destination domains is a valid estimator. For example, a function that always returns 0 is a valid estimator. But this is probably not a very good one. The estimator is supposed to provide a "good" approximation of the "real" parameters of the distribution. The performance of an estimator quantifies how good it is at this job. The most common performance metrics is the mean-squared-error (MSE), itself split into bias and variance.

Definition (Estimation performance metrics)**.** *Recall that an estimator is a random variable, since it is given by a deterministic function g evaluated on a random vector X. The mean-squared-error (MSE) of an estimator $\widehat{\theta}$ is:*

$$mse(\widehat{\theta}) = E[(\widehat{\theta} - \theta)^2]$$

Its bias is given by the formula:

$$bias(\widehat{\theta}) = E[\widehat{\theta} - \theta]$$

and its variance is simply $Var[\widehat{\theta}]$. Recall that for any variable Y, $E[Y^2] = E[Y]^2 + Var[Y]$, hence:

$$mse = bias^2 + variance$$

Finally, the standard deviation (std) is the square root of the variance, and the root-mean-squared-error (RMSE) is the square root of the MSE.

Remark (Vector estimators)**.** *The previous definition is for scalar estimators. When there is more than one parameter to estimate, both the parameters θ and estimators $\widehat{\theta}$ are vectors in dimension p. In this case, the previous definitions are applied parameter-wise, i.e. $mse_i = E[(\widehat{\theta}_i - \theta_i)^2]$. Those coefficient-wise MSEs can be combined into a synthetic performance measure, for example, by summing, with or without weights, or by taking the worst performance across the p parameters.*

One of the most important theorems of Statistics, called *Cramer-Rao bound*, establishes a lower bound on the variance of *any* unbiased estimator. For a given distribution, an unbiased estimator whose variance hits the Cramer-Rao bound is as good as an estimator will ever be.

With the concepts and formalism in place, we now focus on how to perform estimation in practice, that is, the construction of the estimation function g, first by established statistical means, and then by a relatively new Machine Learning approach.

Chapter 6

Maximum Likelihood Estimation (MLE)

Suppose that you are given a family of distributions $D(\theta)$ and a sample $X = (X_1, X_2, ..., X_n)$ and asked to compute an estimate of the parameter θ. How do you get started? The problem is easy in simple cases. For a Bernoulli distribution, the sample frequency is the the obvious estimator of the Bernoulli probability. For a Gaussian distribution with mean μ and variance σ^2, the sample mean and variance are also obvious choices. For an exponential distribution, it may take more thought to estimate the rate λ. The expectation of the exponential distribution is $1/\lambda$, therefore the sample mean is a good estimate of the rate's inverse, hence, the inverse sample mean is a good estimate of the rate. All of those are common sense choices, you can start there, and then, compute MSEs, biases and variances to get a better sense of what performance to expect.

But those common sense insights do not always apply to more complicated distributions: what about a mixture of distributions? How do you get started estimating those probabilities of an unobserved state, along with the parameters that govern different states? And what about a PH distribution? How do you estimate the intensities of jumping from state to state, where you don't get to observe those states, only the time it takes to end up in the absorbing state from an initial state that is not observed either?

Classic statistics provide an "army knife", applicable in principle to arbitrary distributions: maximum likelihood estimation or MLE. The main idea is to estimate the parameters by the value of the parameters that maximizes the probability of observing the sample.

This is best explained through a simple example, the Bernoulli distribution. A variable B with Bernoulli(μ) distribution has probabilities $Pr(B = 1) = \mu$ and $Pr(B = 0) = 1 - \mu$, which we can write more concisely $Pr(B = x) = \mu^x(1-\mu)^{1-x}$ for $x \in \{0, 1\}$. Since the n variables in the sample are independent and identically distributed with the same distribution as B, the probability of the sample is simply the product of n identical probabilities:

$$Pr(B_1 = X_1, B_2 = X_2, ..., B_n = X_n) = \prod_{i=1}^{n} Pr(B_i = X_i) = \prod_{i=1}^{n} \mu^{X_i}(1-\mu)^{1-X_i}$$

This probability of observing the sample, given the value of the parameters (in this case, μ), is called *the likelihood* of the parameter, and denoted $L(\mu)$ (it depends on the sample too, but we have observed the sample at this stage, and consider it fixed). The MLE principle tells us to find the value of μ that maximizes $L(\mu)$:

$$\widehat{\mu} = \arg\max_{\mu} L(\mu)$$

Now, the likelihood $L(\mu)$ is a product. It is generally easier to work with sums. Since the argument that maximizes a function and its logarithm coincide, we easily reduce the hard problem to an easier one, simply by maximizing the log-likelihood:

$$\widehat{\mu} = \arg\max_{\mu} \log L(\mu) = \sum_{i=1}^{n} [X_i \log \mu + (1 - X_i) \log(1 - \mu)]$$

In this simple case, the problem is easily solvable analytically, but in more complicated cases, it has to be solved numerically. Numerical optimization algorithms are generally written to perform minimizations rather than maximizations. For this reason, it is more common to minimize the *negative log-likelihood* (nll), which is of course absolutely equivalent:

$$\widehat{\mu} = \arg\min_{\mu} nll(\mu) \text{ where } = nll(\mu) = -\log L(\mu) = -\sum_{i=1}^{n} X_i [\log \mu + (1 - X_i) \log(1 - \mu)]$$

At the minimum $\widehat{\mu}$, the derivative $dnll/d\mu$ must be 0, hence:

$$-\sum_{i=1}^{n} \left[\frac{X_i}{\widehat{\mu}} - \frac{1 - X_i}{1 - \widehat{\mu}} \right] = 0 \text{ or } \frac{\sum_{i=1}^{n} X_i}{\widehat{\mu}} = \frac{n - \sum_{i=1}^{n} X_i}{1 - \widehat{\mu}}$$

$$(1 - \widehat{\mu}) \sum_{i=1}^{n} X_i = \widehat{\mu}(n - \sum_{i=1}^{n} X_i)$$

$$\sum_{i=1}^{n} X_i - \widehat{\mu} \sum_{i=1}^{n} X_i = \widehat{\mu} n - \widehat{\mu} \sum_{i=1}^{n} X_i$$

$$\sum_{i=1}^{n} X_i = \widehat{\mu} n$$

$$\widehat{\mu} = \frac{1}{n} \sum_{i=1}^{n} X_i = \overline{X}$$

Hence, the MLE effectively corresponds to the common sense estimation of probability by frequency. Now, this is a lot of reasoning and calculation to derive an intuitively evident result, but the whole point of the maximum likelihood machinery is to provide an approach applicable in all cases, including more complicated ones, where common sense doesn't help.

With continuous distributions, we use densities (which are probabilities normalized by interval length) instead of probabilities, and proceed with the exact same steps. Let us formalize the MLE machinery before we explore the continuous example of the exponential distribution.

Definition (Maximum Likelihood Estimator)**.** *Given a family of distributions $D(\theta)$ with pdf $f(x; \theta)$ and an independent sample $X = (X_1, X_2, ..., X_n)$ from this distribution:*

- *The likelihood $L(\theta)$ of some value θ of the parameters is the product of the densities of the sample datapoints X_i, computed with parameters θ:*

$$L(\theta) = \prod_{i=1}^{n} f(X_i; \theta)$$

- *The negative log-likelihood (nll) is (as the name implies):*

$$nll(\theta) = -\log L(\theta) = -\sum_{i=1}^{n} \log f(X_i; \theta)$$

- *The maximum likelihood estimator (MLE) $\widehat{\theta}$ of θ is the value of theta that maximizes the likelihood:*

$$\widehat{\theta} = \arg\max_{\theta} L(\theta)$$

- *It is often more practical to find it as the arg min of the negative log-likelihood (nll):*

$$\widehat{\theta} = \arg\min_{\theta} nll(\theta) \text{ where } = nll(\theta) = -\log L(\theta)$$

- *In particular, the MLE must satisfy:*

$$\frac{\partial nll(\widehat{\theta_j})}{\partial \theta_j} = 0$$

  for all parameters θ_j.

Example (MLE for exponential distribution)**.** *Recall the pdf is: $f(t; \lambda) = \lambda e^{-\lambda t}$, hence:*

$$nll(\lambda) = -\sum_{i=1}^{n} [log\lambda - \lambda\tau_i]$$

$$\frac{dnll(\lambda)}{d\lambda} = -\sum_{i=1}^{n} [\frac{1}{\lambda} - \tau_i] = -\frac{n}{\lambda} + \sum_{i=1}^{n} \tau_i = 0 \iff \lambda = \frac{n}{\sum_{i=1}^{n} \tau_i} = \frac{1}{\overline{\tau}}$$

We have seen two examples where the MLE coincides with common sense estimation. Another well-known example is the Gaussian distribution, where the MLEs for mean and variance are the sample mean and variance, respectively. It is tempting to extrapolate from those few examples and conjecture that the MLE, in addition to providing a general approach to estimation based on a reasonable principle, always leads to estimators with "nice" properties. Indeed, the following theorem (sort-of) confirms this conjecture:

Theorem (Asymptotic efficiency of MLE)**.** *Asymptotically as the sample size grows to infinity, the MLE is unbiased and hits Cramer-Rao's bound.*

This is fantastic news for all people working with infinite samples. (We will see that we do work with virtually infinite samples in the ML approach, unfortunately without the asymptotic guarantees of the MLE). What this means in practice is that the MLE is a good choice with large samples, but not necessarily with small samples. As an example, it is well-known that the sample variance is a biased estimator of the Gaussian variance, by factor $(n-1)/n$, which effectively quickly converges to 1 when the sample grows.

Now, this is a minor flaw for most practical applications. The major drawback of the MLE is tractability. In the simple examples considered so far, the MLE methodology leads to a simple and practical analytic formula. But this is no longer the case, even with slightly more complicated distributions.

Consider, for instance, a mixture of two exponential distributions with probabilities $p_1 = P$ and $p_2 = 1 - P$, and rates λ_1 and λ_2. By the law of total probabilities, its pdf is:

$$f(t; P, \lambda_1, \lambda_2) = P\lambda_1 e^{-\lambda_1 t} + (1 - P)\lambda_2 e^{-\lambda_2 t}$$

A sum of two products. Logarithms will not help. The expressions of the nll and its derivatives with respect to parameters are particularly unsavory and not reproduced here. Suffice to say, the resulting system of 3 nonlinear equations doesn't have an analytic solution.

What can we do in this case? We could invoke numerical algorithms to solve the roots on the nll derivatives, or directly minimize the nll. Beware that those problems have constraints: P must be between 0 and 1, and lambdas must be positive. Anyhow, this is a particularly expensive estimation, and without guarantee of success: numerical optimizers may be stuck on local optima.

An alternative algorithm, particularly effective for the estimation of mixtures, is the so-called expectation-maximization (EM) algorithm, which efficiently converges to a local maximum of the likelihood.

For more complicated distributions, maximum likelihood estimation must be treated on a case-by-case basis. For PH distributions, an efficient instance of the EM algorithm also exists, but its derivation is very hard work, as seen in PhD thesis [3].

For some distributions, such as multivariate PH distributions, it exists (to our best knowledge) no tractable means of maximizing likelihood.

Hence, efficient means of computing the MLE may take considerable work, on a case-by-case basis, that may or may not be successful. This alone justifies research into a different approach, with the promise of a general methodology, applicable to arbitrary distributions without modification, irrespective of complexity, after a substantial, but one-time research and development investment. This is the promise of estimation by Machine Learning.

Chapter 7

Estimation by Machine Learning

7.1 The Machine Learning approach

From the definition of an estimator, it is nothing more, nothing less than a function from $\mathbb{R}^n$ (the size of the sample) into $\mathbb{R}^p$ (the number of parameters). The quality of an estimator is measured by a metric such as MSE. Hence, essentially, the problem is to find a function that optimizes a metric. This is exactly what Machine Learning does.

Mathematical functions are mirrored in code by programming functions, such as the mock code below.

```
    import numpy as np

    def estimator(
        sample: np.array,
    ) -> np.array:
        # logic that computes parameter estimates from sample
        return parameter_estimates
```

Traditionally, programming functions translate in computer code some logic resulting from prior analysis. For instance, we have seen that a good estimator for an exponential distribution is the inverse sample mean. This simple result is based on at least two deep theorems: the Cramer-Rao bound, and the asymptotic efficiency of the MLE, as well as the mathematical derivation of the MLE. All of this was performed as prior analysis, and results in the simple program, that translates the end result in code:

```
    def exponential_rate_estimator(
        sample: np.array,
    ) -> np.array:
        rate = 1 / sample.mean()
        parameter_estimates = np.array([rate])
        return parameter_estimates
```

Machine learning turns this workflow on its head. Instead of translating in code prior logic derived or specified by human beings, we show the ML model some *examples* of inputs, along with corresponding ground truth outputs, and let the model figure the necessary connections and patterns to deduce those outputs from the inputs, and generalize to unseen inputs.

To begin understanding how this is even possible, it may help to approach ML models as parameterized black boxes of the form in figure 7.1, in other terms, a parametrized function $y = f(x; \gamma)$.

The model is *trained* with a dataset of m training examples. Each example i consists of a training input x_i, along with the training target y_i. Training optimizes the model's internal parameters γ so as to minimize a measure of errors between the model's outputs and ground truth targets (generally the MSE):

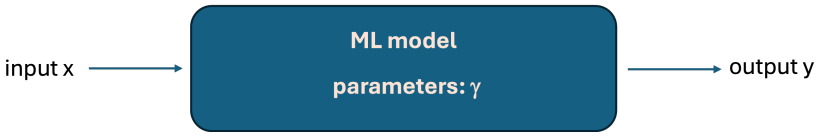

Figure 7.1: Machine Learning model as a parametrized black box. Input x and output y are generally vectors in appropriate dimension.

$$\arg\min_{\gamma} \sum_{i=1}^{m} [f(x_i;\gamma) - y_i]^2$$

Once a model is trained, its parameters are fixed on the optimum, and the model can compute *predictions* $y = f(x;\gamma)$ for "any" input x.

It is generally advisable, before using the trained model in the real-world, to evaluate its performance on a test set of training examples, where we already know the right answer, but that were not made available to the model in the training set. This is often called measuring out-of-sample performance.

Of course, what precedes is an extremely high-level overview of a very complex workflow, involving many moving pieces, where many things can go wrong. For a detailed overview of how ML models are built, evaluated and used, we refer to a classic textbook such as [1].

Here, we list some of the critical pitfalls of the ML approach:

Underfitting happens when the model is not rich enough to correctly represent the data. For example, a linear model of the form $y = \gamma_1 x + \gamma_2$ cannot represent a nonlinear relationship between the inputs and the outputs. This results in a *bias* in the model's predictions. Bias is resolved by increasing the *capacity* of the model, for example, a polynomial of a higher degree r: $y = \sum_{k=0}^{r} \gamma_k x^k$.

Overfitting happens when the model tries to hard to fit the particular fluctuations of the dataset. For example, a polynomial model of degree r can exactly fit a dataset of $r+1$ points. The resulting function will oscillate wildly and fail to generalize to unseen inputs. This results in high *variance* in the model's predictions, in the sense that those predictions are highly sensitive to the specific choice of the training examples. Variance is ideally resolved by increasing the size of the dataset, or otherwise by reducing the capacity of the model (in our example, reducing degree r).

Extrapolation happens when the trained model is applied on new inputs from a different distribution than the training set. For instance, if the training set contains input values in $(0, 1)$, the ML model will perform (very) poorly on a new input of (say) 2. It is critical to adjust the training set to correctly represent the data for which the model will eventually be used in the real world.

7.2 Artificial Neural Networks (ANN)

We use a specific family of ML models called artificial neural networks. ANN come in many different shapes to best serve specific use cases, such as convolutional neural networks for computer vision, or recurrent neural networks for natural language processing. We only use the simplest kind of ANN, called feedforward networks. Those ANN split the computation $y = f(x)$ in a number $L+2$ of layers. Each layer x_l is a vector of dimension n_l. The first layer x_0 is the input layer: $x_0 = x$ and $n_0 = n$, the dimension of the input x. The last layer $l = L+1$ is the output layer: $x_{L+1} = y$ and $n_{L+1} = p$, the dimension of the output y. Intermediate layers $x_1, ..., x_L$ are *hidden* layers, computed by sequential equations:

$$x_{l+1} = g(w_{l+1} x_l + b_{l+1})$$

where w_{l+1} is a n_{l+1} by n_l matrix of *weights*, and b_{l+1} is a vector of *biases* in dimension n_{l+1}, and the nonlinear *activation* function g is applied to the resulting vector, coefficient-wise. Common activation functions are $relu(x) = max(0, x)$, $sigmoid(x) = 1/(1 + e^{-x})$ and its integral $softplus(x) = \int_{-\infty}^{x} sigmoid(u)du = \log(1 + e^x)$.

The *learnable parameters* γ of the ANN are the collection of all weights w_l and biases b_l, which we collectively denote w. Hence, ANN have many parameters. Our simple estimation ANNs have tens of thousands of weights, chat-GPT4 famously has *trillions*. With so many parameters, ANNs are able to represent a wide variety of functions. This is formalized in the celebrated Universal Representation Theorem.

Theorem (Universal representation property of ANN)**.** *A relu (or softplus) network with a single hidden layer can approximate any function to arbitrary accuracy as the size n_1 of the layer grows to infinity.*

Why then do we need multiple hidden layers? This is a question of computation efficiency: ANNs with multiple small layers are smaller and faster than ANNs with single large layers, with similar representation capacity.

The capacity of an ANN is a direct function of the number and size of hidden layers. Capacity is reduced or increased by fine-tuning those sizes, generally called the *architecture* of the ANN.

ANNs are trained by gradient descent or variants thereof, which are the only known optimization algorithms tractable with so many optimization variables. At each step of the algorithm, the gradient of the objective function (MSE between predictions and targets) is computed, and the weights are moved by a small step, called *learning rate*, in the direction opposite to the gradient:

$$w \leftarrow w - \frac{\partial ||f(x;\theta) - y||^2}{\partial w} lr$$

Gradient descent is guaranteed to converge to a local minimum as long as learning rate appropriately decreases between iterations. It also requires the computation of a very high dimensional gradient at every step. The triumph of Deep Learning was the discovery of an extremely efficient algorithm, called *backpropagation*, for the computation of those gradients.

It takes a lot of hard and specialized work to train ANNs in reasonable time. Thankfully, some public platforms, such as Google's TensorFlow, provide simple and intuitive means of defining, training, evaluating and working with ANNs, while hiding complexities behind the scene. For example, they compute gradients with backpropagation automatically. We use TensorFlow, with inspiration from Geron's classic textbook [5].

7.3 Estimation with ML and ANN

Given this brief introduction to Machine Learning, its application to estimation falls in place naturally. Recall that an estimator is a function from $\mathbb{R}^n$ into $\mathbb{R}^p$ with performance measured by MSE. Instead of coding logic such as MLE in an estimation function, we can attempt to *learn* this function from a simulated dataset where inputs are samples, simulated with known parameters, and outputs are these parameters. More specifically:

Algorithm (Estimation by Machine Learning)**.** *To learn an estimation function g to estimate the parameters $\theta \in \mathbb{R}^p$ of a distribution D from a sample of size n:*

1. *Simulate a training set of m examples with inputs $x_i \in \mathbb{R}^n$ and outputs $y_i \in \mathbb{R}^p$. For each example i:*
   (a) *Draw parameters θ from some distribution, which choice requires careful thought: this distribution should represent the set of parameters in the planned application of the trained ANN.*
   (b) *Simulate a sample s of n independent draws of distribution D with parameters θ.*
   (c) *Set input $x_i = s$ and target $y_i = \theta$, thereby reversing the relationship between parameters and samples.*

2. *Train a ML model on this training set, by minimization of the MSE between predictions and training targets. Denote $\gamma*$ the optimal learnable parameters.*

3. *Evaluate the performance on new simulated examples.*

4. *Fix the learnable parameters of the ML model. The trained model is the desired estimation function g:*

$$g(x) = f(x; \gamma*)$$

7.4 Discussion

Inverse problems. Estimation is an example of an inverse problem solved by ML. Inverse problems occur when some function $y = f(x)$ is easy to evaluate, but hard to invert. Sampling $sample = f(parameters)$ is easy. Estimation $parameters = f^{-1}(sample)$ is hard.

Inverse problems are well suited to ML. Since f is easy, we can easily simulate a training set by drawing x and computing $y = f(x)$, and reverse the relationship in the dataset: inputs are $f(x)$ and outputs are x. A ML model trained on this dataset should learn the inverse function, thereby providing, after training, an easy evaluation of $f^{-1}(y)$ for all y within the sampling distribution of the training set.

Since training data is simulated, we are not limited by the availability of data in the real world. We can simulate trillions of examples if we want. We are not even limited by memory or storage space, since we can simulate training examples on the fly, during training. All we need is patience, and we work with virtually infinite datasets. Therefore, inverse problems are generally immune to overfitting. Underfitting is easier to manage: we simply increase capacity until we are happy with out of sample performance.

Extrapolation requires much more care: the trained ML model will only correctly predict parameters in the range of its training set. The resulting estimation function is only valid on the corresponding domain. This being said, our special neural network, introduced next, realizes the miracle of correctly extrapolating outside of the distribution of the dataset. For example, when trained for the exponential distribution on a dataset where ground truth rates are picked from their own $Exp(1)$ distribution, the ANN still correctly predicts rates of order 5 to 7, up to 6 standard deviations away from the distribution of the training set!

Nondeterminism. Another difficulty is that samples are not deterministic functions of parameters. They also depend on the randomness that realized when we simulated the samples for our training set. We expect the ML model to cut through this randomness and correctly interpret the signal encoded in the sample. ML models, in principle, can learn to do that, given enough capacity and training data. But in practice, it takes a lot of trial and error to achieve good results with nondeterministic inputs: experiment with ANN architecture, weight initialization, training epochs, batches per epoch and batch size, learning rate schedule, etc. ML, and Deep Learning in particular, is not always based on solid mathematical grounds that could provide guidance. Rather, it is often based on heuristics and best practices found by trial and error.

To clear a possible confusion: the proposed ML workflow is *not* about learning the MLE. For an ANN to learn the MLE is a very easy problem, since the MLE is a deterministic function of the sample. But it is also futile. The only situation where we could train the ANN on the MLE is when the MLE is available in the first place, and the whole point of the ML machinery is to provide a tractable solution for distributions where MLE is not easily available.

Hence, the MLE is not used in the dataset, and the ANN doesn't know anything about it. What it is trying to learn, is to estimate the true, generative parameters that produced a limited, noisy sample, from the observation of the sample and nothing else. And this is a very hard problem, for human and artificial intelligence alike. The ANN is not trying to reproduce the MLE function, which it doesn't know. It is trying to produce a good estimation function, just like humans designed the MLE machinery for the same purpose. Since the MLE is the best possible estimator, at least for large samples, the ANN will produce estimates near the MLE if it is doing a good job, without having ever seen the MLE. In some sense, the ANN tries

to "rederive" the whole machinery behind the MLE, all by itself, from observation alone. This is legitimate artificial intelligence.

There is surprisingly little literature on the subject of estimation by ML, knowing the attention given to ML, the importance of the estimation problem in Statistics, the difficulty or impossibility to compute MLE for complicated distribution, and how natural this application appears once stated. This being said, this application of ML to Statistics is not a completely new either: a similar methodology using convolutional networks was proposed by [7] in 2023, and applied to max-stable distributions. To the best of our knowledge, it was never tested on exponential or exponential mixture distributions, or more complicated PH distributions.

Our purpose is to apply and evaluate this machinery on several families of increasingly complicated distributions, starting with a simple exponential, and culminating into a general PH distribution (although this is a topic for further research: this thesis stopped at mixtures of exponentials). For each case, we provide Python code and show (out of sample) performance results.

Chapter 8

Deep sets

This chapter introduces a special ANN architecture, particularly well suited to the estimation problem, inspired by the famous Deep Sets paper [9], and explains why this is, in our opinion, a very good design for an estimation network.

8.1 Problem statement

The estimation by ML (EML) machinery outlined in the previous introduction: simulate a dataset of training examples by picking parameters and simulating samples, then train a feedforward ANN to learn the parameters from the samples, is not entirely satisfactory, both in terms of scientific rigor, or in terms of practical results. In our experiments, ANNs trained on samples of some fixed size n only provided decent estimations for samples of the same size. And even in this case, performance was noticeably worse than MLE.

In order to compete with MLE and be considered a legitimate estimation technique, EML should satisfy at least two requirements, listed here in increasing order of difficulty, and urgency.

8.1.1 Permutation-invariant estimation

ANNs work with vectors, which are ordered collections. In a sample $(X_1, X_2, ..., X_n)$, X_1 and X_2 have different semantics and are not interchangeable. Every unit in the next layer is linked to each observation with different weights. Shuffling the sample therefore changes the response of the network.

Statistical samples, on the other hand, are not ordered. All the observations in the sample are independent draws from the same distribution. Their order has no meaningful semantics. Permuting the sample doesn't change anything. The samples (X_1, X_2, X_3) and (X_2, X_3, X_1) are the same, and estimations from those two samples should be identical. In other terms, samples are not vectors, they are sets, *unordered* collections of observations. There is no such thing as a sample (X_1, X_2, X_3). The correct notation for a sample is $\{X_1, X_2, X_3\}$

Therefore, estimators should be permutation-invariant. Notice, that the MLE estimators for the exponential distribution (inverse sample mean) and normal distribution (sample mean and variance) are permutation-invariant. Permutation invariance is an important quality of good estimators, and we want EMLs with this property: to get a different answer by shuffling the sample is at best suspicious.

This is not a difficult problem to solve on its own. A classic feedforward network will eventually learn an almost permutation-invariant estimation, having been exposed to a large number of sample permutations with identical labels during training. But this is immensely wasteful. An ANN with input size $1,000$ and 64 units in its first hidden layer has $64,000$ connection weights between the two layers. It only needs 64.

In order to achieve permutation invariance, the $1,000$ weights from every hidden unit to the $1,000$ inputs must be identical. The ANN will eventually learn to set weights this way, but this is a terrible waste of computation resources. It may also interfere with training, distracting the ANN from learning the predictive patterns in the sample.

Another simple solution is to present samples to the ANN in a canonical form, for instance, ordered in increasing order. This certainly achieves permutation-invariance, but also, further specializes the ANN for a given sample size: the smallest element doesn't have the same distribution in a sample of 32 or 8192 observations.

Estimation is not the only problem where permutation invariance is desirable, and there exists a lot of research on the topic. In particular, the famous paper Deep Sets [9] developed a remarkably elegant and effective solution, a special ANN architecture called a deep set, permutation-invariant by construction, and provably a universal approximator for all permutation-invariant functions.

Although deep sets were developed for the purpose of permutation invariance, it so happens that they also provide a solution to the much harder problem of 'size invariance'.

8.1.2 Estimation with samples of arbitrary size

We highlighted earlier that MLE estimates for exponential rate or normal mean and variance are permutation-invariant. They are also, in some sense, 'size-invariant', because the respective formulas $1/\overline{X}$, $\overline{X}$ and $\overline{X^2}-\overline{X}^2$ do not depend on size, except for averaging. We are going to see that those two properties are intimately related, and that all permutation-invariant functions also carry this 'size invariance' property.

ANNs, on the other hand, are generally trained on inputs of fixed size, and specialized for that size. To train an ANN capable of making estimations with samples of arbitrary size has been the hardest challenge in this project, to the extent that a satisfactory solution was only uncovered in the last weeks, with exciting ramifications yet to be explored, and a fair amount of fine-tuning to complete.

The technical problem of feeding arbitrary-sized inputs to an ANN is an implementation detail, and it can be overcome. The difficulty is to produce an estimator that perform as expected with varying sample size: bias should not depend on size (and hopefully remain very small) while standard error should decrease with size, roughly in inverse proportion to its square root.

In particular, quick fixes such as subsampling or cropping will not fly. Say, the ANN was trained on samples of size 1,000. If you want an estimation based on a smaller sample of size 100, subsampling consists in completing the sample by randomly picking 900 additional observations among the 100 existing ones. This method loses the critical independence property among observations, and results in biased estimates. If you want an estimation based on a larger sample of $10,000$ observations, you could crop it and only pick $1,000$ observations. Evidently, you would waste valuable information and couldn't possibly expect standard error to drop.

More intelligently, you could split your sample into 10 samples of $1,000$ observations each, make 10 predictions and average them. The standard error effectively drops by a factor $\sqrt{1/10}$. The architecture we used for a long time was based on this observation. We trained an ANN on small samples of size 32, and used them to make estimates with samples of 'almost' arbitrary size, as long as it was a multiple of 32[1], by splitting the sample into k samples of 32 observations, making k predictions with the ANN, and averaging them into a final prediction. We abandoned this solution recently, for the reason that it results in biased estimates.

To see this clearly, consider the rate estimate for the exponential distribution. We know the MLE is $1/\overline{X}$. Suppose the ANN trained on small samples of size 32 learns the MLE, and then makes an estimation with a larger sample of size 512. It splits the large sample into 16 samples $X^{(k)}$ of 32 observations each, produces 16 predictions, each one the inverse mean of the corresponding small sample, $1/\overline{X^{(k)}}$, and averages them to produce a final estimate $\overline{1/\overline{X^{(k)}}}$. This is evidently different from the MLE $1/\overline{X}$, the inverse mean of the large sample, and a biased estimate, since $E[1/X] \neq 1/E[X]$. To fix ideas, say that the true rate is 1, and

[1] or dropping the reminder

you have a sample of 64 observations, split into 2 samples with respective sample means 1.2 and 0.8. Then, the mean of the large sample is 1, hence an unbiased estimate is $1/1 = 1$. The first sample gives a prediction of $1/1.2 \approx 0.83$, and the second one $1/0.8 \approx 1.25$. The mean is around 1.02, with 0.02 bias. It may be small enough to remain undetectable among the high variance of small size estimations, but it becomes very visible for large sizes of $8,192$ and beyond. Besides, this method is fundamentally flawed and scientifically incorrect.

We only recently realized that a correct implementation of a deep set resolves this difficult challenge in an elegant and effective manner.

8.2 Deep Sets: theorem

Deep sets are based on the following theorem, showing that all permutation-invariant functions are decomposed into embedding, aggregation and prediction functions.

Theorem (Deep Sets)**.** *Any permutation-invariant function g from $\mathbb{R}^n$ to $\mathbb{R}^p$ can be written as:*

$$g(x) = h\left[\frac{1}{n}\sum_{i=1}^{n} e(x_i)\right]$$

where:

- *e is a* scalar *function, called* embedding function*, from $\mathbb{R}$ into some embedding space E. In theory, the embedding space may be of infinite dimension. In practice, we work with $E = \mathbb{R}^q$, where q is the embedding size. It helps to think of e as a collection of q scalar functions e_j from $\mathbb{R}$ to $\mathbb{R}$. The same q embeddings are applied to the n observations x_i.*
- *the q embeddings of the n observations x_i are averaged across observations, so that the aggregated embedding $\overline{e(xi)}$ is one vector of size q.*
- *the prediction function h from E (in practice: $\mathbb{R}^q$) to $\mathbb{R}^p$ makes a prediction only based on the aggregated embedding.*

It is clear that all functions of the stated form are permutation-invariant, but much less clear that all permutation-invariant functions are of this form.

What about the product function $g(x) = \prod_{i=1}^{n} x_i$? It is obviously permutation-invariant. Can we represent it in the form of the theorem? The answer is yes, of course, although it may not be visible at first glance. Just set $e(x) = \log x$ and $h(y) = e^{ny}$. What about the maximum function? Set $e(x) = e^{\alpha x}$ and $h(y) = \log(ny)/\alpha$. When α grows to infinity, the average $\overline{e(x_i)}$ converges to $e^{\alpha \max(x)}/n$, as the other terms in the sum become negligible, so $g(x)$ effectively represents the maximum function.

More to the point, let us quickly verify that MLEs for exponential and normal distributions follow this pattern. For the exponential distribution, the MLE is $1/\overline{X}$. It only needs one embedding, and it is identity: $e(x) = x$. Hence, the aggregated embedding is $\overline{e(x_i)} = \overline{x}$. The prediction function is the inverse: $h(y) = 1/y$.

The normal distribution needs 2 embeddings to represent the MLEs $\overline{X}$ and $\overline{X^2} - \overline{X}^2$: $e(x) = (x, x^2)$, resulting in aggregates $\overline{e(x_i)} = (\overline{x}, \overline{x^2})$. The prediction function is $h(y_1, y_2) = (y_1, y_2 - y_1^2)$.

More generally, we could directly feed q aggregated embeddings to a prediction function, represented, for example, by a neural network. One choice could be to use $e_j(x) = x^j$, thereby feeding empirical moments. Or $e_j(x) = \mathbb{1}_{\{x<a_j\}}$, feeding an empirical CPD. While these choices make sense, they remain somewhat arbitrary: should we use moments? or CPD? or both? or something else? and should we make different choices for different distributions? It is generally best letting the ANN figure out the best predictive embeddings, for a given problem. This observations leads to the following ANN architecture.

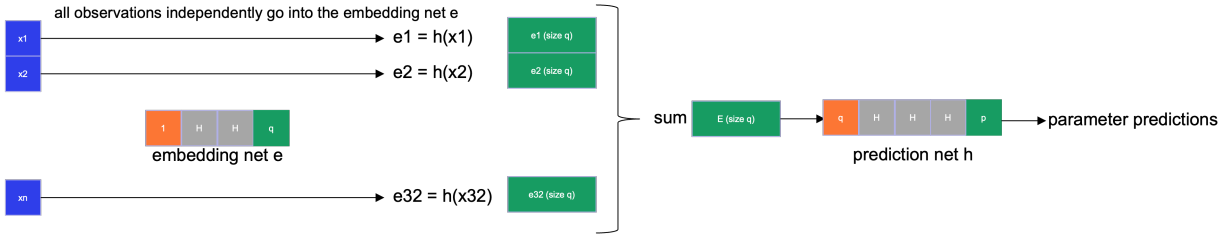

Figure 8.1: Illustration of the deep set architecture

8.3 Deep Sets: architecture

The previous theorem directly provides an ANN architecture, where the embedding function and the prediction function are represented by feedforward neural networks, combined in a deep set.

The embedding network has input of size 1, one observation, and output of size q, the q embeddings. The prediction network has input of size q, the q aggregated embeddings, and output of size p, the predictions for the p parameters. The main ANN is a sequence that accepts samples of arbitrary size for inputs, splits them into observations, feeds all observations to the embedding network, aggregates results, and feeds them to the prediction network. This is illustrated in figure 8.1. The only trainable pieces, with learnable connection weights, are the embedding network and the prediction network.

The following listing implements this exact architecture in TensorFlow.

```
# network
embedding_size = 64
activation = "softplus"

def split_sample_into_observations(sample):
  return tf.expand_dims(sample, axis=-1)

def aggregate_embeddings(embeddings):
  # embeddings shape is [batch_size, num_samples, sample_size, embedding_size]
  # aggregation is across observations in the sample, hence axis -2
  means = tf.reduce_mean(embeddings, axis=-2)
  # [batch_size, num_samples, embedding_size]
  return means

embedding_network = Sequential([
    # input shape = [batch_size, num_samples, sample_size]
    # split subsamples into scalars
    Lambda(split_sample_into_observations), # [batch_size, nun_samples, sample_size, 1]
    # all embeddings go through embedding net
    # hidden layers
    Dense(embedding_size, activation=activation),
    Dense(embedding_size, activation=activation),
    # embedding layer: [batch_size, num_samples, sample_size, embedding_size]
    Dense(embedding_size),
])

aggregation = Lambda(aggregate_embeddings)

prediction_network = Sequential([
    # input shape = [batch_size, num_samples, embedding_size]
    # hidden layers
    Dense(embedding_size, activation=activation),
    Dense(embedding_size, activation=activation),
    Dense(embedding_size, activation=activation),
    # output layer, shape [batch_size, num_samples, par_size]
    Dense(par_size),
    Lambda(activate_predictions),
])
```

```
39
40 model = Sequential([
41     # input shape = [(batch_size), num_samples, sample_size]
42     Input([None, None]),
43     # normalization by mean and standard dev of xs
44     Normalization(axis=None, mean=mean_xs, variance=var_xs),
45     # deep set
46     embedding_network, # [(batch_size), num_samples, sample_size, embedding_size]
47     aggregation, # [(batch_size), num_samples, embedding_size]
48     prediction_network, # [(batch_size), num_samples, par_size]
49 ])
```

We use softplus for activation, after also experimenting with relu and sigmoid. Softplus gave us (marginally) better results for the mixture of two exponentials.

A few other choices had to be made: the size q of the embeddings, and the number and sizes of the hidden layers of the embedding and prediction networks. We followed the following methodology. We used the most complex distribution in our tool set, in this case, the mixture of two exponentials. We started with small networks with 8 embeddings, and one hidden layer of 8 units for both the embedding and the prediction network. We measured performance (we explain how a few pages below) and increased sizes until performance stopped increasing. We ended-up with an embedding size of $q = 64$, two hidden layers for the embedding network and three for the prediction network, all of size 64. The reasoning is that, if this architecture is sufficient for our most complicated distribution, it should work with simpler distributions too (although, often an overkill). Our numerical results did verify this line of thought.

8.4 Deep Sets: benefits

The benefits of switching to deep sets have been significant, to the extent that it turned the project around, at a late stage. We obtained, by far, the best performance measurements, and even witnessed unexpected results, such as the capacity to correctly extrapolate to parameter magnitudes unseen in the simulated training set. We switched to deep sets very late in the project, so there are unexplored ramifications for further research. For example, it is probably worth training with samples of different sizes, to help the ANN learn that 'size doesn't matter'.

Permutation invariance Deep sets were built for the purpose of permutation invariance, and they certainly deliver. Not only are they permutation-invariant by construction, they are smaller and leaner than the previous ANNs we experimented with, they train faster and better, presumably because they are not confused by permutations and focus on predictive patterns.

Correct predictions with samples of arbitrary size By far the most critical benefit of deep sets is to finally provide estimators that accept samples of arbitrary size, and exhibit the correct behavior, whereby larger sizes reduce standard error without bias.

Let us verify that this is the case with the simple example of the exponential distribution. Suppose that the ANN is trained on samples of size 256, such that the embedding network correctly learns the identity function, and the prediction network correctly learns the inverse function. Then, the ANN will always predict $1/\overline{X}$, the MLE, irrespective of sample size. And we will see that this is exactly what happens in our numerical results.

What is truly remarkable is that the ANN effectively learns a correct formula. Once again, the ANN is not trained to reproduce the MLE, it doesn't know what MLE is and it has never seen one. All it sees are examples of noisy samples, along with labels of generative parameters. If the ANN learns the MLE formula, it kind of deduced it by itself.

Also, note that the averaging of the embeddings mechanically reduces standard error with larger samples.

Correct extrapolation to parameters of unseen magnitude If the ANN effectively learns a formula, it is capable of correctly extrapolating outside of the training distribution. Suppose it learns the MLE for the exponential distribution $\hat{\lambda} = 1/\overline{X}$ on a training set where rates are picked uniformly between 0 and 1. Then, incredibly, it will correctly predict the rate from a sample generated with a rate of 10. We are not reaching this far in practice, but we are going to see that a deep set trained on rates picked from en Exp(1) distribution, makes correct predictions for rates up to 5 or even 7, 6 standard deviations away from the training set!

Let us now conclude with numerical results, after a brief description of the training and evaluation methodology.

Chapter 9

Numerical results

9.1 General methodology

9.1.1 Training and estimation

Estimation with the trained network is straightforward: feed a sample of arbitrary size and obtain parameter estimates.

Training is slightly less conventional: instead of feeding one sample for each training example and obtain a squared error, we wanted to get an estimate of bias and variance so as to tweak training towards more desirable results. In particular, we want to express a preference for unbiased estimators, that is, strongly penalize bias during training. To do this, we need to somehow estimate bias and variance.

For this reason, we feed *multiple* samples, generated with the same parameters, for each training example in the training loop. We obtain multiple predictions, which allows us to estimate bias (average of predictions minus ground truth) and variance (of predictions). The (custom) loss function aggregates the two into the per-example loss: $loss = \alpha.bias^2 + (1 - \alpha).variance$. With $\alpha = 0.5$, this is just the MSE. We usually set α higher, around 0.9.

For complex distributions such as mixtures, training is more effective with larger samples. And we need many samples to correctly estimate bias and variance. We settled on 128 samples of size 256 for GPU RAM constraints, which seems to work well in all cases.

For details of normalization, batching, learning schedule, etc. we refer to the source code in our GitHub repo https://github.com/simonsavine/phasetype. The EML notebooks start with "ann_" and are mostly similar, with the exception of simulation routines, in the beginning of the notebook.

The batch size, in particular, the number of examples in a batch that computes the gradient descent step, strongly affects results. All those things: bias weight, number of batches per epoch, number of examples per bacth, number and size of training samples, learning rate schedule, etc. are presently set by hand, distribution by distribution, with a sort of manual gradient descent: change one thing at a time, retrain, and revaluate until maximum performance is reached, and repeat with the next setting. We have been slowly converging to some numbers that seem to work well in all cases, but this is insufficient. Some tinkering is still necessary every time we introduce a new distribution. One development we need is an automated version of this process.

9.1.2 Performance evaluation

After the ANN is trained for a given distribution, it is fixed (and saved on drive) and we evaluate its performance. Standard out-of-sample evaluation is superfluous. This is generally performed by evaluation

of the loss function on a so-called validation set, a batch of data set aside and not used in training. But our training loop generates new data on the fly, such that the training set constantly validates itself.

Instead, we (manually) pick a number of configurations, which are particular assignments of parameters. For instance, for the exponential distribution, we picked 12 configurations with $\lambda = 0.2, 0.4, ..., 2.4$. For each configuration, we evaluate performance on small (128), medium (1,024) and large (8,192) samples. Hence, we perform (in this case) $12 \times 3 = 36$ separate evaluations.

For each evaluation, we independently simulate 64 samples of the appropriate size, all from the same distribution dictated by the configuration, and make 64 predictions with the trained ANN. We produce a graph of the results, where the horizontal axis consists of the 64 experiments, and the vertical axis is where we show the predictions of the ANN, along with the ground truth parameters. We also produce a histogram of the 64 predictions, and compute the bias, standard deviation and RMSE, estimated with the the 64 predictions.

Finally, we compare the predictions with alternative estimations from the same 64 samples. For the exponential and normal distributions, we simply used the MLE. For Erlang and gamma distributions, we used scipy.stats' gamma.fit() function. For mixtures of exponentials, we used an external implementation of the EM algorithm. This is only for comparison, and the only place where those numbers are computed is for the plot. Those estimates are not used anywhere else in the code, and the ANN never sees them.

For example, figure 9.1 shows the results for the exponential distribution, with two configurations, where the rate is set at 0.2 and 0.6, and sample size is 1,024. Recall that the ANN was trained on samples of size 256.

9.1.3 Distribution-agnostic software

In terms of how the code is designed, we prefer a distribution-agnostic machinery. Distributions should be plug and play. Code the appropriate simulation routines, and leave the rest to the notebook, without modification. This is an implementation detail rather than a scientific requirement, but a rather important one. If the machinery must be redesigned from the ground up for every different distribution, it doesn't fulfill one of its key promises, to do away with hard work on a distribution-by-distribution basis.

This goal was almost achieved. The following listing shows the beginning of our notebook for exponential distributions, ann_exp_final.ipynb. This part defines the simulation routines. The rest of the notebook is independent of distributions and (in principle) unchanged from one distribution to the next.

```
# useful definitions
par_names = ["rate"]
par_size = len(par_names)

# dataset generation

def generate_parameters(num_examples):

  uniform = tf.linspace(np.float32(0), 1, num_examples+2)[1:-1]
  uniform = tf.random.shuffle(uniform)

  # rates = 0.01 + exp(0.8)
  rates = 0.01 - tf.math.log(1. - uniform) / 0.8

  # num_examples x par_size
  return tf.reshape(rates, [num_examples, par_size])


def generate_samples(parameters, sample_size):

  num_examples = parameters.shape[0]
  rates = parameters

  return - tf.math.log(1. -tf.random.uniform([num_examples, sample_size])) / rates

  return samples

```

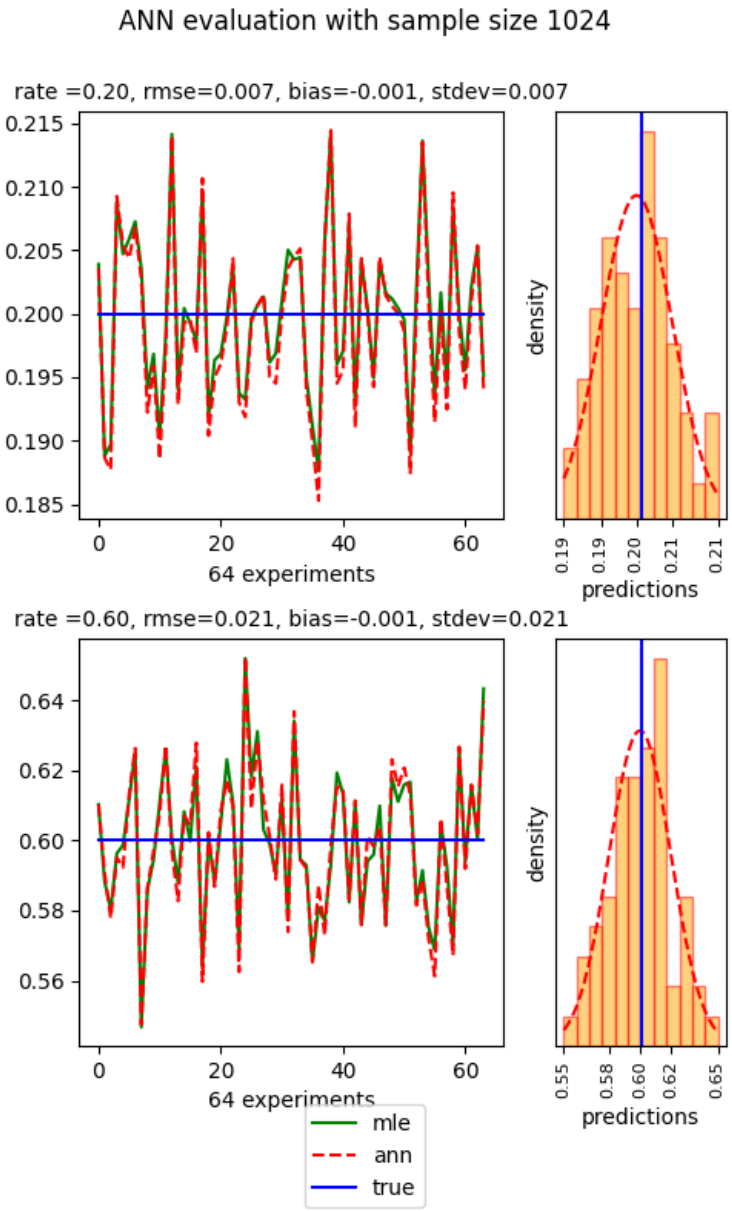

Figure 9.1: (Partial) evaluation results for the trained exponential estimator. Complete results follow. This is mainly to illustrate evaluation methodology.

```
28
29  # make sure average weight is 1, i.e. normalize all weights by the mean
30  # no weights --> return tf.ones([parameters.shape[0], 1])
31  def generate_weights(parameters, samples):
32    return tf.ones([parameters.shape[0], 1])
33
34
35  def generate_dataset(num_examples, sample_size):
36    parameters = generate_parameters(num_examples)
37    samples = generate_samples(parameters, sample_size)
38    weights = generate_weights(parameters, samples)
39
40    return samples, parameters, weights
41
42
43  # activation for parameter prediction
44  def activate_predictions(predictions):
45    return predictions
```

The function activate_predictions() gives the option to apply a transformation to the predicted parameters. For instance, we can apply a relu or softplus activation to force positive predictions. We didn't find it necessary for exponential rates. The ANN never returned a negative number as is. On the other hand, for mixtures, we do want positive probabilities that sum to one, which is easily achieved by activating probability predictions with softmax:

$$softmax(y) = \left(\frac{e^{y_i}}{\sum_j e^{y_j}} \right)$$

Here is the listing for mixtures of two distributions:

```
1  # activation for parameter prediction
2  def activate_predictions(predictions):
3    # split into probas and rates predictions
4    predicted_probs, predicted_rates = tf.split(predictions, num_or_size_splits=2, axis=-1)
5    # activate
6    activated_probs = tf.nn.softmax(predicted_probs, axis=-1)
7    activated_rates = predicted_rates
8    # concat back
9    return tf.concat([activated_probs, activated_rates], axis=-1)
```

We wrote that the goal of a distribution-agnostic software was 'almost' achieved because some details in the network architecture, loss function, training loop, and evaluation routine, still need manual fine-tuning when new distributions are injected. This is onee reason why, for instance, we couldn't find the time to extend to mixtures of 3 or more exponentials, and had to stop at 2 for now.

9.2 First proof of concept: exponential distribution

The proof of concept for the exponential distribution is found in the notebook ann_exp_final.ipynb in our GitHub repo https://github.com/simonsavine/phasetype. The code exactly implements the described methodology. The rate is picked from an Exp(0.8) distribution, then, samples are generated by inversion of the exponential CPD. Details such as normalization or leraning schedule are easily found in the notebook.

We use MLE as a comparison. This is simply the inverse sample mean.

Complete evaluation results are found on the notebook. The following figures show some partial results. Performance is excellent, the ANN matches MLE within a small, virtually invisible, numerical error.

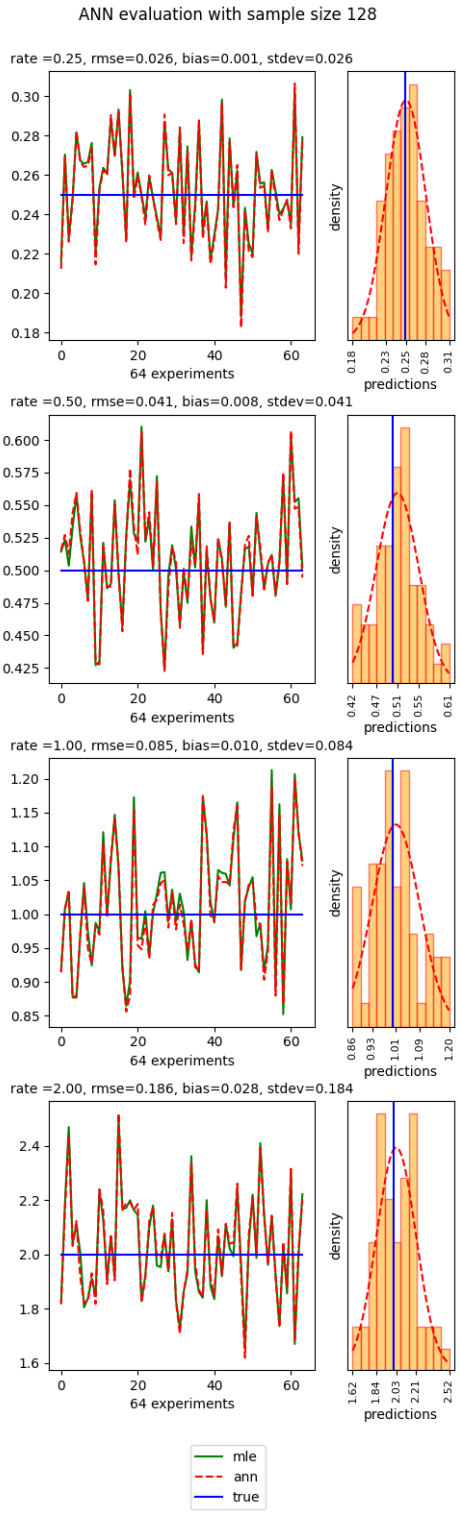

Figure 9.2: Evaluation results for the exponential distribution, with sample size 128

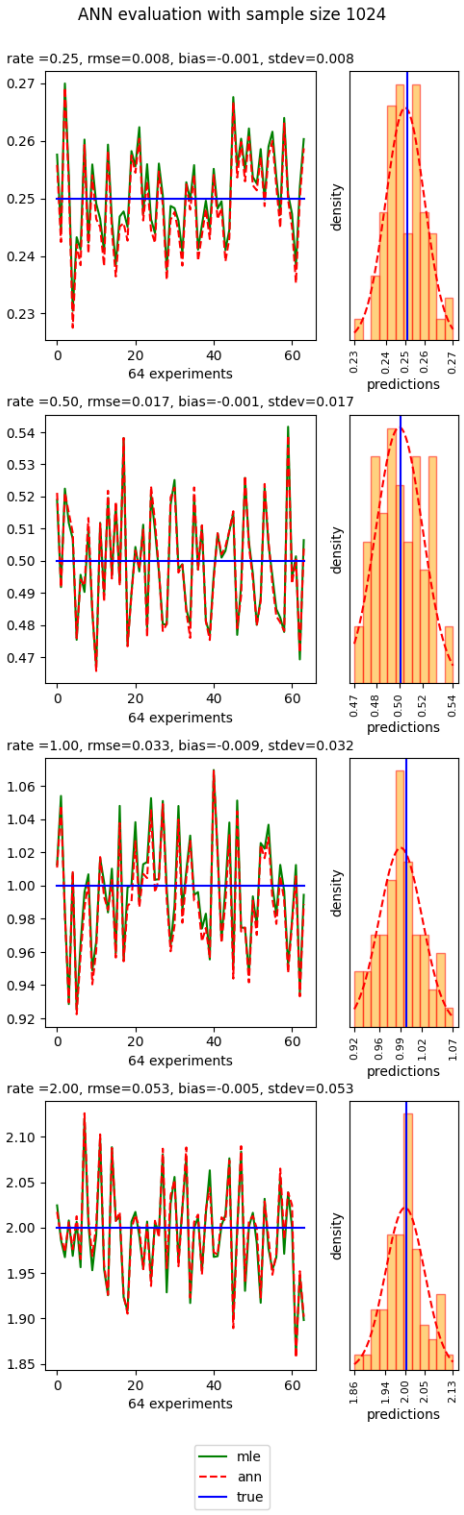

Figure 9.3: Evaluation results for the exponential distribution, with sample size $1,024$

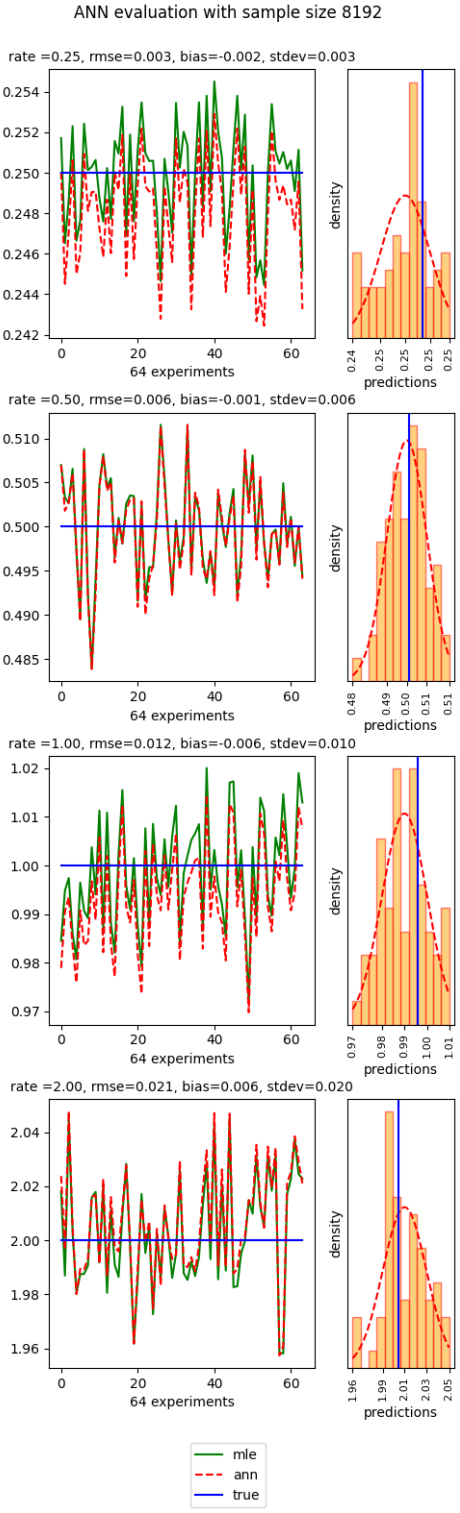

Figure 9.4: Evaluation results for the exponential distribution, with sample size $8,192$

Figure 9.5 shows test results with rates of 5 and 7, distant by up to 6 standard deviations, to illustrate the capacity of the ANN to correctly extrapolate outside of the distribution of the training set.

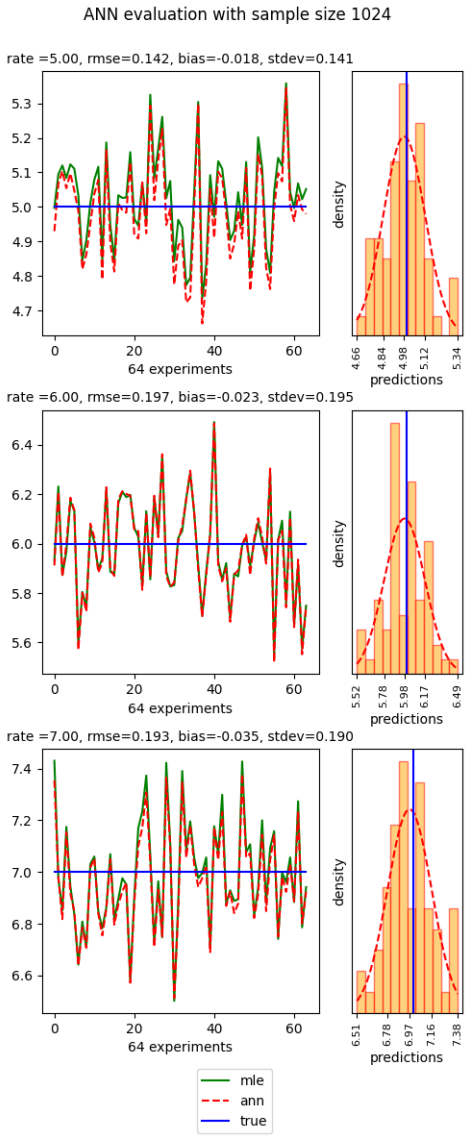

Figure 9.5: Evaluation results for the exponential distribution, with rates outside of training range. Sample size is $1,024$.

We can easily verify that the ANN, indeed, re-derived the MLE, and encodes the formula $g(X) = 1/\overline{X}$. This is performed at the bottom of the notebook. We feed the ANN with *uniform* samples of different random sizes and bounds, obtain 'estimates', and compare with the inverse mean. The curves match (within reasonable numerical error), which confirms that the ANN did, indeed, learn the MLE expression without having ever seen it or its results. The graph is displayed on figure 9.6.

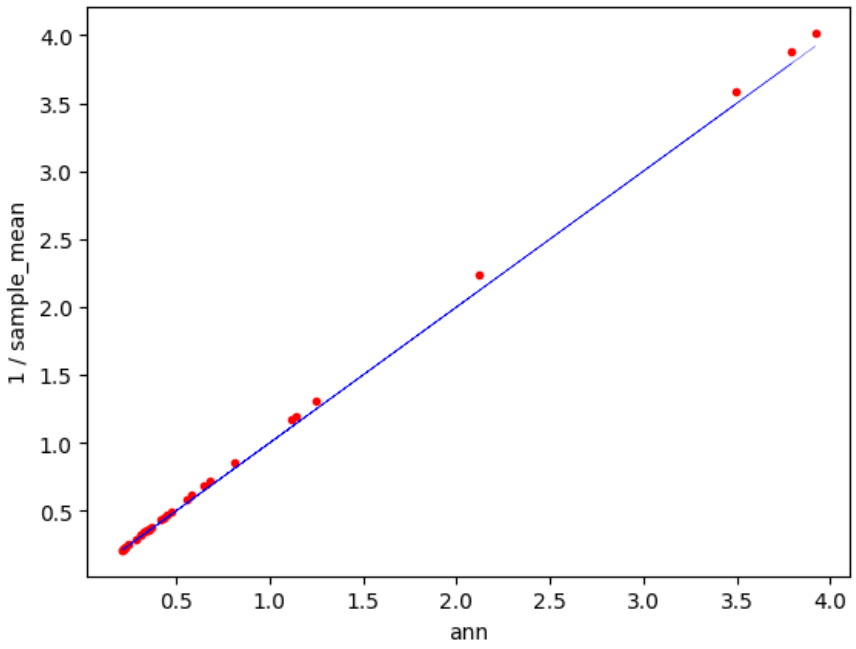

Figure 9.6: ANN 'predictions' against inverse sample mean, 'samples' are of random size and range, and filled with uniform random numbers. The figure illustrates that the ANN effectively learned the inverse mean function.

9.3 Second proof of concept: normal distribution

The normal distribution is not part of the PH family. We however found it very useful to include it in our study, because, like the exponential distribution, this is a simple distribution where we know the right estimation formula, which greatly facilitates reasoning about what the ANN could get right or wrong. The normal distribution also has two parameters, making it a natural choice for testing our environment with more than one parameter before moving on to more complicated distributions.

The proof of concept for the normal distribution is found in the notebook ann_normal_final.ipynb in our GitHub repo.

Everything is mostly identical to the exponential, except, of course, the sampling procedures, and the MLE formulas. The mean parameter is picked with its own normal distribution, with mean 0 and standard deviation 2. Standard deviation is picked from an Exp(0.8) distribution. We also made it harder for the ANN by using the standard deviation rather than the variance. Normal samples are simulated with TensorFlow's built-in function.

The only material difference is that the weight of the bias in the loss was moved from 0.9 to 0.99, otherwise the ANN produces biased estimates. This is an example of necessary per-distribution tuning that remains to be resolved.

Complete evaluation results are found on the notebook. The following figures show some partial results. Each row is a separate configuration, with mean estimates on the left, and standard deviation on the right. As for the exponential distribution, performance is excellent, the ANN matches MLE within a small numerical error.

The last example is magnitudes away from the data in the training set. Once again, the ANN performs extrapolation decently.

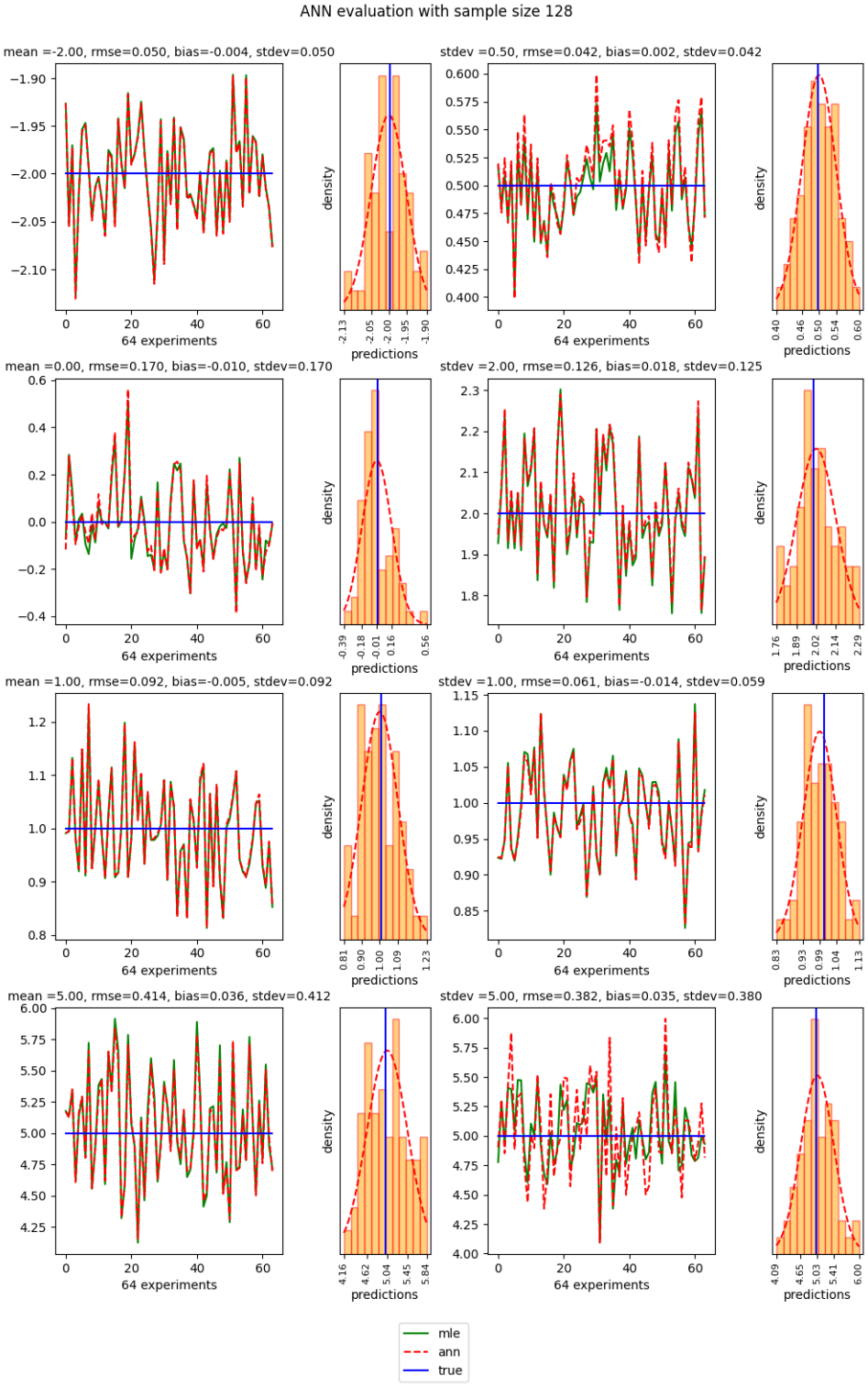

Figure 9.7: Evaluation results for the normal distribution, with sample size 128

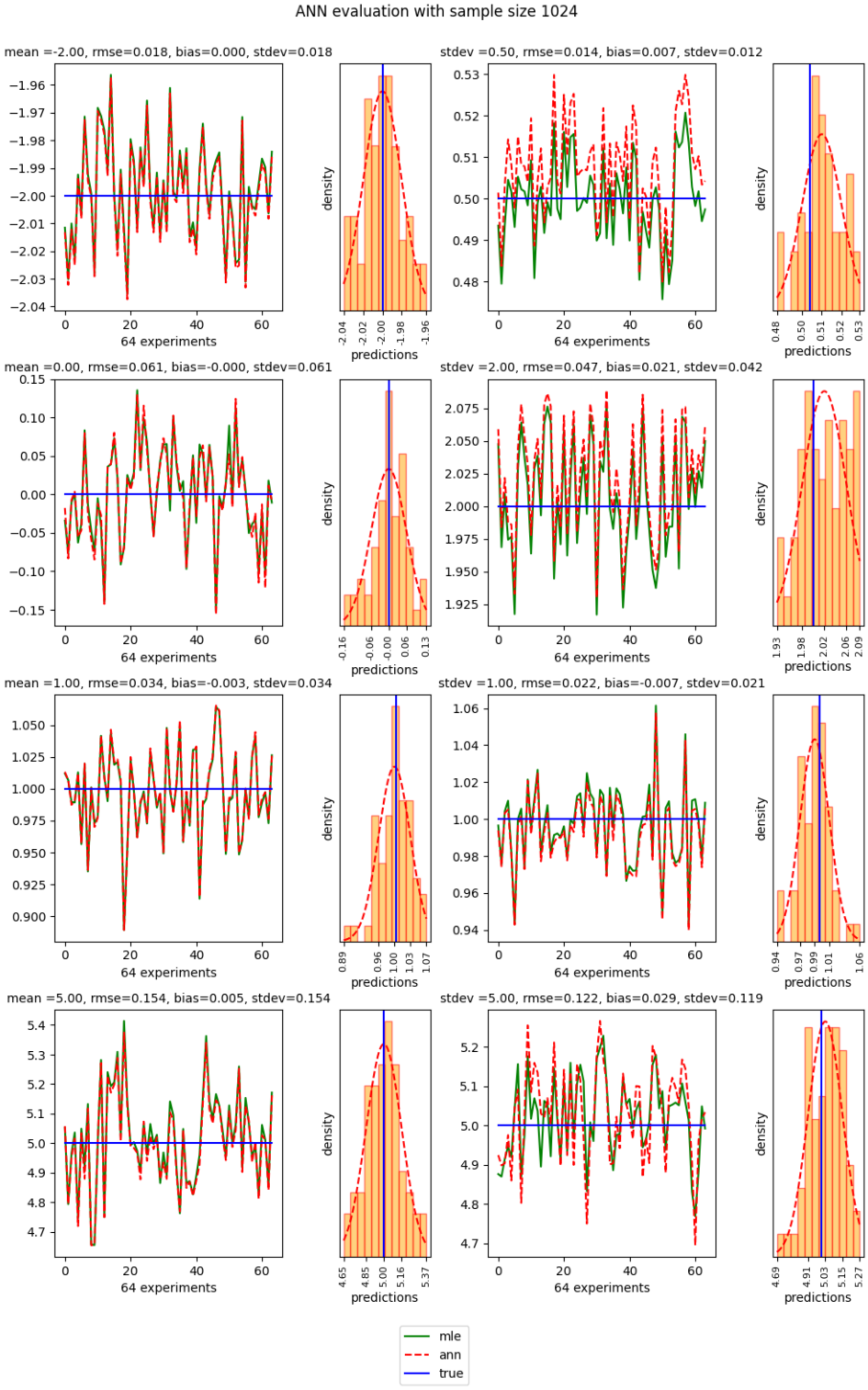

Figure 9.8: Evaluation results for the normal distribution, with sample size $1,024$

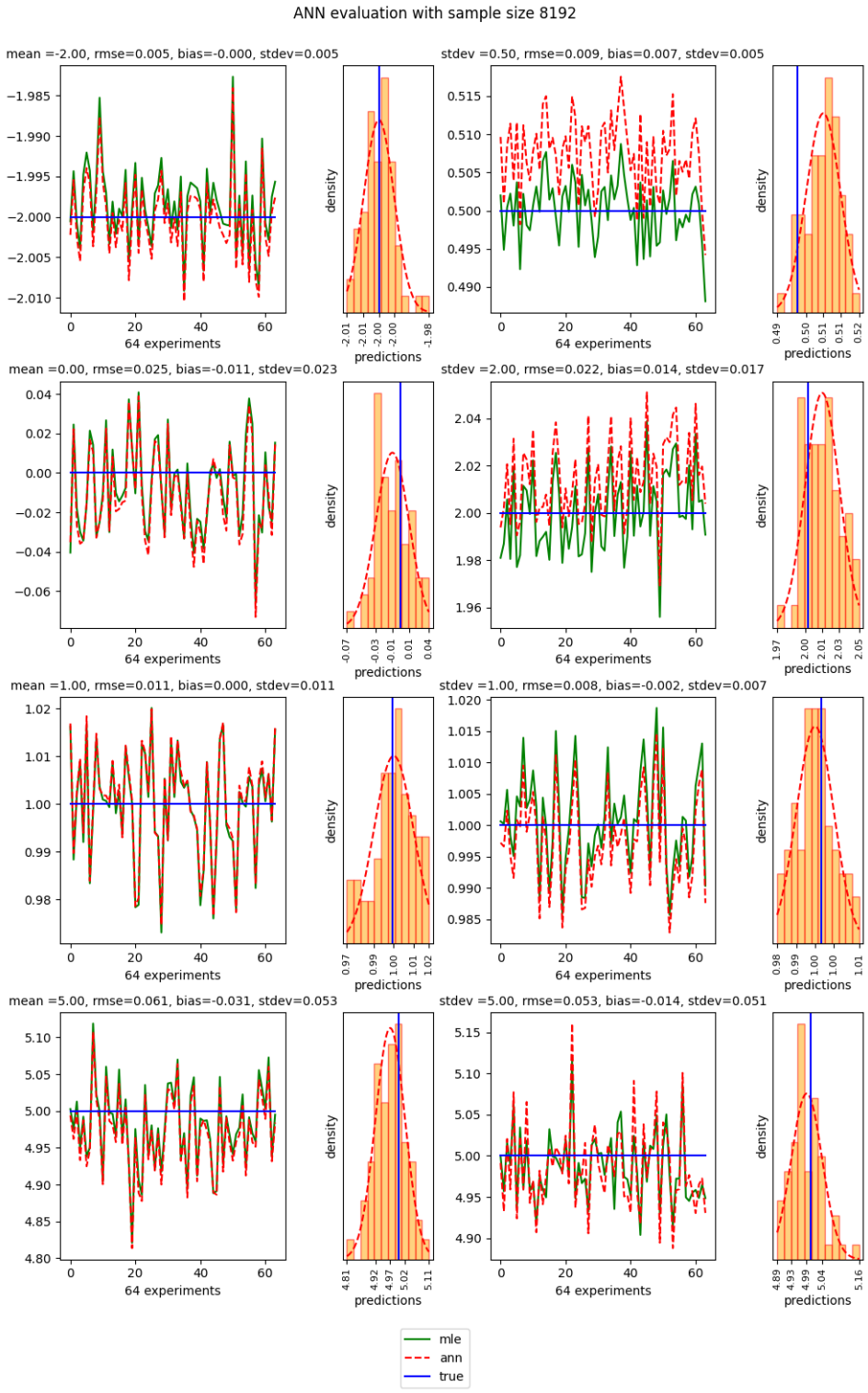

Figure 9.9: Evaluation results for the normal distribution, with sample size 8,192

As we did for the exponential, we can feed the ANN some uniform vectors with random size and extent, and verified that the ANN actually learned the sample mean and standard deviation functions, in figure 9.10.

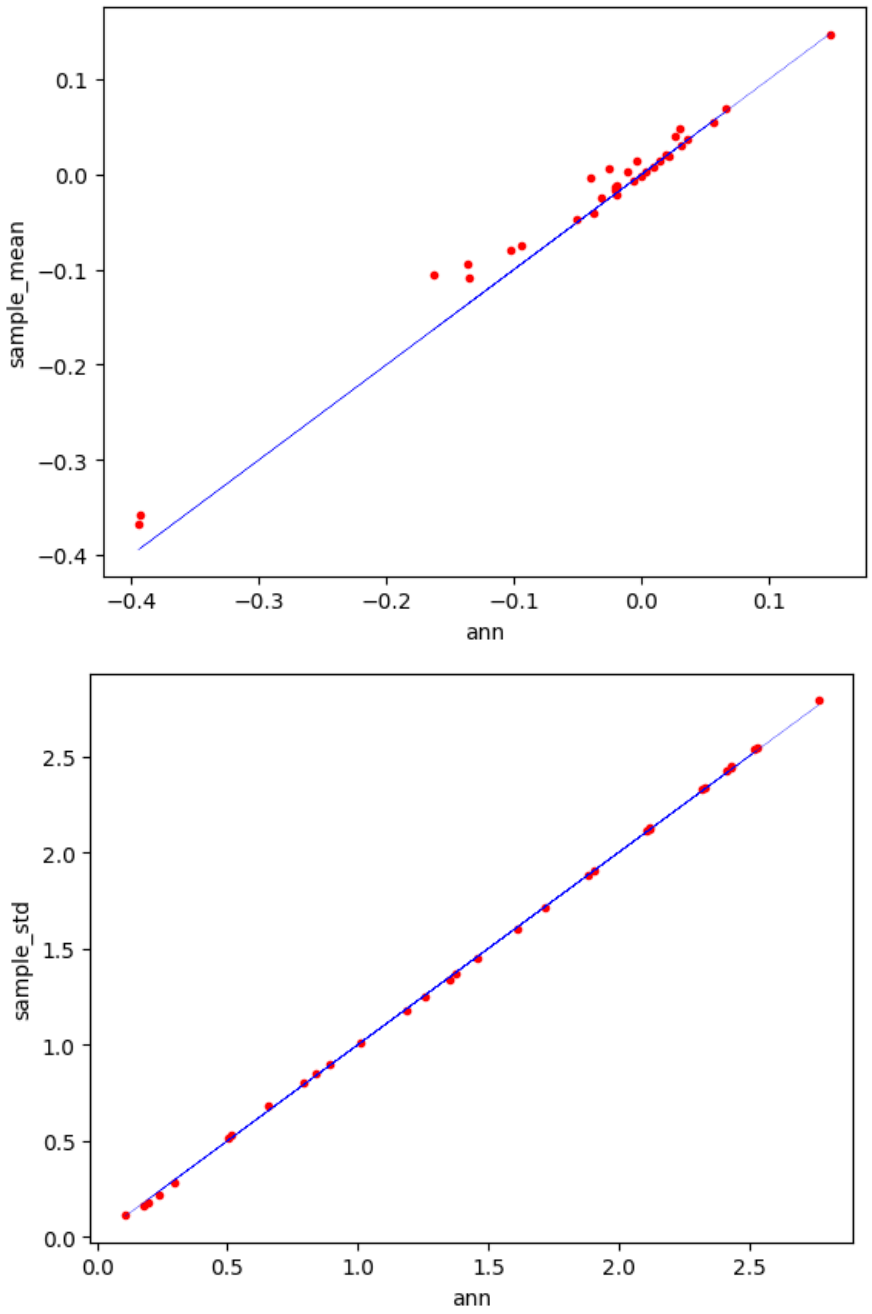

Figure 9.10: ANN 'predictions' against sample mean and standard deviation, 'samples' are of random size and range, and filled with uniform random numbers. The figure illustrates that the ANN effectively learned the sample mean and standard deviation functions.

9.4 Gamma and Erlang distributions

Let us switch gears, and move on to distributions that are harder to estimate. The next one is the Gamma distribution, which also includes the Erlang distribution as a special case with integer shape[1]. Shapes are picked from independent distributions, both Exp(0.5), translated by 0.2. Gamma samples are simulated with TensorFlow's built-in function.

There exists (to our best knowledge) no exact formula for the MLE of the shape parameter. We use scipy.stats' gamma.fit() function as a reference. Its documentation states that it estimates MLE, but it doesn't say how. We trust that SciPy, one of the most heavily used scientific libraries in the world, correctly implements current best practice.

The proof of concept for the gamma distribution is found in the notebook ann_gamma_final.ipynb in our GitHub repo.

The ANN and loss function are identical to normal, but the training loop needed some tinkering. Details are found in the notebook. Sampling from the gamma distribution is computationally heavy, much more so than any other distribution we considered, resulting in much longer training times. In addition, training takes a lot more epochs, typically $2,000$, epochs where 500 are sufficient for other distributions. It would be helpful to understand why.

The network also struggles to learn good estimates with small shapes (less than around 0.5) and/or small rates (less than around 0.25). This is something to investigate. One idea for further research is to train multiple ANN with shapes of different ranges, estimate by obtaining predictions from multiple ANNs, and picking the one that maximizes a criterion such as likelihood. For now, we under-weighted small shapes and small rates so that they don't interfere with training. We also had to remove examples with rates less than 0.2 from the training set, as they interfered with training.

Complete evaluation results are found on the notebook. The following figures show some partial results, all with samples of size $1,024$, with small, medium and large shapes. Results with other sample sizes are available on the notebook. SciPy is unstable with small samples.

Each row is a separate configuration, with shape estimates on the left, and rate on the right.

Performance is much worse than for exponential or normal distributions. SciPy's routine struggles too, illustrating that the estimation of gamma parameters is a hard problem.

The ANN has decent performance, and clearly beats SciPy, with small shapes. It is roughly on par with medium shapes, where both algorithms provide solid results: shape error is within 0.5, which is good enough to estimate Erlangs, and rate error is within 10% of its value. Large shapes are more interesting: the ANN completely fails to estimate the Erlang(5, 0.1) distribution. Recall we removed small rates from the dataset, as they were interfering with training. Sadly, unlike with exponential or normal distributions, it doesn't look like the ANN is capable of extrapolation[2]. The ANN beats SciPy with the 7.5 shape. And they both struggle with the very large shape of 10. For the ANN, it could be another extrapolation problem, but it is interesting that SciPy struggles too.

Gamma estimation definitely deserves further research and tinkering, but the results are encouraging in regards of the difficulty of the problem, as evidenced by the poor performance of SciPy's algorithm.

[1] We also wanted to try building a 'proper' Erlang estimator, by forcing the ANN to return integer shapes, but this remains a topic for further research.

[2] This matter with small rates really deserves further investigation: why is it such a problem? And why did we not see problems with small rates with the exponential distribution?

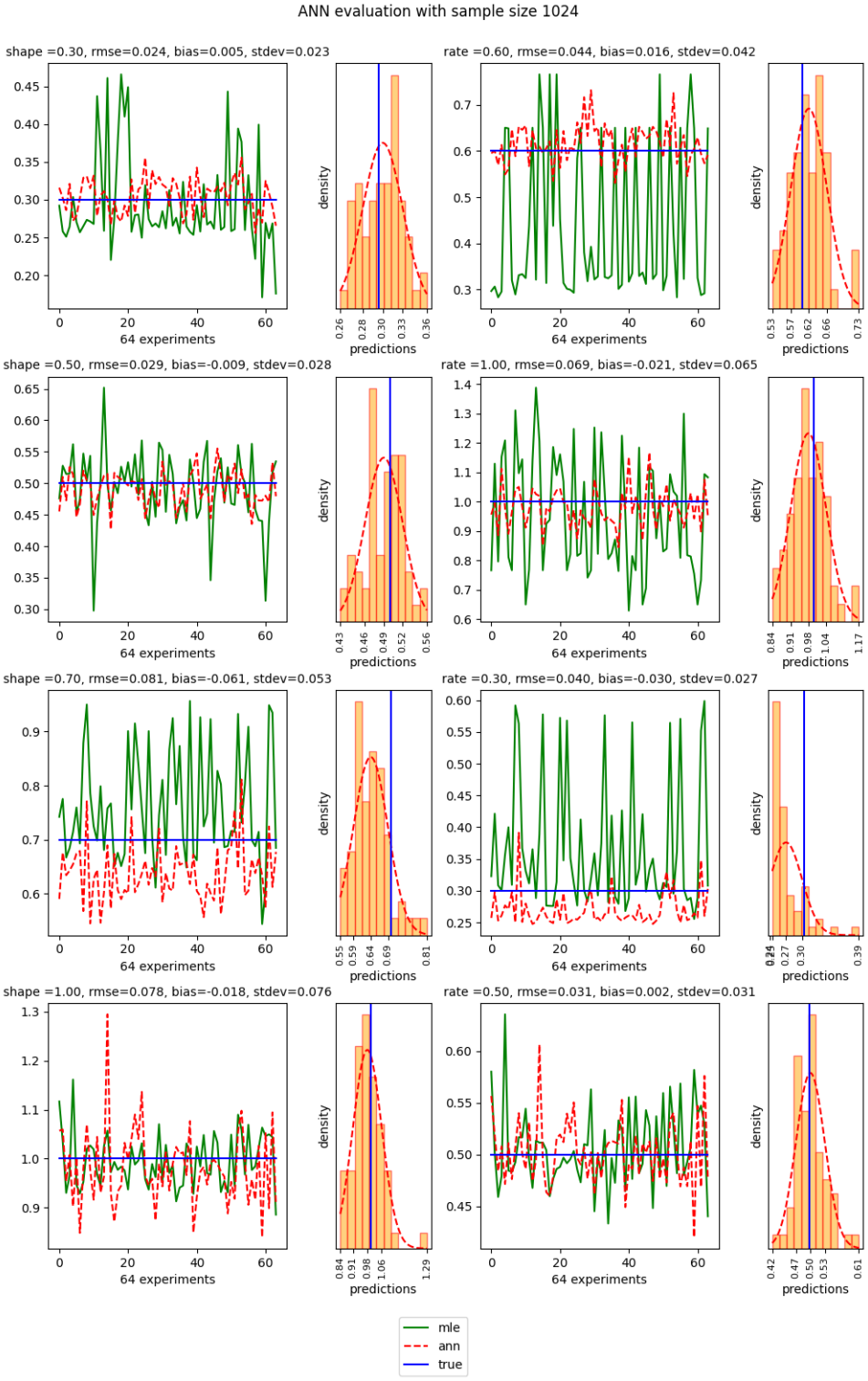

Figure 9.11: Evaluation results for the gamma distribution, with small shapes

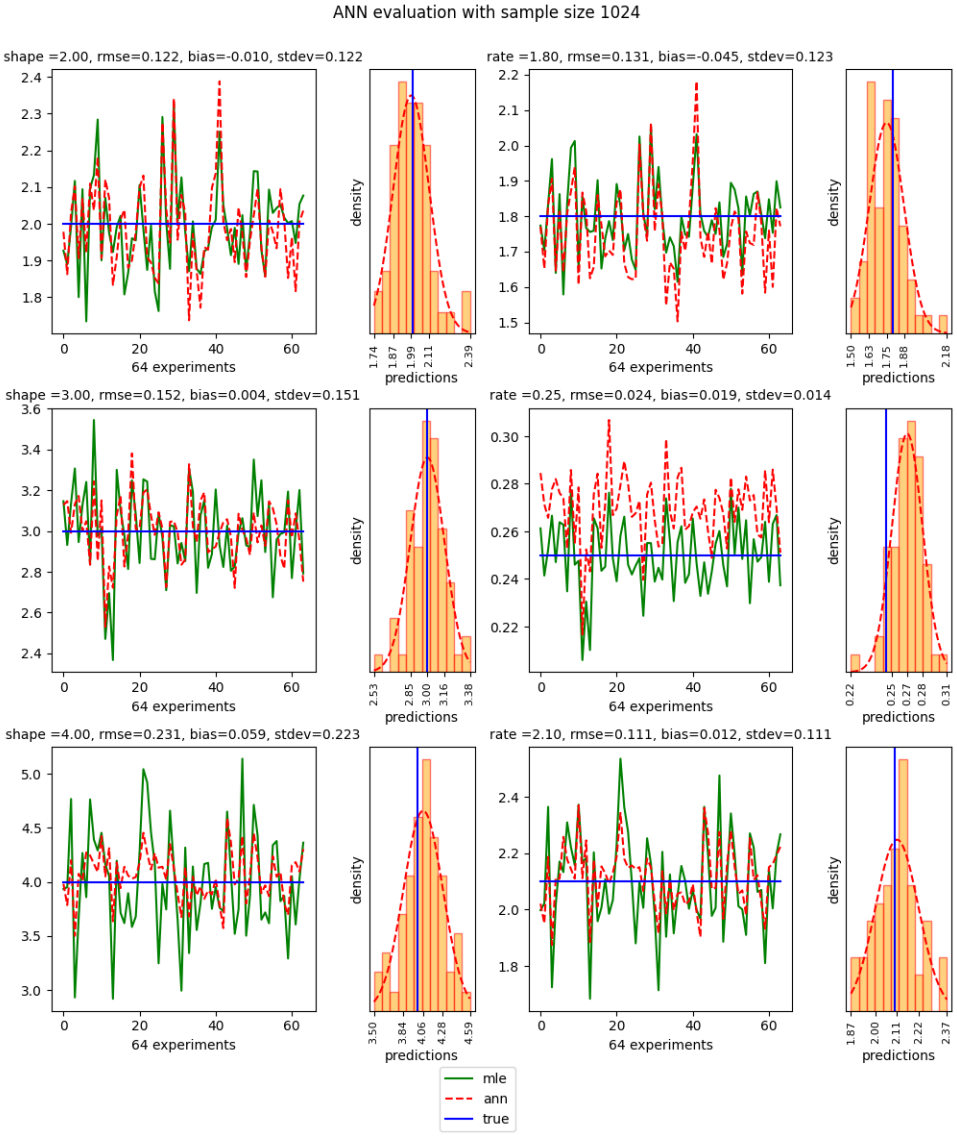

Figure 9.12: Evaluation results for the gamma distribution, with medium shapes

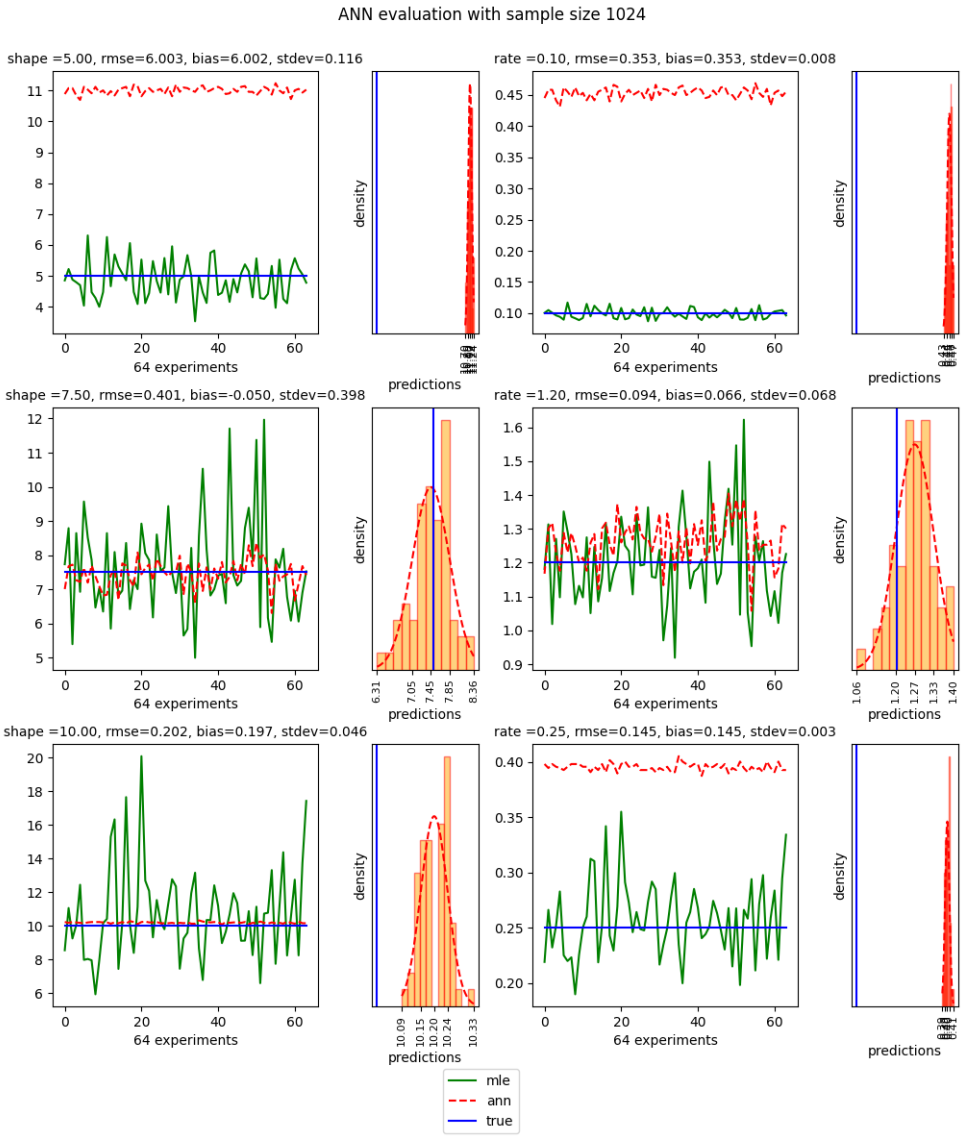

Figure 9.13: Evaluation results for the gamma distribution, with large shapes

9.5 Mixture of two exponential distributions

The most complicated distribution considered in this study is the mixture of exponential distributions. The next step (for further research) is to experiment with mixtures of more than two exponentials, and then, throw Erlangs into the mix. The mixture is hard to estimate, but easy to sample. First, pick the hidden state for every observation, then, simulate the observation from the corresponding distribution. Probabilities are picked from independent Exp(1) distributions, then normalized by the sum[3]. Rates, as usual, are picked from Exp(0.8) distributions, independently.

One particularity of mixture distributions, that prevented the ANN from learning anything, before it was fixed, is the non-uniqueness of representation. The mixture (λ_1, λ_2) with probabilities (p_1, p_2) is evidently

[3]For mixtures of two, we could have picked p_1 uniformly is $(0, 1)$ and set $p_2 = 1 - p_1$, but wanted future-proof code that works with more than 2 distributions.

the same as (λ_2, λ_1) with probabilities (p_2, p_1), but the loss function penalized inverted answers, confusing the training process. This was easily resolved by sorting parameters in the canonical order $\lambda_1 < \lambda_2$.

This is not the only problem, however, and the ANN was still unable to learn. When the two rates are close, the mixture reduces to an exponential distribution, so it is impossible to read probabilities from the sample. But the ANN had to make an estimate. The estimate could only be random, yet, the loss function penalized incorrect answers, once again confusing training. The same happens when one probability is close to 0. It is impossible to predict the corresponding rate. Once identified, these problems were easily resolved with per-sample weights: we under-weighted examples with similar rates or small probabilities. And then the ANN could learn.

There exists no analytic expression for the MLE for mixtures. Current best practice for estimation is the expectation-maximization (EM) algorithm. We used the open-source implementation in the well-maintained and documented pomegranate[4] library by Jacob Schreiber, a respected researcher from Stanford. Presumably, this is an implementation in line with current best practices.

The proof of concept for the mixture of two exponential distributions is found in the notebook named ann_mix_2exp_final.ipynb in our GitHub repo.

There is no tinkering with architecture or training loop, since this is the distribution for which the generic settings were set in the first place.

Complete evaluation results are found on the notebook. The following figures show some partial results. Each row is a separate configuration, with estimates left to right being p_1, p_2, λ_1 and $\lambda 2$.

[4] https://pomegranate.readthedocs.io/en/latest/index.html

Figure 9.14: Evaluation results for the mixture of exponential distributions, with sample size 128

Figure 9.15: Evaluation results for the mixture of exponential distributions, with sample size $1,024$

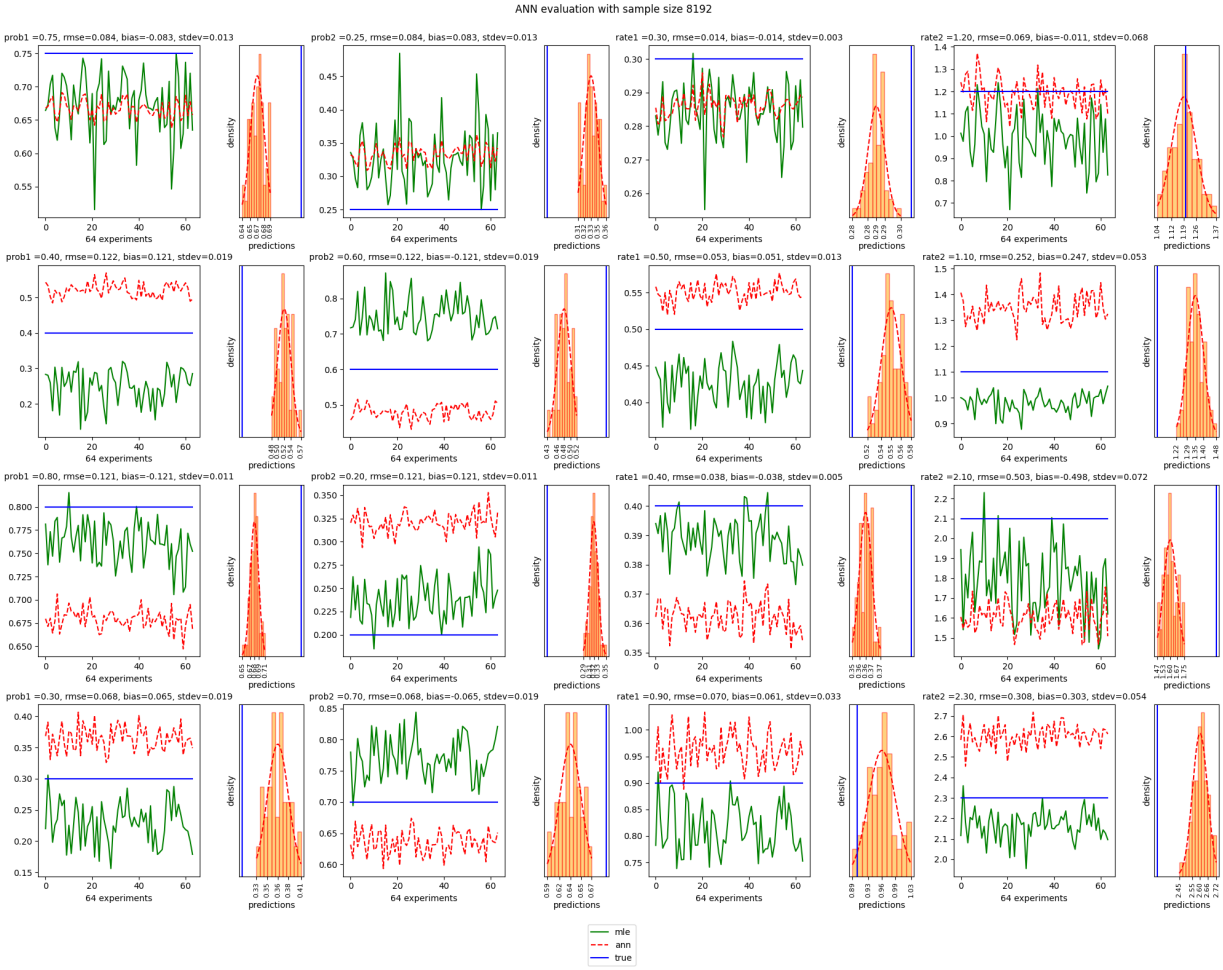

Figure 9.16: Evaluation results for the mixture of exponential distributions, with sample size $8,192$

With small samples, although both algorithms show an abysmal performance, the ANN is generally on par or better than EM. With medium samples, the ANN has a decent performance and clearly beats EM. This is a great achievement for the project. With large samples, both algorithms, again, are roughly on par (and neither does great given the large sample size).

There is scope for further improvement. Recall that the "right" ANN architecture was only uncovered a few weeks prior to the project's deadline.

Conclusion

This thesis started with a review of PH distributions and their main properties. The first part of this document is mainly a report of a Bachelor student's journey to understand these topics and make them understandable to others. It also develops and implements an algorithm for the simulation of general PH distributions, and provides an illustration with a bimodal mixture of Erlang distributions.

Then, the project turned into the exploration of end-to-end estimation by Machine Learning, as proposed in last year's paper [7], working with a number of distributions, mainly from the PH family. The main idea is to train a neural network to learn an estimation function, on a dataset where labels are distribution parameters, and inputs are samples drawn with these parameters.

The goal was to eventually bring the two parts together and estimate parameters of the general PH distribution: probability vector and intensity matrix. Sadly, we didn't go that far, so the two parts are essentially independent.

One key achievement of the second part was the design of a special neural network architecture, inspired by the Deep Sets of the famous paper [9]. This architecture turned things around, unfortunately at a late stage in this project. Its key benefit, besides permutation invariance, is that resulting ANNs accept samples of arbitrary size, and behaves as expected, with standard error reducing when sample size grows. A special training procedure was also designed, whereby multiple random samples are drawn from the same distribution for every training example, so the training loop can estimate and minimize bias. Without this, training results in biased estimators. After the ANN is trained, estimation is performed by feeding samples of arbitrary size and obtaining parameter estimates.

Performance was systematically evaluated and compared to MLE estimates. The ANN closely matches MLE for simple distributions, exponential and normal. It manages to learn the MLE formula, so it can extrapolate to samples of arbitrary size, drawn with parameters of magnitude unseen in the training set. All this without having ever seen a MLE, or having any knowledge of the generative distribution, its pdf or moments. Incredibly, the ANN 're-derives' MLE expressions by itself, from examples of samples labeled with true generative parameters.

We then moved on to more complex distributions, gamma and mixture of exponentials, which are much harder to estimate. There is no analytic expression for the MLE, so estimation is performed numerically, with algorithms such as expectation-maximization (EM). We have tested (presumably) best practice implementations of those algorithms, and performance is nowhere close to the analytic MLEs of the exponential or normal distributions, illustrating that the estimation of those distributions is a hard problem.

While we didn't achieve the stated goals of estimating general PH distributions, or write a software compatible with arbitrary distributions, the results are very promising, and warrant further research on this exciting topic:

- Further fine-tune the ANN and the training loop. For instance, explore the idea of training on samples of different sizes.
- Develop a truly distribution-independent software, so that users can plug and play with arbitrary distributions. Research training loop heuristics that work well for all distributions without manual tinkering.
- Investigate and improve estimation for gamma and Erlang distributions. Experiment with multiple ANNs depending on the magnitude of the shape parameter.
- Test on many more distributions, particularly, mixtures of more than 2 distributions, up to and including general PH distributions.
- Discriminate between distributions, not only predict parameters. For a given sample, return the most likely distribution, maybe based on likelihood, along with parameter estimates.

The reason why the thesis was split into two apparently distinct projects, is that the initial intention was to estimate general PH distributions with Machine Learning. There remains a fair amount of work to get there. But the thesis did succeed in providing a solid and encouraging proof of concept. We are now convinced that estimation of general PH is within our reach.

Bibliography

[1] C. Bishop. *Pattern Recognition and Machine Learning*. Springer Verlag, 2006.

[2] M. Bladt and B. F. Nielsen. *Matrix-Exponential Distributions in Applied Probability*. Springer, 2017.

[3] L. J. R. Esparza. Maximum likelihood estimation of phase-type distributions. *Technical University of Denmark. IMM-PHD-2010-245*, 2011.

[4] R. Feres. Notes for math 450 continuous-time markov chains and stochastic simulation. *Washington University in St. Louis*. https://www.math.wustl.edu/~feres/Math450Lect05.pdf.

[5] A. Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2022.

[6] A. Horvath, M. L. Scarpa, and M. Telek. *Principles of Performance and Reliability Modeling and Evaluation, chapter 10: Phase-Type and Matrix Exponential Distributions in Stochastic Modelling*. Springer, 2016.

[7] A. Lenzi, J. Bessac, J. Rudi, and M. L. Stein. Neural networks for parameter estimation in intractable models. *Computational Statistics Data Analysis*, 185:107762, 2023.

[8] K. Sigman. 4703-07-notes-mc. *Columbia University*, 2007. https://www.columbia.edu/~ks20/4703-Sigman/4703-07-Notes-MC.pdf.

[9] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola. Deep Sets. *arXiv e-prints*, page arXiv:1703.06114, Mar. 2017.