Università degli Studi di Milano-Bicocca

Scuola di Economia e Statistica

Corso di Laurea in

Scienze Statistiche ed Economiche

Forecasting Spatio-Temporal Data with Bayesian Neural Networks

Relatore: Prof. Stefano Peluso

Correlatore: Prof. Mirko Cesarini

Tesi di Laurea di:

Federico Ravenda

Matricola 829449

Anno Accademico 2021-2022

Contents

Abstract

In the last decades, the availability of spatial and spatio-temporal data has substantially increased, mainly due to the advances in computational tools which allow us to collect real-time data coming from GPS, satellites, etc. This means that nowadays in a wide range of fields, from epidemiology to ecology, to climatology and social science, researchers have to deal with geo-referenced data i.e., including information about space (and possibly also time). There are countless scientific papers in peer reviewed journals using more or less complex and innovative statistical models (especially Bayesian models), to deal with the data spatial and/or temporal structure, covering a wide range of applications.
Machine Learning and Deep learning have attracted tremendous attention from researchers in various fields e.g., AI, computer vision, and language processing, but also from more traditional sciences e.g., physics, biology, and manufacturing. The added value provided by these algorithms is the few or no assumptions to be met. They are far more flexible than statistical models as they have relaxed requirements about collinearity, normal distribution of residuals, etc. Thus, they have high uncertainty tolerance.
In spite of the many pros neural networks have, some cons have also to be mentioned such as, for example, lack of interpretability (black box models) and the high computational cost w.r.t. traditional algorithms.
Neural networks, image processing tools such as convolutional neural networks, sequence processing models such as recurrent neural networks, and regularisation tools such as dropout, are used extensively. However, representing model uncertainty is very important in fields such as physics, biology, business, and manufacturing. With the recent shift in many of these fields towards the use of Bayesian uncertainty, new possibilities arise from deep learning. The goal of this thesis is twofold: to create a model that is able to infer both the spatial and temporal components and to combine the advantages of both approaches, namely the flexibility of a neural network and the quantification of uncertainty offered by a traditional bayesian regression model. To implement this model different tools were used:

- Embeddings, a relatively low-dimensional space into which we translate high-dimensional vectors, to model the spatio-temporal components in order to feed them to neural networks.

- A Neural Networks Architecture able to handle sequences of data and quan-

tify the uncertainty w.r.t. each prediction.

To achieve that the Bayesian learning paradigm is applied to neural networks which results in a flexible and powerful nonlinear modelling framework that can be used for Forecasting task. Within this framework, all sources of uncertainty are expressed and measured by probabilities. This formulation enables a probabilistic treatment of a priori knowledge, domain specific knowledge, model selection schemes, parameter estimation methods, and noise estimation techniques. The main contributions of this thesis are the following:

- Use of Bayesian Neural Networks for forecasting purposes and uncertainty quantification;
- Use of Probabilistic Layers to model the response variable;
- Use of different types of Embeddings to synthesize Temporal and Spatial components;
- Use of Recurrent, Convolutional Layer and Attention Mechanism to model dynamical temporal sequences of data;

The Neural Network Architecture thus implemented reaches satisfactory goodness-of-fit performances, delivers precise prediction of events over varying size time interval, and outperforms traditional parametric and nonparametrics temporal and spatio-temporal techniques.

This Thesis is structured in the following way:
In **Chapter 1** the main problem of traditional neural networks in prediction problems is discussed as well as different forms of uncertainty that Bayesian Neural Networks are able to take into account, and a review of the state-of-the-art spatio-temporal models present in literature is done.
In **Chapter 2** a distinction is made between probabilistic and non-probabilistic models, some of the traditional and more recent distributions that are used to model counting data are presented, and Bayesian Neural Networks (BNNs) are both intuitively and formally introduced using two different types of approaches: *Variational Inference* and *MC Dropout*.
In **Chapter 3** different types of approaches to create Spatial and Temporal embeddings are analyzed. Entity Embedding, Graph based methods and Variational AutoEncoders' approaches are introduced from a theoretical and practical point of view.
In **Chapter 4** several neural network architectures are introduced to model sequential data, and the Spatio-Temporal Bayesian Neural Network (**BSTNN**) architecture implemented for this thesis is formally introduced.
In **Chapter 5** the BSTNN architecture is applied for real data and predictions are compared with other spatio-temporal models, and in **Chapter 6** conclusions and possible developments are discussed.

Chapter 1

Bayesian Learning - An Introduction

In Section 1.1 a practical example of why using traditional neural networks can sometimes lead to results that are not intuitive and completely wrong is discussed. In Section 1.2 the concepts of aleatoric and epistemic uncertainty and how Bayesian Neural Networks are able to take these two aspects into account are introduced. In Section 1.3 a review of the state-of-the-art temporal and Spatio-Temporal models is made.

1.1 What's Wrong with Non-Bayesian Neural Networks? The Elephant in the Office Example

Deep Learning models can sometimes tell a completely wrong story. Usually, the predictions of traditional Deep Learning models are highly reliable when applied to the same data used in training, which somehow can take us into a false sense of security. Let's consider two images (as shown in Fig. 1.1: one, on the left, represents an elephant in its habitat, the other one, on the right, represents an elephant in an office).

Suppose the aim is to classify these two images using an architecture such as VGG16 [1] [2], trained on ImageNet.
In ImageNet there are a lot of elephant images that belong to 2 classes: 'Indian Elephant' and 'African Elephant'. Since VGG16 is a top architecture with high performance on ImageNet dataset, we can expect it is able to correctly classify the two elephant images. However, the same network that nicely classified the elephant in the left image as an african elephant species completely fails for the image on the right — the Deep Learning model can't see the elephant in the

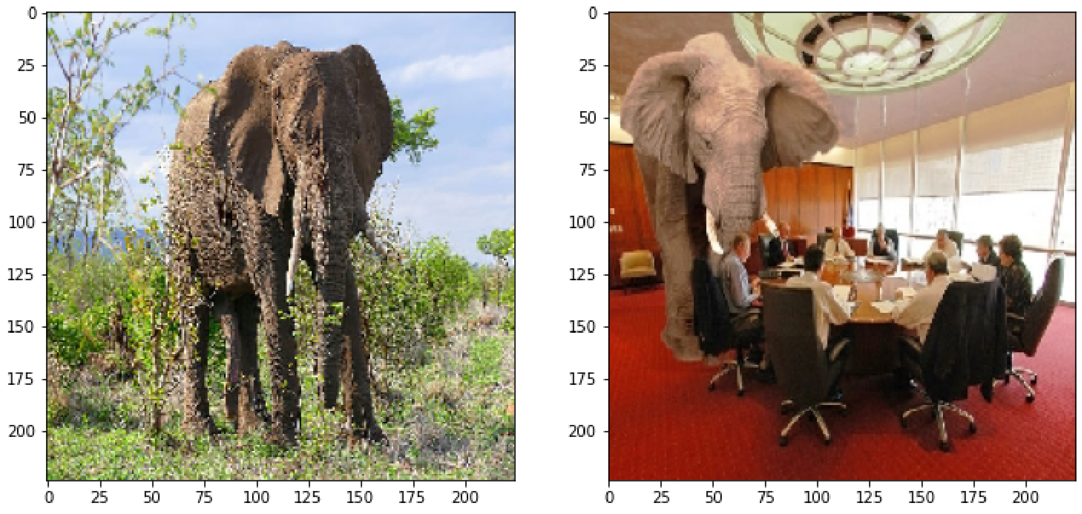


Figure 1.1 – On the left-side an elephant in its natural environment, while on the right side an elephant in an office surrounded by a group of workers

office!
Below the top 5 predictions of the high-performance network are considered:

1. library
2. wall_clock
3. shoe_shop
4. restaurant
5. toyshop

They are not even close!
Why does the Deep Learning model fail to see the elephant? This is because the training set used to fit the VGG16 model has no pictures of elephants in rooms. Not surprisingly, the elephant images in the training set show these animals in their natural environment. This is a typical situation where a trained Deep Learning model fails: when presented with an instance of a novel class or situation not seen during the training phase. Exaggerating a bit, but Deep Learning crucially depends on the big lie:

$$P(train) = P(test)$$

The condition $P(train) = P(test)$ is always assumed; but in reality, the training and the test data don't often have the same distribution. How can the network tell us that it feels unsure about a specific prediction? The solution is to introduce a new kind of uncertainty: the *epistemic uncertainty*.

1.2 Aleatoric Uncertainty vs. Epistemic Uncertainty

The notion of uncertainty [3] is crucial in machine learning and constitutes a key element of model methodology. Due to the steadily increasing relevance of machine learning for practical applications and related issues such as safety requirements, new problems and challenges have recently been identified by machine learning researchers, and these problems may call for new methodological developments. In particular, this includes the importance of distinguishing between two different types of uncertainty, often referred to as *aleatoric* and *epistemic*.
Uncertainty occurs in different flavours in the machine learning field, therefore different approaches are required to deal with the several settings and learning problems from an uncertainty modeling point of view. Machine learning is essentially concerned with extracting patterns from data and building models, often using them for the purpose of prediction (classification or regression). As such, it is inseparably connected with uncertainty. Indeed, learning in the sense of generalizing beyond the data seen so far is necessarily based on a process of induction. Such models are never probably correct, they try to investigate the true generating function distribution, but only hypothetical and therefore uncertain, and the same holds true for the predictions produced by a model. In addition to the uncertainty inherent in inductive inference, other sources of uncertainty exist, including incorrect model assumptions and noisy or imprecise data. Needless to say, a trustworthy representation of uncertainty is desirable and should be considered a key feature of any machine learning method, especially the safety-critical application domains such as medicine, epidemiology, or quality check fields.
Bayesian Neural Networks can take into account two different types of uncertainties [4].

Aleatoric uncertainty is also known as statistical uncertainty. In Statistics, it is representative of unknowns that differ each time we run the same experiment (train the model). In deep learning, it refers to the uncertainty of the model outputs. As shown in the Figure 1.2 above, given that the black line is the prediction, the orange area would be the aleatoric uncertainty.
We can regard it as the confidence level of the prediction. In other words, it tells us how confident our prediction results are. If the interval is small, the actual value would have a larger chance to have a closer value towards our prediction value. On the contrary, if the interval is large, the actual value may have a big discrepancy with our prediction value.
The prototypical example of aleatoric uncertainty is coin flipping: The data-generating process in this type of experiment has a stochastic component that cannot be reduced by any additional source of information. Consequently, even the best model of this process will only be able to provide probabilities for the

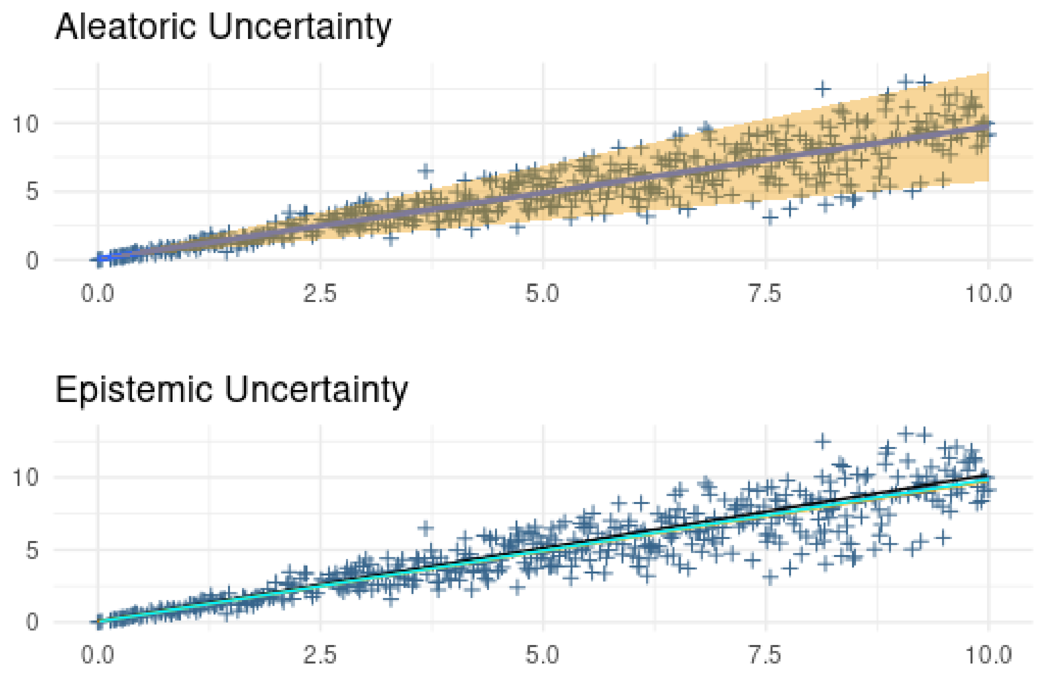


Figure 1.2 – Graphical Representation of Aleatoric and Epistemic Uncertainty

two possible outcomes, heads and tails, but no definite answer.

Epistemic uncertainty is also known as a systematic uncertainty. In deep learning, epistemic uncertainty refers to the uncertainty of the model weights. As shown in the Fig. 1.2, every time we train the model, the weights may slightly vary. This variation is actually epistemic uncertainty.
More in general, it refers to the ignorance of the agent or decision maker, and hence to the epistemic state of the agent instead of any underlying random phenomenon. As opposed to uncertainty caused by randomness, uncertainty caused by ignorance and lack of informations can theoretically be reduced on the basis of additional information.

In other words, epistemic uncertainty refers to the reducible part of the (total) uncertainty, whereas aleatoric uncertainty refers to the irreducible part.
In the context of a probabilistic neural network: given an input x, the final layer of the network gives a probability distribution, either on the set of classes in the case of classification or on a point of prediction in the case of regression. In this setting, epistemic uncertainty is usually interpreted as the uncertainty of the parameters of the neural network and could be reduced if we had a larger training dataset or if we could add some more information. Aleatoric uncertainty, on the other hand, appears in the probabilistic prediction itself realized by maximizing likelihood inference.

1.3 Spatio-Temporal Forecasting - State Of The Art

The adoption of spatial and temporal statistical tools for the analysis of data that move over time and space is gaining more and more attention in the literature, as longer geolocalized-time-series-data and faster computational methods become available. There are mainly two lines of research regarding these methods, the first one purely statistical, the second one concerning the use of Machine Learning tools.
In this section I will sum up the state of the art for spatial and temporal modelling, to the best of my knowledge.
Following are a few methods that are frequently used in the statistics literature:

- From a statistical point of view, Latent Gaussian Random Fields have become increasingly popular for modeling spatio-temporal data. The Integrated Nested Laplace Approximation (INLA, [5]) is a method for approximate Bayesian inference that allows one to efficiently estimate such models. It is an established alternative to other simulation-based methods, such as Markov Chain Monte Carlo because of its computational efficiency and ease of use via the `R-INLA` package. In [6], the authors used mortality data from 2005–2015 to explore spatio-temporal variation in suicide rates (SRs) to predict year and county-specific SRs, while in [7] INLA were applied to data from 71,057 children aged 0 to under 10 years from rural north-east South Africa living in 15,703 households over the years 1992–2010 to do inferences and predictions about the effects on HIV/TB mortality due to greater birth weight, older age, and more antenatal clinic visits during pregnancy.

- Spatio-temporal Autoregressive models are often used in longitudinal or time series data, and models the outcome variable as it depends linearly on it's own previous values in space and time. In [8] authors use autoregressive models to spatio-temporal data related to house prices from 1969 to 1991 from Fairfax County Virginia while in [9] they use an autoregressive model to analyse office transaction prices in the Paris property market.

- Spatial Multivariate Age-Period-Cohort (**APC**) effects takes into consideration APC effects as well as differential geographical effects on behavior. Often used in cancer models to assess relationships of where people live, how that effects their behavior, in addition to classic APC effects in cancer. An application can be founded in [10], where authors model the US state-level opioid overdose moratlity in men and women aged 15-64 years.

- P-Splines models provides smoothed parameter estimates along space and time on a large, global scale. The smoothing is carried out in three dimensions (longitude, latitude, and time). This method can be useful if significant changes at different time points are expected. In the work of [11] penalized splines for smoothing risks in both the spatial and the temporal dimensions are applied to model mortality data due to brain cancer in continental Spain over the period 1996-2005.

As the number, volume and resolution of spatio-temporal datasets increase rapidly, traditional statistics methods for dealing with these types of data are becoming overwhelmed [12]. With the advances of deep learning techniques, deep leaning models such as convolutional neural network (CNN) and recurrent neural network (RNN) have enjoyed considerable success in various machine learning tasks due to their powerful hierarchical feature learning ability in temporal domains, and have been widely applied in various temporal data mining tasks. Trajectories and time series can be both represented as sequences while Spatial maps can be both represented as graphs and matrices, depending on different applications. For example, in urban traffic flow prediction, the traffic data of a urban transportation network can be represented as a traffic flow graph [13] or cell region-level traffic flow matrix [14].

1.3.1 Deep Learning Models for Spatial and Temporal Data

CNN. Convolutional neural networks (CNN) is a class of artificial neural networks that are applied in computer vision tasks. A CNN model contains the convolutional layer, which is responsible for determining the output of neurons of which are connected to local regions of the input through the calculation of the scalar product between their weights and the region connected to the input volume. Compared with a traditional Multilayer Perceptron neural Network, CNNs have the following distinguishing features that make them achieve much generalization on computer vision problems: 3D volumes of neurons, local connectivity and shared weights. CNN is designed to process image data. Due to its powerful ability in capturing the correlations in the spatial domain, in the last years is gaining importance in mining spatio-temporal data.
RNN and LSTM. A recurrent neural network (RNN) is a class of artificial neural network where connections between nodes form a directed graph along a sequence. RNN is designed to recognize the sequential characteristics and use patterns to predict the next likely scenario. They are widely used in natural language processing (NLP) field. A major issue of standard RNN is that it only has short term memory due to the issue of vanishing gradients. Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) networks are an extension for RNNs, which are capable of learning long-term dependencies of the input data.
GraphCNN. GraphCNN is recently architecture widely studied to generalize CNN to graph structured data. The graph convolution operation applies the convolutional transformation to the neighbors of each node, followed by pooling

operation. By stacking multiple graph convolution layers, the latent embedding of each node can contain more information from neighbors which are multihops away. After the generation of the latent embedding of the nodes in the graph, one can either easily feed the latent embeddings to feed-forward networks to achieve node classification of regression goals, or aggregate all the node embeddings to represent the whole graph.
An application can be found in [15] [16] where the authors propose a novel graph neural network architecture for spatial-temporal graph modelling, by developing a novel adaptive adjacency matrix and learn it through Node Embedding in order to capture the hidden spatial dependency in the data and they stacked dilated 1D convolution component in order to handle very long sequences. In [17] authors use a forecasting approach for COVID-19 case prediction that uses Graph Neural Networks and mobility data. The proposed approach learns from a single large-scale spatio-temporal graph, where nodes represent the region-level human mobility, spatial edges represent the human mobility based inter-region connectivity, and temporal edges represent node features through time.
Seq2Seq. A sequence to sequence (Seq2Seq) model aims to map a fixed length input with a fixed length output where the length of the input and output may differ. It is widely used to various NLP tasks such as machine translation, speech recognition, and online chatbot. Seq2Seq is general framework that can be used to any sequence-based problem. Seq2Seq model generally consists of 3 parts: encoder, intermediate (encoder) vector and decoder. Due to the powerful ability in capturing the dependencies among the sequence data, Seq2Seq model is widely used in ST prediction tasks where the Spatio-Temporal data present high temporal correlations such as urban crowd flow data and traffic data.
Applications of Seq2Seq model can be found in [18] where authors use an encoder-decoder architecture, where both the encoder and the decoder consist of multiple spatio-temporal attention blocks to model the impact of the spatio-temporal factors on traffic conditions while in [19] an encoder-decoder framework with attention mechanism is proposed for multivariate time series forecasting.

1.3.2 Research Question & Contributions

The Research Question that guided the advancement of this thesis is as follows: *Can a neural network emulate and improve performances of traditional statistical Spatio-Temporal models, in which the spatial and temporal components are explicitly modeled and a Bayesian framework is incorporated for uncertainty quantification, and, at the same time, exploit the advantages of neural networks such as flexibility and the fact that do not need strict statistical assumptions to be met?*

The architecture implemented in this thesis differs from those state-of-the-art listed in the previous Subsec. in the use of:

- Probabilistic and variational layers to take into account uncertainty;
- A hybrid approach in which spatial and temporal components are modeled through embeddings;
- Convolutional, Recurrent, and Attention layers to model sequential data.

This architecture outperforms traditional methods that can be found in literature on the datasets considered in Chapter 5 and it's flexible enough to be easily extendable to other applications as well.

Chapter 2

Bayesian Neural Networks

In Section 2.1 the differences between a probabilistic and a non-probabilistic regression approach is discussed. Section 2.2 explains how to build a probabilistic deep learning model and introduces the distributions that are used to model counting data. Section 2.3 discusses the Bayesian approach for deep learning and summarizes the differences between Deterministic and Bayesian Neural Networks, while in Section 2.4 Bayesian Neural Networks are formally introduced and the two methods to guide the learning process of this type of networks are discussed.

2.1 Non-Probabilistic & Probabilistic Regression

Non-probabilistic regression (a.k.a. curve fitting) is the science of putting lines through data points [20]. Considering two features x and y as, respectively, the covariate and the outcome. With linear regression in its most simple form (simple linear regression), a straight line is put through the data points. In the case of just one covariate, linear regression model has only two parameters, a and b:

$$y = a \times x + b$$

After the definition of the model, the parameters a and b need to be determined so that the model can be actually used to predict a single best guess for the value of y, given x. In the machine learning vocabulary, the step of finding good parameter values is called training and it is done by fitting the model's parameters to the training data. One of the main drawback of non-probabilistic regression is highlighted in Fig. 2.1: the curve correctly follows data trends (synthetic data), but cannot explain how much it feels unsure. Furthermore, in a non-probabilistic model, for each x value only one best guess can be obtained.

One of the major advantages of probabilistic models is that they provide an idea about the uncertainty associated with predictions. In other words, a probabilistic

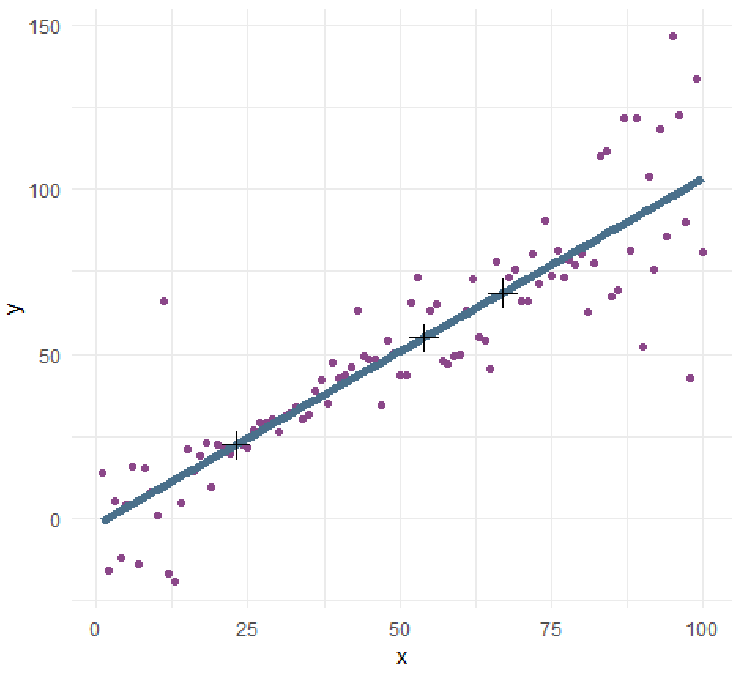


Figure 2.1 – A simple linear regression on simulated data. The real generating function is y = x. The dots are the observed data points; the straight line is the linear model. The positions of the "+" symbol indicate the predicted best guesses for three x values (23, 54, 67)

machine learning model can explain uncertainties on its prediction. These concepts related to uncertainty and confidence are extremely useful when it comes to critical machine learning applications such as disease diagnosis and autonomous driving. When training a probabilistic model on a set of data, instead of getting only a single best guess every x, a whole probability distribution is returned (see purple curve in Fig. 2.2 showing predictions' distributions) .
The solid line blue line in Fig. 2.2 indicate the positions and exactly matches the regression line in Fig. 2.1, which is predicted from a non-probabilistic model. The gray area that surrounds the mean indicate an interval in which 99% of all observations values are expected by the model.

2.2 Probabilistic Deep Learning

As already discussed in Chapter 1.2 there are two different types of uncertainties to take into account when training a model. To capture data-inherent variability, a conditional probability distribution (CPD) is used: $p(y|x)$. With this distribution, the outcome variability of y by a model is captured. To refer to this inherent variability in the Deep Learning field, the term *aleatoric uncertainty* is used. In this subsection the focus is on developing and evaluating probabilistic models to quantify the aleatoric uncertainty. The quantification of the uncertainty has real practical importance for making critical or costly decisions based

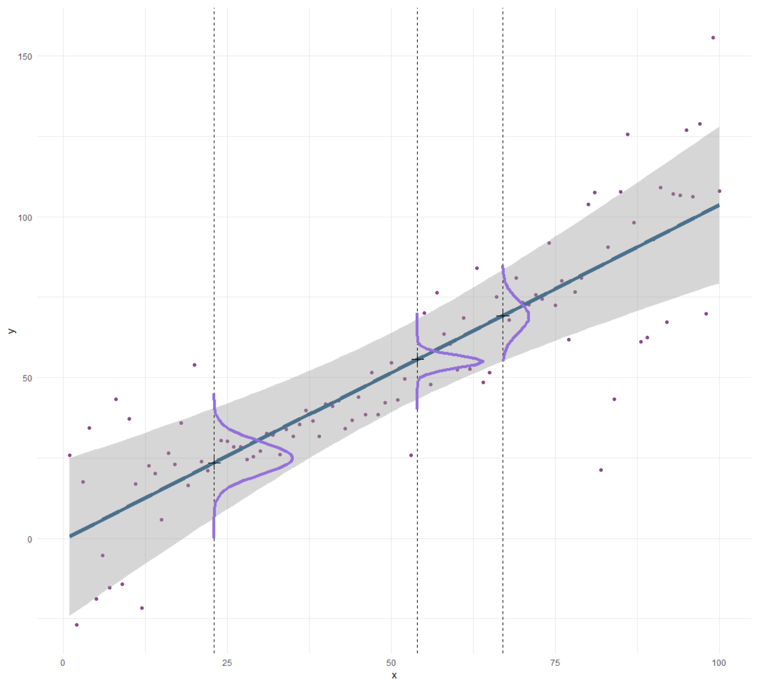


Figure 2.2 – Scatter plot and regression model for the example dataset. The dots are the observed points. At each test value (23, 54, 67), a Gaussian distribution is fitted (in purple) that describes the probability distribution. For the three values, the predicted probability distributions are shown. The solid blue line indicates the positions of the mean values of all distributions corresponding to the x values between 0 and 100. The gray area indicates an interval in which 99% of all values are expected by the model.

on the predictions and such information can only be derived from a probabilistic model that predicts not a single value, but a reliable probability distribution over all possible domains. To fit a probabilistic deep learning model, the idea is to use a neural network to determine the parameters of the predicted conditional probability distribution (CPD) for the outcome. The key steps for developing a probabilistic Deep Learning model are the following:

- Pick an appropriate distribution model for the outcome.
- Set up a neural network that has as many output nodes as the number of model parameters.
- Derive from the picked distribution the Negative-Log-Likelihood (NLL) function and use that function as the loss function to train the model (See the Appendix 7 for a formal derivation).

When setting up a probabilistic model, one of the most crucial aspects is to pick the right distribution model for the response. E.g., when the outcome is continuous, a good choice is to use a Normal distribution with two parameters to estimate: a mean μ and a standard deviation σ. Instead, if we are dealing with a classification task, the outcome will be categorical and a proper choice of the outcome distribution would be a multinomial distribution (if the categories are

more than 2) and the parameters to estimates are: $p_i, i = 1, ..., q$ where q is the number of categories.
Sometimes the outcomes are modeled as counts. There are many cases where researchers want to model count data e.g., counting the number of people y_i that are on a given image x_i, or the number of persons killed in a road accident during a certain year based on some features x, or the number of units sale by a store during a given hour. To set up a probabilistic model for counting data, two distributions are mainly used that can only output integers and, thus, are suitable for count data: the Poisson distribution and the zero-inflated Poisson (ZIP) distribution. The Poisson distribution has one parameter while the ZIP distribution has two parameters.

2.2.1 A simple distribution for counts data: Poisson

Around the year 1900, Ladislaus von Bortkiewicz wanted to model the number of soldiers kicked to death by horses each year in each of the 14 cavalry corps of the Prussian army. As *"training data"*, they had data from 20 years. In the book *"The Law of Small Numbers"*, he modeled the number of soldiers kicked to death by horses by the Poisson distribution. This distribution assigns a probability to each possible outcome: a probability to observe zero events per unit or to observe one event per unit, two events per unit, and so on. Because it's a probability distribution, all probabilities add up to one. The average number of events per unit plays an important role because it defines the only parameter of the Poisson distribution, which is often called *rate* and is usually indicated in formulas with the symbol λ. There is a formula for the Poisson distribution that defines the probability for each possible value k of the counted events. The observed probability is:

$$P(y = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

where $k!$ indicates the factorial of k ($k! = 1 \times 2 \times 3 \times ... \times k$). Instead of a density, like in the gaussian case, a Poisson returns a real probability (sometimes also called *probability mass function*). A unique property of Poisson distributions is that λ not only defines the expected value, but also the variance. Thus, to define uniquely a Poisson distribution the only parameter to consider is λ. Another advantage of using a Poisson is that the model predicts outcome distributions that only assign probabilities to values that can actually be observed: Poisson distribution can only predict non negative outcomes while a gaussian distribution has domain in all $\mathbb{R}$ (in regression problems where the aim is, for example, to count the number of people queuing for the post office, a negative number doesn't make sense).

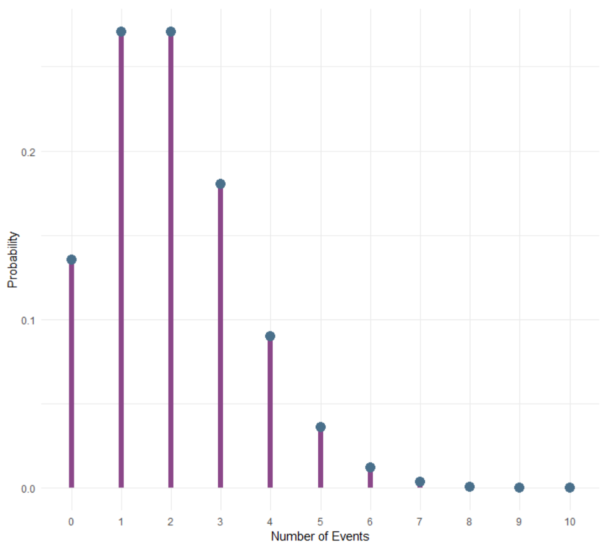


Figure 2.3 – Poisson distribution for the case $\lambda = 2$

2.2.2 Extending the Poisson distribution to a zero-inflated Poisson (ZIP) distribution

The zero-inflated Poisson (**ZIP**) [21] distribution takes into account the scenarios where there are many zeros, more than those expected in a Poisson distribution. In the ZIP distribution, you can model the excess of zeros by the introduction of a zero-generating process inspired by a coin tossing. The coin has a probability p to show heads. If this happens, than the outcome is zero; if not, a traditional Poisson distribution to predict the outcome is used. The Zero-Inflated Poisson distribution needs two parameters to be defined:

- The probability p of producing additional zeros

- The rate of a Poisson distribution

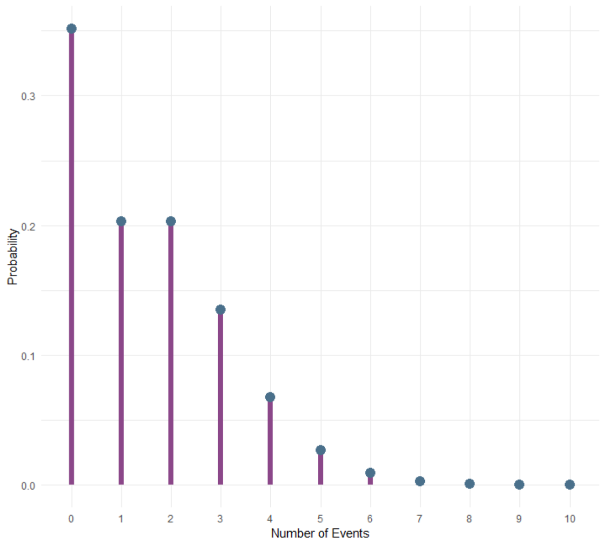


Figure 2.4 – Zero-Inflated Poisson distribution for the case $\lambda = 2$ and structural probability of zero cases = .25

2.2.3 Flexible distributions in modern architectures - A Discretized Logistic Mixture

Many real world data like audio samples or images come from high and complex dimensional distributions. One way to model complex distributions is to use mixtures of simple distributions e.g., Normal, Poisson, or logistic distributions.
Sometimes we face regression tasks in which we want to predict a discrete numerical outcome (e.g., counting data) with a complex distribution and values can vary within a defined range. A Poisson might be too simple for modelling the outcome. Therefore, the Google researchers find a way to use a mixture of distributions as CPD (Conditional Probability Distribution) [22].
The logistic distribution density has a bell-like shape, similarly to a Normal distribution. Fig. 2.5 shows the densities of the logistic functions with different values for the scale parameter on the left and the corresponding cumulative distribution function on the right. Therefore, an appropriate CDF for modelling counts data should model discrete values, however the logistic distribution is for continuous values without lower and upper limits. Therefore, the idea is to discretize the logistic distribution and clamp the values to a limited possible range. Thus, Quantization is the process that takes a continuous value and represent it using a single value from a discrete set and it is largely applied in the audio signal field.
To handle more flexible distributions, several quantized logistic distributions can be mixed. For the mixing, it can be used a categorical distribution that determines the weights (mixture proportions) of the different distributions in the mix. Fig. 2.6 shows the resulting distribution. This distribution can be easily extended to more flexible outcome distributions not mixing only two but, e.g., mixing two, four, or ten distributions together.

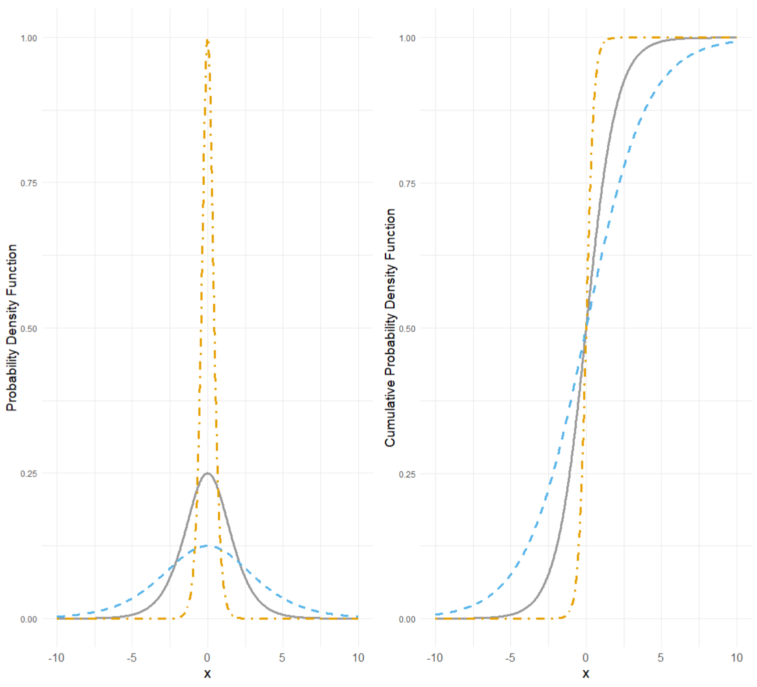


Figure 2.5 – Three logistic functions with 0.25, 1.0, and 2.0 values for the scale parameter. On the left is the probability density function (PDF) and on the right, the cumulative probability distribution function (CDF).

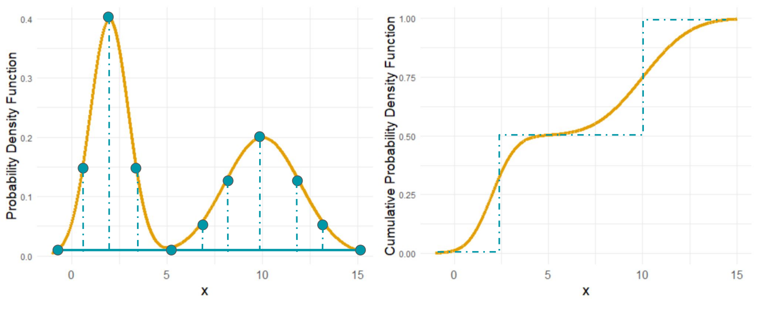


Figure 2.6 – The resulting discrete distribution when mixing two logistic distributions

2.3 The Bayesian Approach For Probabilistic Modelling

The idea of setting up models that incorporate the uncertainty about its parameter values via a probability distribution is quite old.
Thomas Bayes developed this approach in the 18^{th} century. Nowadays, its work is the foundation of the homonym research field in Statistics.
The Bayesian approach fit probabilistic models that capture different kinds of uncertainties. It's an alternative way of doing statistics and interpreting probability with respect to traditional frequentist methods.
In frequentist statistics, probability is a stranger concept (it is, indeed, replaced with confidence) and it is defined by analyzing repeated (frequent) measurements.
In Bayesian statistics, in contrast, probability is defined in terms of *degree of belief*. The more likely an outcome or a certain value of a parameter is, the higher the degree of belief in it.
Bayesian methods can be used to calculate the distribution of a model parameter given some data. The key step relies on Bayes' theorem. This theorem states, in mathematical notation, that

$$P(w|D) = \frac{P(D|w)P(w)}{\int P(D|w')P(w')\mathrm{d}w'} \tag{2.1}$$

where the terms mean the following:

1. D is some data, e.g. x and y value pairs: $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$. This is sometimes called the *evidence*.
2. w is the value of a model weight.
3. $P(w)$ is called the *prior*. This is our 'prior' belief on the probability density of a model weight i.e., the distribution that we postulate before seeing any data.
4. $P(D|w)$ is the *likelihood* of having observed data D given weight w.
5. $P(w|D)$ is the *posterior* density of the distribution of the model weight at value w, given training data. It is called *posterior* since it represents the distribution of our model weight *after* taking the training data into account.

The term $\int P(D|w')P(w')\mathrm{d}w' = P(D)$ does not depend on w (as the w' is an integration variable). It is only a normalisation term. For this reason, from this point on the Bayes' theorem will be written as

$$(P(w|D) = \frac{P(D|w)P(w)}{P(D)} \tag{2.2}$$

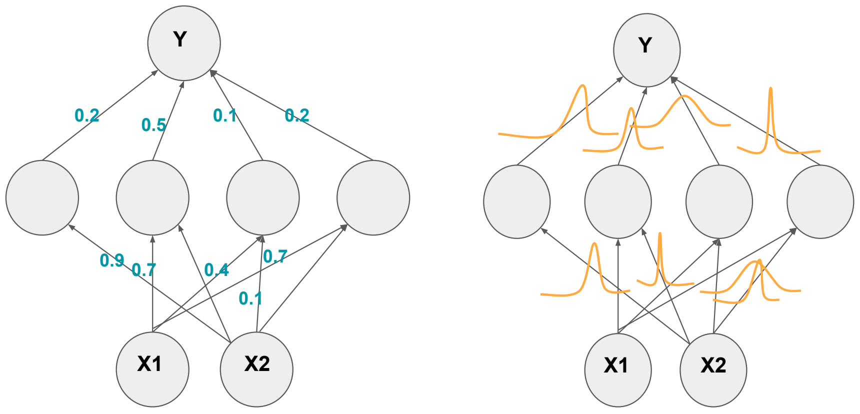


Figure 2.7 – Differences between a traditional Neural Network and the Bayesian one

Bayes' theorem gives us a way of combining data with some *"prior belief"* on model parameters to obtain a distribution for these model parameters that considers the data, called the *posterior distribution*.
To transpose the Bayesian framework into Deep Learning the main idea is the following. In a traditional neural network, as shown in the left side of Fig. 2.7, each weight has a single value. The true value of this weight is not certain. A lot of this uncertainty comes from imperfect training data, which might not exactly describe the distribution of the data where the sample was extracted from.

From a bayesian point of view, the idea is to include such uncertainty in deep learning models. This is done by changing each (neuron) weight from a single deterministic value to a *probability distribution*. The network will then learn the parameters of these distributions.
Consider a neuron weight w_i. In a standard (deterministic) neural network, this has a single value $\hat{w}_i$, learnt via backpropagation (see Appendix 7 for a briefly explanation of backpropagation).
In a neural network with weight uncertainty, each weight is represented by a probability distribution, and the backpropagation is used to learn the distribution *parameters*. Suppose, for example, that each (neuron) weight has a normal distribution. This has two parameters: a mean μ_i and a standard deviation σ_i.
To summarise:

- In classic deterministic Neural Networks: $w_i = \hat{w}_i$
- In Bayesian Neural Network weights uncertainty is represented by normal distribution: $w_i \sim N(\hat{\mu}_i, \hat{\sigma}_i)$.

Since the weights are uncertain, the feedforward value of some input x_i is not

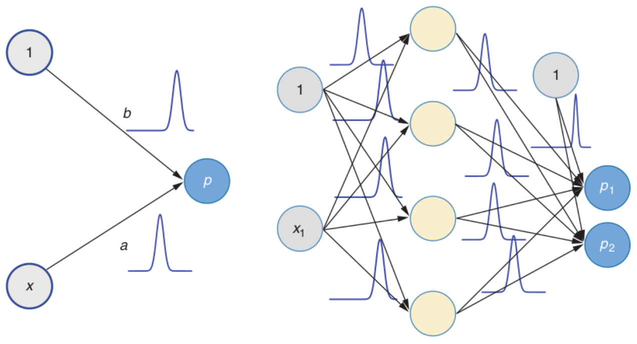


Figure 2.8 – A graphical representation of a Bayesian simple linear regression problem by a simple NN without hidden layer and only one output node. This provides an estimate for the expected value of the outcome (on the left). In a Bayesian variant of the NN for linear regression, distributions replace the slope (a) and intercept (b). This is also possible for deep networks, yielding a Bayesian neural network (BNN). A simple example of such a network is shown on the right side.

constant. A single feedforward value is determined in two steps:

- Sample each network weights from their respective distributions, this gives a single set of network weights.
- Use the sampled weights to determine a feedforward value $\hat{y}_i$.

Hence, the key question is how to determine the parameters of the distribution for each neuron weight.

2.4 Bayesian Neural Networks

The main idea of a Bayesian Neural Network is to apply the Bayesian approach described in the previous subsection to a traditional neural network architecture. Fig. 2.8 (on the left) shows the simplest Bayesian networks: **Bayesian linear regression**. Compared to standard probabilistic linear regression, the weights aren't fixed but follow a particular distribution. Intuitively, it's easy to transpose and expand this idea to a neural network with multiple layers and neurons. Fig. 2.8 (on the right) shows such a simple bayesian neural network.

Traditionally in statistics, to solve a simple linear regression problem like the one in Fig. 2.8 it is common to use an approach called Markov Chain Monte Carlo (MCMC for short) [23]. MCMC provides a way to sample from any probability distribution. Given the simple Bayes formula in Eq. 2.2 we are interested in sampling from the posterior, but even if it's easy to calculate the posterior in the functional form of $likelihood \times prior$, in most of the cases we are not able to calculate the evidence ($p(D)$).

Markov Chain Monte Carlo methods are a class of algorithms for sampling from a probability distribution based on constructing a Markov chain that has the desired distribution as its stationary distribution. The state of the chain after a number of steps is then used as a sample of the desired distribution. The quality

of the sample improves as a function of the number of steps.
The first MCMC algorithm was the *Metropolis-Hastings algorithm* [24] (see the Appendix 7 for a formal treatment).
It has the advantage that, given enough computations, it's exact. It also works for small problems, say 10 to 100 variables, but not for larger networks such as Deep Learning models with typically millions of weights.
There are two alternative approaches to obtain an approximation for the normalized posterior: one is the ***Variational Inference (VI)*** Bayes; the other is ***Monte Carlo (MC) dropout***. To build a Bayesian Neural Network `keras` and `Tensorflow Probability` [25] [26] [1] packages were used. The variational Bayes approach is welded into TFP, providing Keras layers to do the VI. MC dropout is a simple approach that can be easily done in standard Keras library as well.

2.4.1 Variational Inference

In situations where the analytical solution for a Bayesian Neural Network can't be determined or the use of the MCMC methods is not recommended due to the computational inefficiency, a technique to approximate the Bayesian model is needed: in this context, a Variational Inference [27] approach is useful.
It is useful to recall that, the intuition of combining the Bayes approach and Deep Learning is that with Bayesian Neural Networks, each weight is replaced by a distribution. Normally, this is quite a complicated distribution, and this distribution isn't independent among different weights. The idea behind the variational inference Bayes method is that the complicated posterior distributions of the weights are approximated by a simple distribution called *variational distribution.*
Variational Bayes methods approximate the posterior distribution with a second function, called a *variational posterior*. This function has a known functional form, and hence avoids the need to determine the posterior $P(w|D)$ exactly. Of course, approximating a function with another one has some risks, since the approximation may be very bad, leading to a posterior that is highly inaccurate. To mitigate this, the variational posterior usually has a number of parameters, denoted by θ , that are tuned so that the function approximates the posterior as well as possible.
Often Gaussians are used as parametric distributions (see Fig. 2.9). The Gaussian variational distribution is defined by two parameters:

- The mean;
- The variance.

[1] TensorFlow Probability (TFP) is a library for probabilistic modeling and inference in TensorFlow. It provides integration of probabilistic models with deep neural networks, gradient-based inference via automatic differentiation, and scalability to large datasets and models via hardware acceleration (e.g., GPUs) and distributed computation.

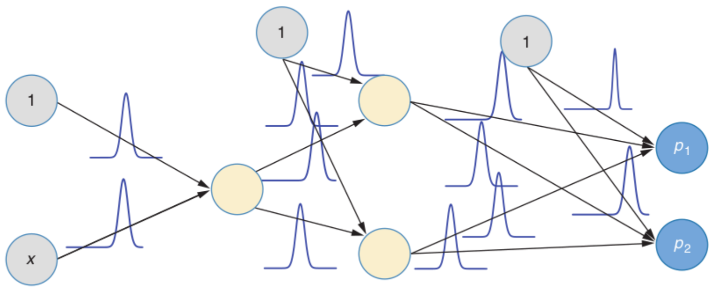


Figure 2.9 – A Bayesian network with two hidden layers. Instead of fixed weights, the weights now follow a distribution.

Instead of learning a single weight value w, the network has to learn the two weight distribution parameters: w_μ for the mean of the Gaussian and w_σ for the spread of the Gaussian. Besides the type of the variational distribution that is used to approximate the posterior, also a prior distribution is required. A common choice is to use the standard normal, N(0,1), as the prior [28]. Instead of $P(w|D)$, we assume the network weight has density $q(w|\theta)$, parameterized by θ. $q(w|\theta)$ is known as the *variational posterior*. We want $q(w|\theta)$ to approximate $P(w|D)$, so we want the *"distance"* between $q(w|\theta)$ and $P(w|D)$ to be as small as possible.
To get an intuition about the meaning of the different parameters in Fig. 2.9, let's focus on a deep Bayesian Neural Network where parameters θ replace the weights of the non-Bayesian variant of the neural network. Focusing on a single parameter θ in a Bayesian network, it isn't fixed but follows a distribution. Instead of determining the posterior directly, we approximate it with a simple, variational distribution, such $q_\lambda(\theta)$ as a Gaussian (see the bell-shaped density in Fig. 2.9). There are infinitely many Gaussians, but these make up only a subgroup of all possible distributions. The job of variational inference is to tune the variational parameter λ so that $q_\lambda(\theta)$ gets as close as possible to the true posterior $p(\theta|D)$.
This *"difference"* between the two distributions is usually measured by the *Kullback-Leibler divergence* D_{KL}.
The Kullback-Leibler divergence between two distributions with densities $f(x)$ and $g(x)$ respectively is defined as

$$D_{KL}(f(x)||g(x)) = \int f(x) \log\left(\frac{f(x)}{g(x)}\right) \mathrm{d}x.$$

Note that this function has value 0 (indicating no difference) when $f(x) \equiv g(x)$, which is the result we expect. We use the convention that $\frac{0}{0} = 1$ here.
Viewing the data D as a constant, the Kullback-Leibler divergence between $q(w|\theta)$ and $P(w|D)$ is hence

$$= \int q(w|\theta) \log \left(\frac{q(w|\theta)}{P(w|D)} \right) \mathrm{d}w$$

$$= \int q(w|\theta) \log \left(\frac{q(w|\theta)P(D)}{P(D|w)P(w)} \right) \mathrm{d}w$$

$$= \int q(w|\theta) \log P(D) \mathrm{d}w + \int q(w|\theta) \log \left(\frac{q(w|\theta)}{P(w)} \right) \mathrm{d}w - \int q(w|\theta) \log P(D|w) \mathrm{d}w$$

$$= \log P(D) + D_{KL}(q(w|\theta)||P(w)) - \mathbb{E}_{q(w|\theta)}(\log P(D|w)) \ (2.3)$$

where, in the last line, the following equality holds

$$\int q(w|\theta) \log P(D) \mathrm{d}w = \log P(D) \int q(w|\theta) \mathrm{d}w = \log P(D)$$

since $q(w|\theta)$ is a probability distribution and hence integrates to 1. If we consider the data D to be constant, the first term is a constant too, and we may ignore it when minimising the above. Hence, we are left with the function

$$L(\theta|D) = D_{KL}(q(w|\theta)||P(w)) - \mathbb{E}_{q(w|\theta)}(\log P(D|w))$$

Note that this function depends only on θ and D, since w is an integration variable. This function has a nice interpretation as the sum of:

- The Kullback-Leibler divergence between the variational posterior $q(w|\theta)$ and the prior $P(w)$. This is called the ***complexity cost***, and it depends on θ and the prior but not the data D.
- The expectation of the negative loglikelihood $\log P(D|w)$ under the variational posterior $q(w|\theta)$. This is called the ***likelihood cost*** and it depends on θ and the data but not the prior.

$L(\theta|D)$ is the loss function to minimise to determine the parameter θ. Note also from the above derivation, that we have

$$\log P(D) = \mathbb{E}_{q(w|\theta)}(\log P(D|w)) - D_{KL}(q(w|\theta)||P(w)) + D_{KL}(q(w|\theta)||P(w|D))$$

$$\geq \mathbb{E}_{q(w|\theta)}(\log P(D|w)) - D_{KL}(q(w|\theta)||P(w)) =: ELBO(2.4)$$

which makes sense because $D_{KL}(q(w|\theta)||P(w|D))$ is nonnegative. The final expression on the right hand side is therefore a lower bound on the log-evidence, and is called the *evidence lower bound*, often shortened to **ELBO**. The ELBO is the negative of our loss function, so minimising the loss function is equivalent to

maximising the ELBO.
Maximising the ELBO requires a tradeoff between the Kullback-Leibler term and the expected log-likelihood term. On the one hand, the divergence between $q(w|\theta)$ and $P(w)$ should be kept small, meaning the variational posterior shouldn't be too different from the prior. On the other hand, the variational posterior parameters should maximise the expectation of the log-likelihood $\log P(D|w)$, meaning the model assigns a high likelihood to the data. The above ideas are used to create a neural network with weight uncertainty, which is called a Bayesian Neural Network. From a high level point of view, this works as follows. Suppose we want to determine the distribution of a weight w of a particular neural network. The idea is to:

1. Assign to the weight a prior distribution with density $P(w)$, which represents our beliefs on the possible values of this network before any training data. This may be something simple, like a unit Gaussian.
2. Assign to the weight a variational posterior with density $q(w|\theta)$ with some trainable parameter θ.
3. $q(w|\theta)$ is the approximation for the weight's posterior distribution. θ is tuned to make this approximation as accurate as possible as measured by the ELBO.

The remaining question is then how to determine θ. Neural Networks are typically trained via a backpropagation algorithm, in which the weights are updated by perturbing them in a direction that reduces the loss function.
The aim is to do the same here, by updating θ in a direction that reduces $L(\theta|D)$. Hence, the function we want to minimise is

$$L(\theta|D) = D_{KL}(q(w|\theta)||P(w)) - \mathbb{E}_{q(w|\theta)}[\log P(D|w)]$$
$$= \int q(w|\theta)(\log q(w|\theta) - \log P(D|w) - \log P(w))\mathrm{d}w \quad (2.5)$$

In principle, we could take derivatives of $L(\theta|D)$ with respect to θ and use this to update its value. However, this involves doing an integral over w, and this is a calculation that may be very computationally expensive. Instead, we want to write this function as an expectation and use a **Monte Carlo approximation** to calculate derivatives. This function can be written as:

$$L(\theta|D) = \mathbb{E}_{q(w|\theta)}(\log q(w|\theta) - \log P(D|w) - \log P(w)) \tag{2.6}$$

However, taking derivatives with respect to θ is difficult because the underlying distribution and the expectation is taken with respect to θ. One way this can be handled is with the ***reparameterization trick***.

Briefly, the reparameterization trick is a way to move the dependence on θ around so that an expectation may be taken independently of it.
For example, suppose $q(w|\theta)$ is a Gaussian, so that $\theta = (\mu, \sigma)$. Then, for some arbitrary $f(w; \mu, \sigma)$,

$$\mathbb{E}_{q(w|\mu,\sigma)}[f(w;\mu,\sigma)]$$

$$= \int q(w|\mu,\sigma) f(w;\mu,\sigma) \mathrm{d}w$$

$$= \int \tfrac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\tfrac{1}{2\sigma^2}(w-\mu)^2\right) f(w;\mu,\sigma) \mathrm{d}w$$

$$= \int \tfrac{1}{\sqrt{2\pi}} \exp\left(-\tfrac{1}{2}\epsilon^2\right) f\left(\mu+\sigma\epsilon;\mu,\sigma\right) \mathrm{d}\epsilon$$

$$= \mathrm{E}_{\epsilon\sim N(0,1)}(f\left(\mu+\sigma\epsilon;\mu,\sigma\right))$$

where the change of variable $w = \mu + \sigma\epsilon$ is used. The dependence on $\theta = (\mu, \sigma)$ is now only in the integrand and the derivatives can be taken with respect to μ and σ:

$$\tfrac{\partial}{\partial\mu}\mathbb{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) = \tfrac{\partial}{\partial\mu}\mathbb{E}_{\epsilon\sim N(0,1)}(f\left(w;\mu,\sigma\right))$$
$$= \mathbb{E}_{\epsilon\sim N(0,1)}\tfrac{\partial}{\partial\mu} f\left(\mu+\sigma\epsilon;\mu,\sigma\right)$$
$$\tfrac{\partial}{\partial\sigma}\mathbb{E}_{q(w|\mu,\sigma)}(f(w;\mu,\sigma)) = \tfrac{\partial}{\partial\sigma}\mathbb{E}_{\epsilon\sim N(0,1)}(f\left(w;\mu,\sigma\right)) = \mathbb{E}_{\epsilon\sim N(0,1)}\tfrac{\partial}{\partial\sigma} f\left(\mu+\sigma\epsilon;\mu,\sigma\right)$$

Finally, the expectation can be approximated by its Monte Carlo estimate:

$$\mathbb{E}_{\epsilon\sim N(0,1)}\tfrac{\partial}{\partial\theta} f\left(\mu+\sigma\epsilon;\mu,\sigma\right) \approx \tfrac{1}{n}\textstyle\sum_i^n \tfrac{\partial}{\partial\theta} f\left(\mu+\sigma\epsilon_i;\mu,\sigma\right), \qquad \epsilon_i \sim N(0,1).$$

The above reparameterization trick works in cases where $w = g(\epsilon, \theta)$ i.e., where the distribution of the random variable ϵ is independent of θ.
Putting this all together and considering the loss function $L(\theta|D) \equiv L(\mu, \sigma|D)$, thus we have :

$$f(w;\mu,\sigma) = \log q(w|\mu,\sigma) - \log P(D|w) - \log P(w)$$

$$\frac{\partial}{\partial\mu} L(\mu,\sigma|D) \approx \sum_i \left(\frac{\partial f(w_i;\mu,\sigma)}{\partial w_i} + \frac{\partial f(w_i;\mu,\sigma)}{\partial\mu} \right)$$

$$\frac{\partial}{\partial\sigma} L(\mu,\sigma|D) \approx \sum_i \left(\frac{\partial f(w_i;\mu,\sigma)}{\partial w_i}\epsilon_i + \frac{\partial f(w_i;\mu,\sigma)}{\partial\sigma} \right)$$

$$f(w;\mu,\sigma) = \log q(w|\mu,\sigma) - \log P(D|w) - \log P(w)$$

where $w_i = \mu + \sigma\epsilon_i$, $\epsilon_i \sim N(0,1)$.
Actually, only a single sample ϵ_1 is usually taken for each training point. This leads to the following backpropagation scheme:

- Sample $\epsilon_i \sim N(0,1)$
- Let $w_i = \mu + \sigma\epsilon_i$
- Calculate
$$\nabla_\mu f = \frac{\partial f(w_i;\mu,\sigma)}{\partial w_i} + \frac{\partial f(w_i;\mu,\sigma)}{\partial \mu}$$
$$\nabla_\sigma f = \frac{\partial f(w_i;\mu,\sigma)}{\partial w_i}\epsilon_i + \frac{\partial f(w_i;\mu,\sigma)}{\partial \sigma}$$
- Update the parameters with some gradient-based optimiser using the above gradients.

This is how the network learn the parameters of the distribution for each neural network's weight.

2.4.2 Monte Carlo Dropout

Variational Inference allows one to fit a Bayesian Deep Learning model by learning an approximative posterior distribution for each weight. Most of the time a Gaussian is used to approximate the posterior (which is also the default in libraries such as Tensorflow Probability). These bayesian neural networks have twice as many parameters compared to their non-Bayesian versions, because each weight is replaced by a Gaussian weight distribution that's defined by two parameters (mean and standard deviation).
A desirable goal is to find a way to use a bayesian neural network with as many parameters as its non-Bayesian counterpart.
In 2015, a Ph.D. student, Yarin Gal, was able to show that the dropout method was similar to Variational Inference, allowing to approximate a bayesian neural network [29].
The traditional dropout [30] approach during training was introduced as a simple way to prevent a neural network from overfitting. When doing dropout during training, some randomly picked neurons in the Neural Network are set to zero. This is done in each update run. Mathematically speaking, each neuron has some probability p of being ignored, called the dropout rate. The dropout rate is typically set to be between 0 (no dropout) and 0.5 (approximately 50% of all neurons will be switched off).
Because the neurons are actually dropped, the weights of all connections that start from the dropped neuron are simultaneously dropped.
In most deep learning frameworks, this can easily be done by adding a dropout

layer after a weight layer and giving the dropout the probability p^* as an argument to the Dropout Layer.
The main idea behind this methodology is that fewer complex features are learned when using dropout. Because dropout forces the Neural Network to deal with missing information, it yields more robust and independent features.
During the prediction step, the observations go back to the full Neural Network with fixed weights, but considering only one detail: the learned weights need to be downweighed of $w^* = p^*w$. This down-weighting accounts for the fact that during training each neuron gets, on average, p^* less inputs than in the full Neural Network. The connections are, thus, stronger than these would be if no dropout is applied. During the application phase, there's no dropout and, hence, we down-weight the too strong weights by multiplying those with p^*.
During training, dropout can easily enhance the prediction performance. If we turn it on during testing we can use it as a Bayesian Neural Network.
As already said, in a Bayesian Neural Network, a distribution of weights replaces each weight. When using dropout, the weight distribution is simpler and essentially consists of only two values: 0 or W. The idea is that we can treat many different networks (with different neurons dropped out) as Monte Carlo samples from the space of all available models. To do that a dropout at test time is applied. Then, instead of one prediction, we get many, one by each model. We can then average them or analyze their distributions:

$$p(y|x_{test}, D) = \sum_i p(y|x_{test}, w_i)p(w_i|D) \tag{2.7}$$

The same input x from the test set is predicted T-times from a conditional probability distribution (CPD).
For each prediction, we get different CPDs,

$$p(y|x_{test}, w_i)$$

corresponding to a sampled weight constellation w_i.
The dropout probability p^* isn't a parameter but it is defined when the Neural Network is created. We turn on dropout during training and use the Negative Log-Likelihood loss function, which is minimized by tuning the weights via stochastic gradient descent (SGD). In Gal's framework [29], dropout is similar to fitting a Bayesian Neural Network via the Variational Inference method as illustrated in the previous subsection, except that the distribution from Fig. 2.10 is taken instead of the Gaussian usually used in the Variational Inference approach. By updating the w values, we actually learn the weight distributions that approximate the weight posteriors.
It's called MC dropout because for each prediction, we make a forward pass through another thinned version of the neural network, resulting from randomly dropping neurons. Starting from several dropout predictions we end up with a

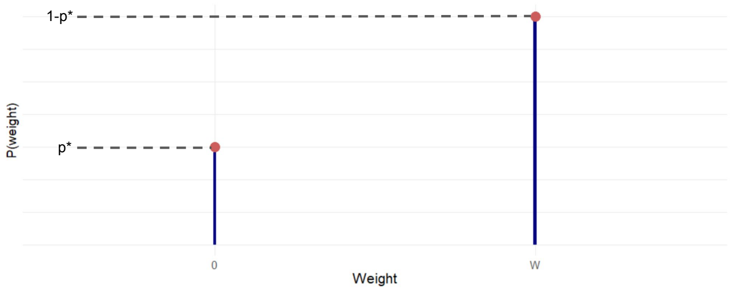


Figure 2.10 – The (simplified) weight distribution with MC dropout. The dropout probability p^* is fixed when defining the neural network. The only parameter in this distribution is the value of w.

Bayesian predictive distribution:

$$p(y|x_{test}, D) = \frac{1}{T}\sum_t^T p(y|x_{test}, w_t)$$

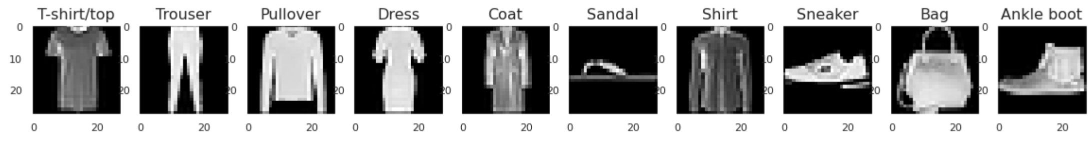
This is an empirical approximation to equation 2.7. The resulting predictive distribution captures both the epistemic and the aleatoric uncertainties.

The architectures discussed in Section 4.3 can be implemented both via Variational Inference and via MC Dropout. For computational efficiency point of view, results in Chapter 5 were obtained via the MC Dropout implementation.

2.4.3 Classification case study with novel classes

In 1.1, the *Elephant in the Office*'s classification task is discussed. When presenting a new input image to a trained classification neural network, for each class seen during the training phase a predicted probability is returned. To predict the Elephant image a high performance convolutional neural network (**VGG16**) trained on ImageNet completion images was used, but the model wasn't able to find the elephant in the office (the elephant wasn't among the top five classes either). In this subsection we want to highlight, through a simple case study, what are the differences in explaining uncertainties, between a Neural Network and a Bayesian Neural Network. Since training a BNN on ImageNet is computationally intensive, a small built-in dataset provided by keras is used: **The Fashion MNIST**.
This dataset is a collection of 60'000 images **28x28** in grayscale that represents 10 different categories of clothes. For this task, a traditional non-Bayesian neural network is implemented as well as two different types of BNNs, the first one using *Variational Inference* and the second one using the *MC Dropout* approach.
In order to see how these different types of architectures express uncertainty, we

drop all the images from one class (in particular, the *sneaker class*) from the training examples and leave it in the test examples.
When a sneaker image is shown to the trained Neural Network, the latter estimates probabilities for the classes on which it was trained. All of these classes are wrong, but the traditional architecture still can't assign a zero probability to all classes because the output needs to add up to one, which is enforced using the softmax layer.
The training set now features 9 classes and the goal is to investigate how differ-



ent networks express uncertainties both for known ad unknown classes. During test time, a traditional probabilistic CNN for classification yields a multinomial probability distribution for each input image. In this case, one that's fitted with nine outcome classes. The probabilities of the k classes provide the parameters for this multinomial distribution.
In the non-Bayesian NN, we have fixed weights, and for one input image, we get one multinomial *Conditional Probability Distribution*:

$$p(y|x, w) = MN(p_1(x, w), ..., p_9(x, w))$$

If the same image is predicted T times, we always get the same results.
In a Bayesian Neural Network fitted with **VI**, the fixed weights are replaced with *Gaussian distributions*. During test time, we sample from these weighted distributions by predicting the same input image not only once, but T times. For each prediction, we get one multinomial CPD: $p(y|x, w_t) = MN(p_1(x, w_t), ..., p_9(x, w_t))$. Each time we predict the image, we get a different CPD $p(y|x, w_t)$, corresponding to the sampled weight constellation w_t.
In the BNN fitted with MC dropout, fixed weights are replaced with binary distributions. For each prediction, we get one multinomial CPD: $p(y|x, w_t) = MN(p_1(x, w), ..., p_9(x, w))$. Each time we predict the image, we get a different Conditional Probability Distribution ($p(y|x, w_t)$) corresponding to the sampled weight constellation w_i. Looking at the network results predicted over a known class for two examples first, the following considerations can be done:

- In the left panel of Fig. 2.11, all networks can correctly classify the image of a boot.
  Taking a look at the unknown class image, all predictions in Fig. 2.11 are wrong because, as previously said, there was no sneaker images in the training set. But it can be observed that the BNNs can better express their uncertainty. Of course, the Bayesian networks also predict a wrong class in

all of their T runs. But what can be seen for each of the T runs is that the distributions vary quite a bit. When comparing the **VI** and **MC dropout** methods, VI shows more variation.

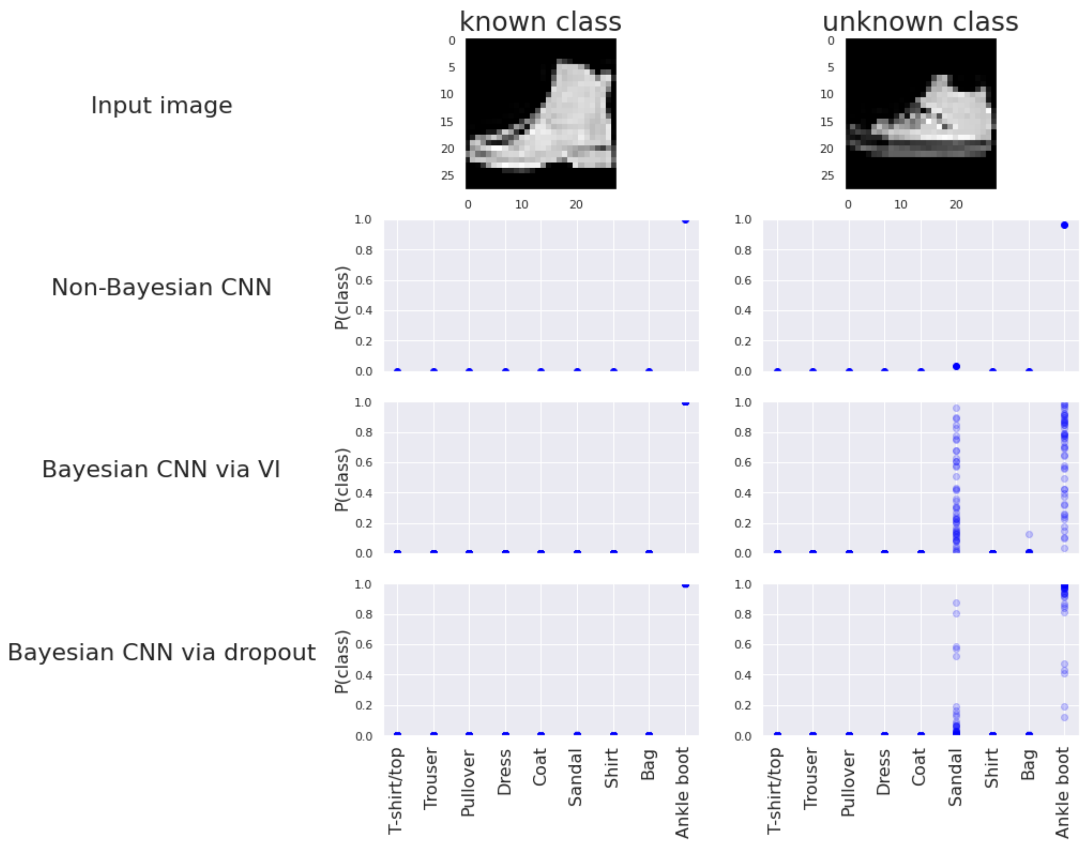


Figure 2.11 – Upper panel: images presented to train CNNs included an image from the known class *boots* (left) and an image from the unknown class *sneakers* (right). Second row of plots: corresponding predictive distributions resulting from non-Bayesian NN. Third row of plots: corresponding predictive distributions resulting from BNN via VI. Fourth row of plots: corresponding predictive distributions resulting from BNN via MC dropout.

In the case of traditional, non-Bayesian Neural Networks, we get one Conditional Probability Distribution for each image. We classify the image to the class with the highest probability: $p_{pred} = max(p_k)$. The Conditional Probability Distribution only expresses the aleatoric uncertainty, which would be zero if one class gets a probability of one and all other classes get a probability of zero. It can be used p_{pred} as a measure for certainty or $-log(p_{pred})$ as a measure for certainty or as a measure for uncertainty, which is the well-known NLL:

$$NLL = -log(p_{pred})$$

Another frequently used measure for the aleatoric uncertainty (using not only the probability of the predicted class) is entropy. There's *no epistemic uncertainty* when working with a non-Bayesian NN. Here's the formula:

$$Entropy : H = -\sum_{k=1}^{9} p_k log(p_k)$$

For each image, we predict T multinomial Conditional Probability Distributions:

$$MN(p_1(x, w_t), ..., p_9(x, w_t))$$

From a Bayesian point of view approach, for each class k, we can determine the mean probability

$$p_k^* = \frac{1}{T} \sum_{t=1}^{T} p_{kt}$$

We classify the image to the class with the highest mean probability

$$p_{pred}^* = max(p_k^*)$$

In the literature, there's no consensus on how to best quantify the uncertainty that captures both epistemic and aleatoric contributions. In fact, that's still an open research question (at least if we have more than two classes). Note that this mean probability already captures a part of the epistemic uncertainty because it's determined from all T predictions. Averaging causes a shift to less extreme probabilities away from one or zero. We could use $-log(p_{pred}^*)$ as the uncertainty:

$$NLL^* = -log(p_{pred}^*)$$

The entropy based on the mean probability values, p_k^* , averaged over all T runs is an even better-established uncertainty measure. But also, the total variance (sum of the variances for the individual classes) of the multidimensional probability distribution can be used to quantify the uncertainty:

- $Entropy^* : H^* = -\sum^9 p_k^* log(p_k^*)$
- Total Variance: $V_{TOT} = \sum^9 var(p_k) =$
  $= \sum^9 \frac{1}{T} \sum^T (p_{k_t} - p_k^*)^2$

Since we can calculate the uncertainty of every prediction with a Bayesian Neural Network, we could filter out the observations with high uncertainty and predict a new class, the sneaker class, which is something we can't do with traditional NNs.

Chapter 3

How to Model Spatial & Temporal Components - An Approach Based On Embeddings

In Section 3.1 limitations and disadvantages of dummy approach are discussed and in Section 3.2 different types of embeddings for modelling spatial and temporal static components are introduced (Entity Embedding & Node2Vec). Section 3.3 explains how to extract dynamical spatio-temporal embeddings from the latent space of a Variational AutoEncoder (VAE).

3.1 Limitations of the Dummy Approach

The operation of encoding categorical variables using a Dummy approach is actually a simple embedding where each category is mapped to a numerical vector of several 0s and a single 1 signaling the specific category.
The main drawbacks of a dummy approach are the followings:

- For high-cardinality variables - those who have a high number of categories - the dummy vector dimensionality becomes intractable leading to possible problems related to the *curse of dimensionality* [31]. Thus, for each additional category - referred to as an entity - a further dimension must be added to the dummy encoded vector. For example, if we are working with data related to all the Italian municipalities ($7'904$), to encode this information we will end up with a $(7'904 - 1)$-dimensional vector.

- The mapping is not informative: *"similar"* categories are not placed closer to each other in dummy vector space.
  Measuring similarity between dummy encoded vectors using the cosine distance, ends up with a similarity of 0 for every comparison between entities.

Solving these problems requires the use of more complex and efficient techniques.

3.2 Why Embeddings?

An embedding is a mapping of a *discrete or categorical* variable to a vector of continuous numbers. In Machine Learning context, embeddings are low-dimensional, learned continuous vector representations of discrete variables. Embeddings are useful because they allow one to reduce the category variable dimensionality and meaningfully represent categories in the transformed space.
The concept of embeddings comes from the **Natural Language Processing** field (NLP), where algorithms such as Word2Vec [32] [33], Glove [34] and FastText [35] are widely used to convert a word into a vector or "array of numbers". This methods are able to put words with similar meaning closer to each other in a word space, thus a more meaningful word representation compared to using the one hot encoding.
The traditional machine learning models rely on a tabular input that is feature engineered. Features could be numerical (discrete or continuous) or categorical. However, it is very hard capturing the spatial and temporal dimensions in this way. By using neural networks and embedding layers it is easy to capture meaningful representation of these 2 components.

3.2.1 Entity Embeddings

The idea of Entity Embedding [36] [37] is to map categorical variables into Euclidean spaces using a function approximation problem, therefore the categories are turned into *"entity"* (a.k.a. *category*) embeddings".
The goal of entity embedding is to map discrete values to a multi-dimensional space where values with similar function outputs are close to each other. The mapping is learned by a neural network as the one described in Fig. 3.1 during the standard supervised training process. Entity embedding not only reduces memory usage and speeds up neural networks compared with one-hot encoding, but it reveals the intrinsic properties of the categorical variables by mapping similar values close to each other in the embedding space. Entity embedding is useful as it helps neural network to better generalize when the data is sparse and statistics is unknown. As entity embedding defines a distance measure for categorical variables it can be used for visualization purposes for categorical data too.
In principle a neural network can approximate any continuous function and piecewise continuous function [38] [39]. However, it is not suitable to approximate arbitrary non-continuous functions as it assumes certain level of continuity in its general form. During the training step the continuity of the data guarantees the convergence of the optimization, and during the prediction step it ensures that slightly changing the values of the input keeps the output stable. Structured data with categorical features may not have continuity at all and even if it has a little bit, it may not be so obvious. The continuous nature of neural networks limits their applicability to categorical variables. Therefore, naively applying neural

networks on structured data with integer representation for categorical variables does not work well.
In Fig. 3.1 a neural network from which we extract the embedding representations is shown. The network works as follow:

- In the first layer (the yellow one) there are as many input layers as the number of categorical variables to be encoded (in this particular case 3 categorical variables). Before feeding into the Input layers, categorical features are One-Hot-Encoded or Label Encoded.

- In the second layer (the blue one) there are as many Embedding layer as the number of Inputs. These are the layers in charge of transforming the variables from integer to continuous.

- All embedding layers outputs are then concatenated. The merged layer is treated like a traditional input layer in a fully connected neural network. Other Dense layers can be stacked on top of it (in this case 2 Dense Layers with 128 and 32 neurons).

- An Output Layer with only one neuron, since we face a regression task.

The whole network can be trained with the standard backpropagation method. In this way, the entity embedding layer learns about the intrinsic properties of each category, while the deeper layers learn complex decision making criteria based on the embedding values. The embedding layer dimensions are hyperparameters that need to be pre-defined. There is a *rule of thumb* in these cases: the bound of the dimensions of entity embeddings are

$$min(number\ of\ categories/2, 50)$$

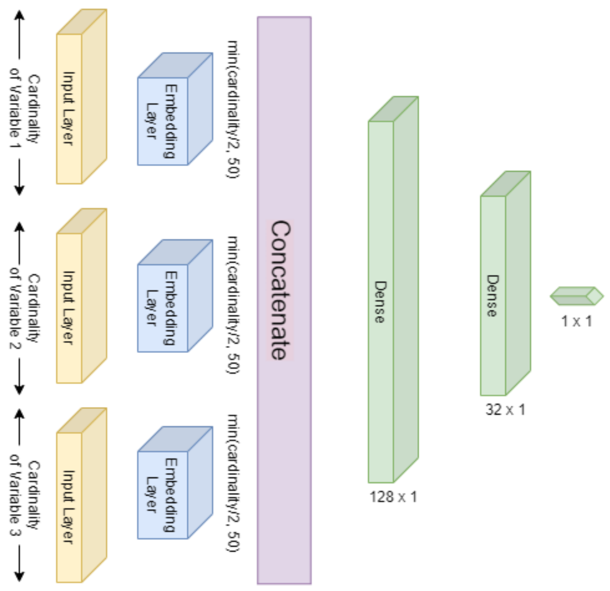
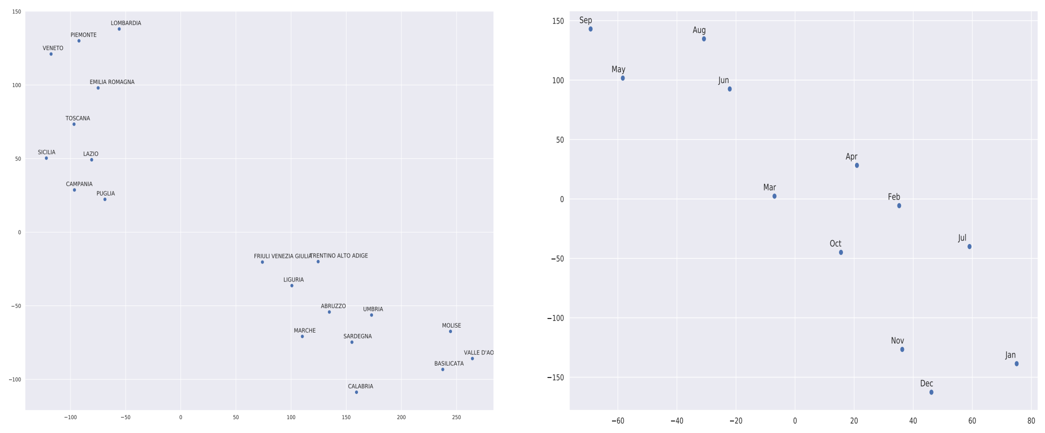


Figure 3.1 – A graphical representation of the neural network used for extract Embeddings

An example of an entity embedding application for real data can be seen in Figure 3.2. The two plots represent the spatial and temporal components (regions & months) extracted from the historical time-series of deaths due to COVID-19 for each of the 20 Italian regions. Starting from these data, a neural network was implemented as in Fig. 3.1. The embeddings were extracted as weights from the second layer. These embeddings (10-dimensional and 6-dimensional, respectively) were represented in 2D-dimension using t-SNE [40] algorithm for visualization purposes.
In short, the idea of t-SNE is that observations that are close together in high-dimensional spaces are also close in the two-dimensional space. Fig. 3.2 shows that the network seems to have learned the spatial and monthly components' characteristics:

- In Fig. 3.2 **(a)**, it can be observed that regions that have had a similar incidence of deaths from COVID-19 lie close together in the dimensional-reducted space, creating natural clusters between regions which can be intuitively justified: Lombardy, Veneto, and Piedmont are the most affected regions and lie in the reduced space in the top left of the plot. On the other hand, at the bottom right, regions that have had a number of deaths well below the average can be observed. In the middle, clusters of regions that share similar behaviors are visible.

- In Fig. 3.2 **(b)**, natural clustering can also be seen at the month level. In the lower right of the Fig. 3.2 there are months in which the incidence is, on average, higher, while in the upper left where it is lower.



(a) Plot of the 2D Regions' Embeddings

(b) Plot of the 2D Months' Embeddings

Figure 3.2 – A plot of the Embeddings extracted using the Entity Embedding Approach. For visualization purposes t-SNE was applied to represent Embeddings in 2 dimensions.

3.2.2 Working with Graphs - Node Embeddings

A graph can be defined as $G = (V, E)$ where V is a set of nodes and E is a list of edges. An edge is a connection between two nodes.
The goal of node embedding is to encode nodes so that similarity in the original network is approximated by similarity in the obtained embedding space (e.g., computed using embedding dot product). Most of node embedding algorithms generally consist of three steps:

1. Define an encoder (i.e., a function that maps nodes to embeddings). Fig. 3.3 illustrates the process. The encoder maps the nodes u and v to Z_u and Z_v in the Embedding Space.

2. Define a node similarity function (i.e., a measure of similarity in the original network), which specifies how the relationships in vector space map to the relationships in the original network.

3. Optimize the parameters of the encoder so that similarity of u and v in the network approximate the dot product between node embeddings: $similarity(u, v) \approx z_u^T z_v$

3.2.3 Spatial Static Embeddings with DeepWalk and Node2Vec

The *DeepWalk* [41] is an algorithm proposed for learning latent representations of vertices in a network. It uses a randomized path traversal technique to provide insights into localized structures within networks.
DeepWalk uses random path building through graphs to reveal latent patterns

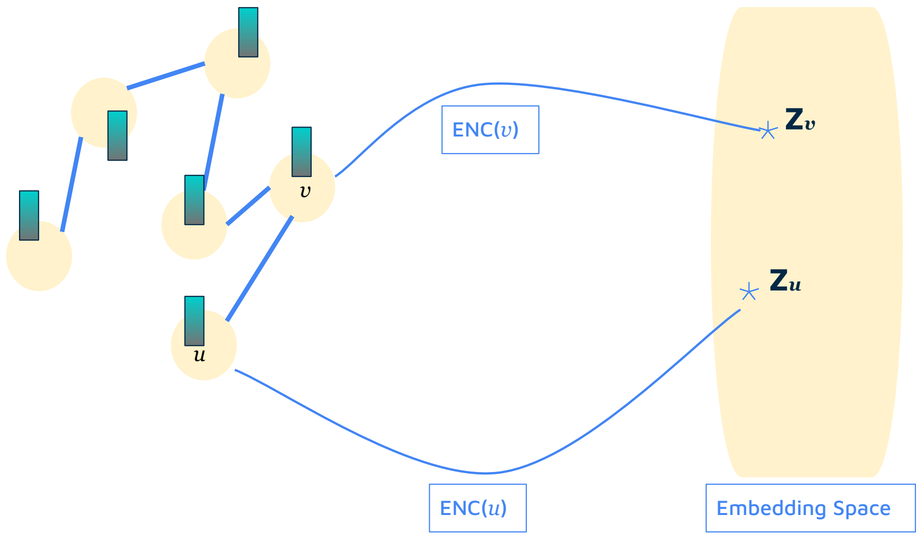


Figure 3.3 – shows how an encoder maps nodes from a graph (on the left) to an emebdding space.

in the network, these patterns are then learned and encoded by neural networks to produce embeddings.
These random paths are generated as follow:
starting from the target root, a neighbor is randomly selected and added to the path, then a neighbor of the latter node is randomly selected and added, this is repeated until the walk reaches the desired number of steps.
Fundamentally, a random walk is a way of converting a graph into a sequence of nodes which are used to train a **Word2Vec** [32] model. Basically, for each node in the graph, the model generates a random path of connected nodes. Then, a Word2Vec (skip-gram [42]) model is trained upon the random paths.

The **Node2Vec** [43] algorithm adds an additional step to the ideas presented by Deepwalk. It uses a combination of the Depth-first search (DFS) [44] and Breadth-first search (BFS) [45] algorithms to extract the random walks. This combination of methods is controlled by two parameters: **p** (*return parameter*) and **q** (*in-out parameter*).
The *bias in random walks* concept is presented before introducing the BFS and DFS like walks. As a result, walk sampling will be no more totally random. The simplest way to bias a random walk is by edge weighting. Starting from a node, some edges might be more likely to be selected in a walk. Edges can have weights and in this case, an edge weight describes the probability of the edge being selected or represents how strong a connection is between two nodes. Given some the edge weight $w_{u,v}$ between nodes u and v, the transition probability from node u to node v is $\pi_{u,v} = w_{u,v}$. If the graph is undirected, the same holds for the reverse transition i.e., $\pi_{u,v} = w_{u,v} = w_{v,u} = \pi_{v,u}$. In this way, an

initial definition of transition probability from one node to the other is given. A normalized transition probability for one node can be computed by normalizing all the node's weights. And by doing so for every node, we introduce the concept of bias.
We can extend the idea of network bias, introducing p and q as parameters of the graph.
In the node2vec algorithm, the parameters p and q control the level of exploration versus exploitation in the random walks that are used to generate node embeddings.
The parameter p determines the likelihood of immediately revisiting a node that was just visited in the previous step of the random walk. A high value of p means that there is a high likelihood of immediately revisiting a node, while a low value of p means that there is a low likelihood of immediately revisiting a node.
The parameter q determines the likelihood of transitioning to a node that is farther away from the current node in the graph, as measured by the shortest path distance between the nodes. A high value of q means that there is a high likelihood of transitioning to a node that is farther away, while a low value of q means that there is a low likelihood of transitioning to a node that is farther away.
Basically, if p is large it favors exploration and the random walks will be large, otherwise the algorithm will stay locally. q has a similar but opposite behaviour.

In this thesis, Embeddings have been implemented with Node2Vec approach. These Embeddings represent geographic information (the edges' weights of the graph represent the distance between locations). It is expected that, in contexts where the geographic location is important to explain the phenomenon (nearby sites have similar behaviours), these Embeddings are particularly informative.

3.3 Spatio-Temporal Embeddings with Variational Autoencoders

The general idea of autoencoders is pretty simple and consists in creating a neural network stacking both an encoder and a decoder, and then learning the best encoding-decoding scheme using an iterative optimisation process. So, at each iteration some data are fed, the encoded-decoded output is compared with the initial data and the errors are backpropagated through the architecture to update the network weights. Thus, intuitively, the overall autoencoder architecture (encoder+decoder) creates a bottleneck for data that ensures only summary information (e.g., latent variables) are learned from the input and later used to rebuild it.
Autoencoders can identify non linear relationships among input data and latent variables represented by embeddings.
After a traditional Autoencoder has been trained we could think that, if the

latent space is regular enough, we could take random points from that latent space and decode it to get a new content.
However the regularity of the latent space for autoencoders is a difficult point that depends on the distribution of the data in the initial space, the dimension of the latent space and the architecture of the encoder.
Why does this happen?
A dimensionality reduction with no reconstuction loss often comes with a price: the lack of interpretable and exploitable structures in the latent space (lack of regularity).
Suppose we consider an encoder and a decoder powerful enough to put any N initial training data onto the real axis and decode them without any reconstruction loss. In such case, the high degree of freedom of the autoencoder that makes possible to encode and decode with no information loss (despite the low dimensionality of the latent space) leads to a severe overfitting implying that some points of the latent space will give meaningless content once decoded.
This lack of structure among the encoded data into the latent space is pretty normal: the autoencoder is solely trained to encode and decode with as few loss as possible, no matter how the latent space is organised.
In the last years, deep learning based generative models have gained more and more interest due to some improvements in the field.
Variational Autoencoders (VAEs) [27] [46] are powerful generative models, now having applications in different fields, from generating fake human faces, to producing purely synthetic music.
The variational autoencoder (VAE) is a particular type of Deep Generative Model realized using an autoencoder neural network that consists of an encoder network and a decoder network, the former encodes a data sample to a latent representation and the decoder samples directly from latent variables. **Variational Autoencoders (VAE)**, differ in that the sampling is taken from a distribution parameterized by the latent variables.
In a nutshell, a VAE is an autoencoder whose encodings distribution is regularised during the training in order to ensure that its latent space has good properties allowing to generate new data.
To be clear, let's say we have an autoencoder with two latent variables and we draw samples randomly and get two samples of **0.4** and **1.2**. We then send them to the decoder for data generation.
In a VAE, these samples don't go to the decoder directly. Instead, they are used as a **mean** and a **variance** of a **Gaussian Distribution**, and the network use them to draw samples from the gaussian distribution to be sent to the decoder for data generation purpose. Gaussian distributions in VAEs are assumed to be **i.i.d.** and therefore covariance matrix will be diagonal.
The aim of VAE is to create a *nicely* distributed latent space where latent variables' distributions for different data classes are as follows:

- Evenly spread in order to have a better variation to sample from

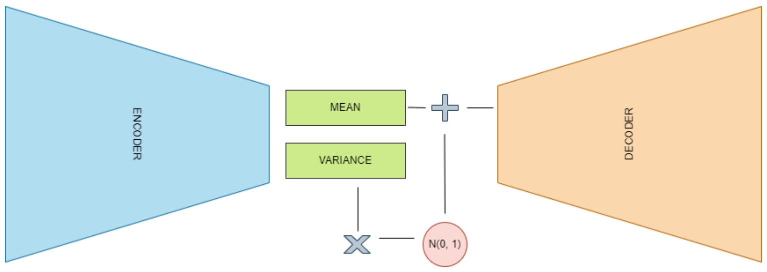


Figure 3.4 – Gaussian Sampling in a VAE

- Overlap slightly with each other to create a continuous transition

For variational autoencoders, the encoder model is sometimes referred to as the **recognition model** whereas the decoder model as the **generative model**.
When using generative models, the goal can be to generate a random, new output, that looks similar to the training data. But more often, the goal is to alter, or explore variations on available data, and not just in a random way either, but in a desired, specific direction.
As already said, when training an autoencoder, the encoded latent variables go straight to the decoder. With a VAE, there is an additional sampling step between the encoder and the decoder. The encoder produces the mean and variance of Gaussian distributions as latent variables, and we draw samples from them to send to the decoder. Here comes a problem: sampling is **not** back-propagable and therefore it is not trainable. To solve this, we can use a *reparameterization trick* where we cast the Gaussian random variable $N(mean, variance)$ into $mean + sigma \times N(0, 1)$. The sampling becomes an affine transformation and the error could be backpropagated from the output back to the encoder.
The sampling from a standard Gaussian Distribution **N(0, 1)** can be seen as an independent input to the VAE, and do not need to backpropagate back to (the training) inputs.
This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.
Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the *"area"*, how much the encoding can vary from the mean (as shown in Fig. 3.5). As encodings are generated at random from anywhere inside the *"circle"* (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well. This allows the decoder to decode not only specific encodings in the latent space (leaving the decodable latent space discontinuous), but to manage slightly variations, as the decoder is exposed to a range of variations during training.
The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale i.e., for similar samples.

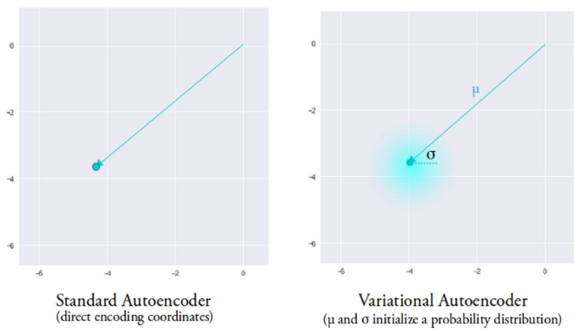

Standard Autoencoder
(direct encoding coordinates)
Variational Autoencoder
(μ and σ initialize a probability distribution)
μ
σ

Figure 3.5 – Difference between an Autoencoder and a Variational Autoencoder in the latent space from an intuitive point of view.

The loss function that is minimised when training a VAE is composed of a *"reconstruction term"* (on the final layer), that tends to make the encoding-decoding scheme as much performant as possible, and a *"regularisation term"* (on the latent layer), that tends to regularise the organisation of the latent space by making the distributions returned by the encoder close to a standard normal distribution.
The other term is what it is typically used for simple autoencoders to compare the generated data with the input data. This is called the ***reconstruction loss***, which measures the difference in reconstructed data with the target data. This can be either **Binary-crossentropy** or **Mean Squared Error**.
The total loss is made up of the Kullback-Leibler loss plus the reconstruction loss:

$$loss = ||x - x_{reconstructed}||^2 + KL[N(\mu_x, \sigma_x), N(0,1)]$$

The regularisation term is expressed as the Kullback-Leibler divergence between the returned distribution and a standard Gaussian.
VAEs can be used as a regularization technique to prevent overfitting in deep learning models. By learning a continuous latent space, VAEs can encourage the model to learn more general features that are applicable to a wider range of inputs, rather than just memorizing the training data. Some of the main advantages of regularization in VAEs include:

- **Improved generalization**: By encouraging the VAE to learn more general features, regularization can help the VAE to generalize better to new data points that it has not seen before. This can lead to better performance on the test set, and make the VAE more robust to changes in the data distribution.

- **Reduced overfitting**: Overfitting occurs when a model becomes too closely tied to the training data, and is unable to generalize to new data points. By using regularization, VAEs can avoid overfitting and achieve better performance on the test set.

- **Better interpretability**: VAEs that are regularized can be more interpretable because the latent space is more likely to have a clear meaning. For example, in an image generation VAE, the latent space might represent different attributes of the image, such as shape, color, and texture. This can make it easier to understand what the model is doing and how it is making decisions.

- **Improved stability**: Regularization can also help to improve the stability of VAEs, by encouraging the model to learn more robust and stable features. This can make the VAE less sensitive to small variations in the training data, and can help to prevent the model from learning features that are not meaningful or relevant to the task.

In this thesis a variational autoencoder has been used to extract spatio-temporal embeddings from the latent space. The autoencoder was built by stacking LSTM layers as described in Fig. 3.4. The idea is that observations that share the same characteristics have a similar latent representation.

Chapter 4

Neural Networks for Time Series Data - How to Model Sequential Data

In Section 4.1 Convolution operation and Convolutional layers are introduced, while in Section 4.2 Neural Networks' architectures for modelling sequential data are discussed, in particular LSTM, GRU, and Self-Attention layers are reviewed. In Section 4.3 a Neural Network architecture able to take into account spatio-temporal informations and uncertainty in predictions, named Bayesian Spatio-Temporal Neural Network **(BSTNN)**, is proposed and described.

4.1 Convolutional Neural Networks

Many problems involve forecasting or classification working on univariate time series data. Unfortunately, there is frequently a large amount of noise in time series data in addition to the signal.
In time series analysis, some sort of smoothing technique is used prior to analysis. One could apply a moving average to smooth a time series, and then apply a forecasting or classification technique after that. Convolutional neural networks [47] (**CNN**) provide an architecture to learn smoothing parameters.
Even if convolutional layers were initially developed for computer vision tasks, their shift-invariant characteristics have allowed convolutional layers to be applied in other contexts, such as natural language processing, time series, recommender systems, and signal processing.
Convolution is a linear operation that involves multiplying an input with weights to produce an output. The multiplication is performed between an array of input data and an array of weights, called kernel. The operation applied between the input and the kernel, is a sum of an *element-wise* dot product. From a pragmatic point of view, suppose a simple 1-Dimensional Convolution (**Conv1D**) operation is the one reported in Fig. 4.1 where there is an input array of six elements and a

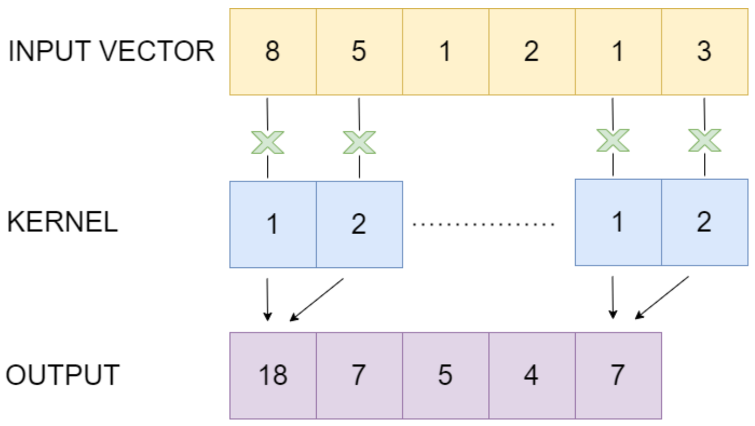


Figure 4.1 – An easy example of 1D convolution. The kernel (blue) is shifted along the 1 dimension of the input vector (yellow) and returns an output value based on a sum of an element-wise dot product.

kernel of two. We will multiply each consecutive element pair in the input vector by the kernel, one pair after another, shifting the kernel one position ahead until the end of the input vector, and get the output vector. Conv1D layers can learn to identify patterns or features in the time series data by sliding a window of weights (also known as a kernel or filter) over the input data and performing a dot product between the weights and the input data at each position. This process is repeated for each position in the input, and the output is a series of feature maps that capture different patterns in the input data. Furthermore, they can learn to recognize patterns in the data without having to consider the entire time series at once, which can be useful for both long data sequences and for real-time data processing.
In summary, Conv1D layers are often used for time series data because they are well-suited for processing one-dimensional input, and are able to learn local features from the data in an efficient manner.

4.2 Recurrent Neural Networks

A Recurrent Neural Network (**RNN**) is a type of neural network that is designed to process sequential data, such as time series, natural language, or speech. Usually, neural networks are *feedforward* (activations flow only in one direction, from the input layer to the output layer). RNNs are called “recurrent” because they use feedback connections (internal loops) to allow information from previous time steps or input elements to be used in the computation of the current output. In a RNN, the input at each time step is processed by a hidden layer, and the output of the hidden layer at each time step is passed as input to the next time step. This allows the network to maintain an internal state or memory of the input sequence, which can be useful for tasks that require understanding of the context and dependencies between the elements in the sequence. Each recurrent neuron (which is also called a *memory cell*) has two sets of weights:

one for the inputs x_t and the other for the outputs of the previous time step, y_{t-1}. Calling these weights as w_x and w_y, the output is computed by the equation:

$$y_t = \phi(w_x^T x_t + w_y^T y_{t-1} + b)$$

where b is the bias vector and ϕ is the activation function.
In the Keras library, there are different implemented recurrent layers: from the simplest one (`SimpleRNN`) to the more complex `LSTM` and `GRU`. These variations differ in the way they process the input and maintain the internal state, and are chosen based on the specific requirements of the task at hand. In practice, `SimpleRNN` is too simplistic to be of real use: just like any deep neural network, when using a simple RNN to run over many time steps behind, it may suffer from the *vanishing/exploding gradient problem* and take forever to train. The `LSTM` and `GRU` layers are designed to solve these problems.

4.2.1 LSTM & GRU

The *Long Short-Term Memory* (LSTM) cell was proposed for the first time in 1997 by Seep Hochreiter and Jürgen Schmidhuber [48] and it was the culmination of their research on the vanishing gradient problem. The *Gated Recurrent Unit* (GRU) [49] cell is a modern and simplified version of the LSTM cell and it seems to perform just as well.
LSTM and **GRU** layers are a variant of the **SimpleRNN** layer. The main advantage of using LSTM and GRU networks is that they are able to effectively capture long-term dependencies in the data by using special units called gates to control the flow of information through the network. These gates allow the network to selectively retain or forget information from previous time steps, which enables the network to maintain a more stable internal state and better preserve the relevant information for the current task. Without claiming to go deeply into this topic, the most important thing to keep in mind is what these cells meant to do: they allow past information to be reinjected at a later time, thus fighting the vanishing-gradient problem, which is a common issue when training traditional RNNs on long sequences of data.
Furthermore, both LSTM and GRU networks are generally more efficient to train compared to traditional RNNs, due to their ability to effectively capture long-term dependencies and maintain a stable internal state. This can be particularly useful for tasks with long sequences of data or large datasets.

4.2.2 Self-Attention Layers for Long Term Predictions

Feeding huge datasets to train models, it is possible that a few important parts of the data might be ignored by the models. Paying attention to important information is necessary and it can improve the performance of a model. This can be achieved by adding an additional attention mechanism to the models. Using attention layers in neural network's architecture can improve significantly

the network's performance memorizing long sequences of the information or data [50].
A self-attention module takes in n inputs and returns n outputs.
What happens in this module?
Briefly, from an intuitive point of view, the self-attention mechanism allows the inputs to interact with each other ("self") and find out who they should pay more attention to ("attention"). The outputs are aggregates of these interactions and attention scores.
Self-attention layers can be useful for time series data because they allow the model to weight different parts of the sequence differently when making a prediction. This can be helpful because it allows the model to focus on the most important parts of the sequence when making a prediction, rather than simply averaging or summing over all parts of the sequence.

4.3 BSTNN - A Bayesian Spatio-Temporal Neural Network

The goal of this thesis is to build a neural network's architecture that is able to learn patterns starting from data moving over space and time (unlike traditional "ad hoc" architectures that can only manage temporal information), and which is able to take uncertainty into account in its forecasts like a traditional Bayesian-statistical model.
The architecture thus built will be referred to hereafter as a Bayesian Spatio-Temporal Neural Network (**BSTNN**). In the following paragraph, 2 different architectures will be introduced.
Even if the architectures are slightly different, they share the same *"modus operandi"*: they accept the historical series of data and the temporal and spatial embeddings as input and they return the forecasts for the following T temporal istants.
One of the characteristics of these architectures is that, even as the forecast interval increases, they are able to provide satisfactory predictions with low errors scores compared with other traditional models, as will be discussed in Chapter 5. Fig. 4.2 and Fig. 4.3 show the two types of architectures, they essentially vary from the use of LSTM layers (followed by a self attention layer) stacked subsequently to the concatenation of the outputs of the Conv1D layers. The generated embeddings can be derived from the 3 methods described in Chapter 3.

Model Architecture

This paragraph introduces the two types of architectures implemented. They are very similar and share most of the network skeleton.
Suppose we want to analyze the temporal trend of a particular phenomenon in

N different sites. Formally, for each site $j = 1, ..., N$ a phenomenon is observed in T temporal instants and we want to make predictions for the following $T + h$ temporal instants.
The architectures (Fig. 4.2, and Fig. 4.3) are structured as follows:

In the first layer there are N inputs, as many as the sites considered. Each of these goes through two convolutional layers (**Conv1D**) which create a temporal smoothing of the time series. The output of each of the last convolutional layers is flattened [1] and:

1. Focusing on the architecture represented in Fig. 4.2, the convolved data are concatenated to the temporal and spatial embeddings (the red box). For each time instant the same input data will be fed to the network N times, each time coupled with a different spatial embedding, whereas the embeddings are unique for each site.
   This is done for a twofold reason. The network obtains as input context information on the temporal trend of each site via the temporal sequences and embeddings and is forced to discriminate each location on the basis of the embeddings: if these are informative, the network will be able to discriminate the trend of each site.
   Once the information of the embeddings in input to the network has been inserted and they have been concatenated with the information coming from the convolutional layers, dense layers are stacked with decreasing number of neurons, up to the output layer (a dense layer or a probabilistic layer based on discretized logistic mixtures as already discussed in 2.2.3).
   The rationale of providing both data and spatial embeddings as network input is that the embedding will teach the network how to compute the output for a specific site, while taking into account the input data collected in several sites.

2. Focusing on the architecture represented in Fig. 4.3 a reshaping is performed to fed convolutional layers' otputs to recurrent layers. In this case LSTMs are stacked, followed by an attention layer to extract sequential information from the data. After that, the result is concatenated to the information contained in the embeddings. They are followed by dense stacked layers, up to the output layer.

In the first architecture **(1)**, the sequential information is managed only by the Convolutional layers which process a smooth of the time series, without the network being able to capture long-term dependencies on the data through recurrent layers. For this reason, in the first architecture, the embeddings generated from VAE are added as input. Since in the second architecture **(2)** the LSTM layers should process sequential information, it is preferred not to insert the embeddings generated from the VAE to avoid possible redundant informations.

[1]Flattening is a procedure required to prepare data for the subsequent layers

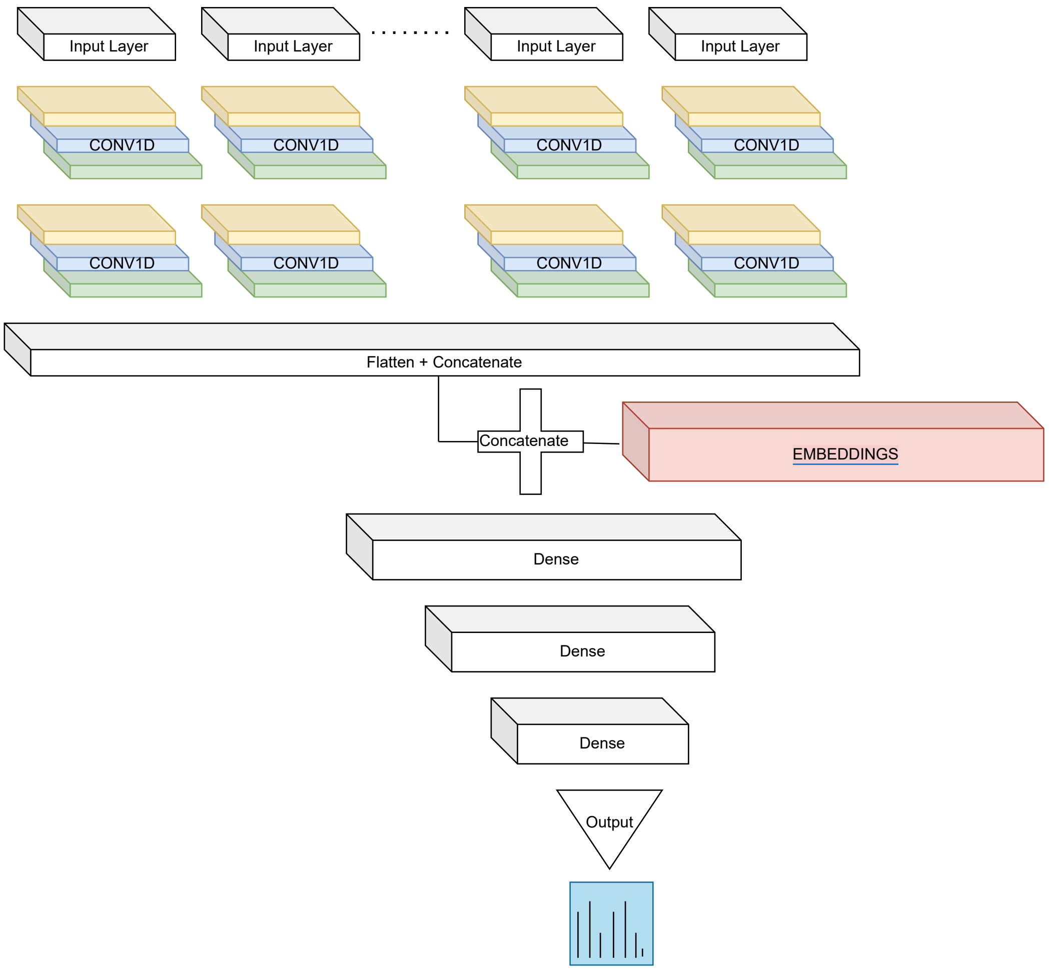


Figure 4.2 – BSTNN with Dense stacked layers after stacked Con1D layers, without LSTM layers

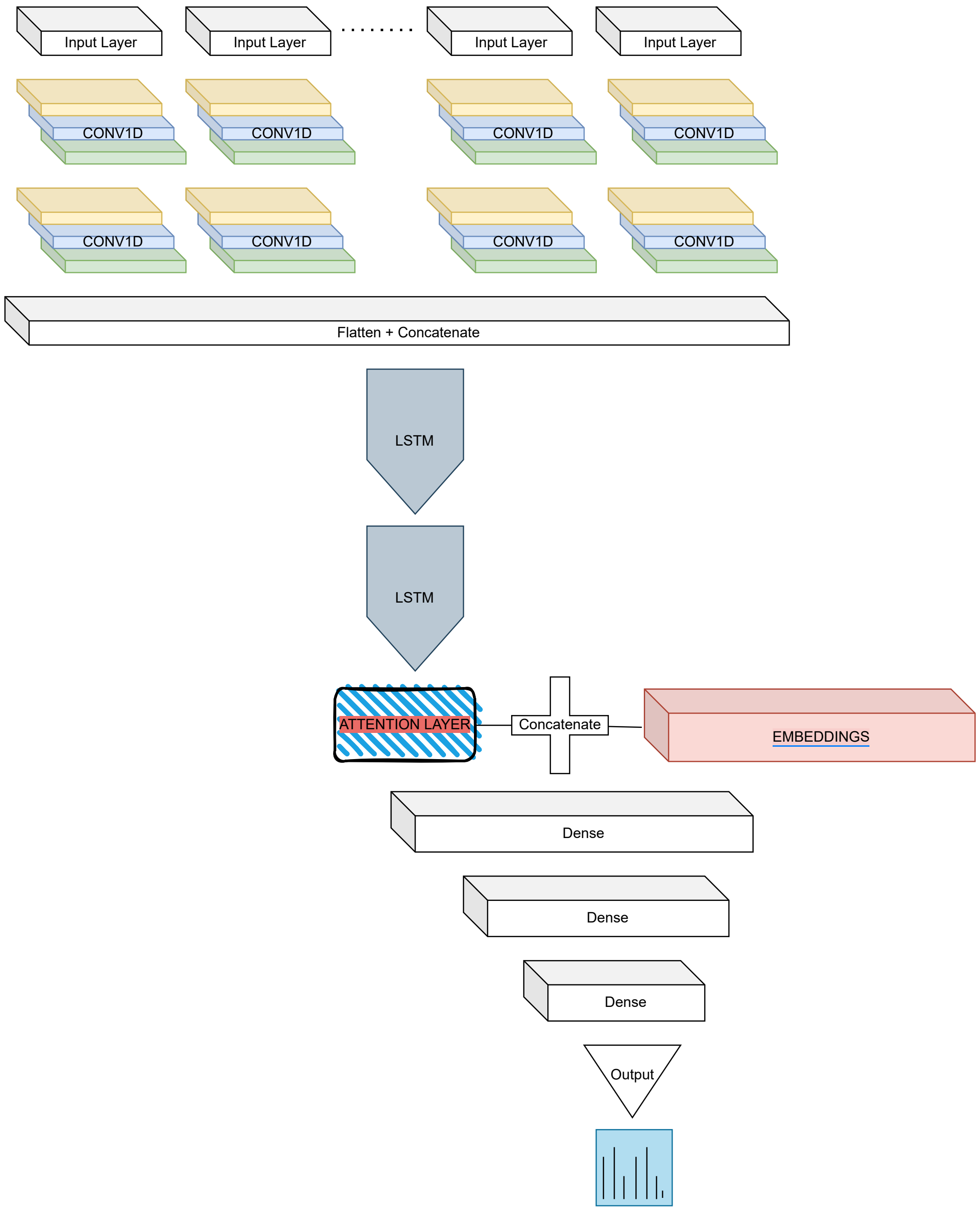


Figure 4.3 – BSTNN with stacked Dense and LSTM layers after stacked Con1D layers

Chapter 5

A Bayesian Spatio-Temporal Neural Network Applied on Real Data

In this Chapter BSTNN and other state of the art models are evaluated in 2 different datasets: in Section 5.1 data from Out-Of-Hospital Cardiac Arrests (OHCAs) in Ticino region, Switzerland, are considered, while in Section 5.2 data from New Daily-Deaths from COVID-19 in Italy are used.

5.1 An Easy Application: Forecasting Monthly OHCAs in Switzerland's Cantons

The adoption of spatial statistical tools for the analysis of cardiac arrests is gaining more and more attention in the medical literature.
The data used to evaluate the models described in Chapter 4 refer to the number of Out-Of-Hospital-Cardiac Arrests (**OHCA**s) that occurred in the 26 Swiss cantons, on a monthly basis, from January 2019 to October 2020 (**572** observations in total).
The idea is to use the months from January 2019 to August 2020 as training and the last 2 months to evaluate and compare the performance of different models. Two different models are used for comparison purposes with the BSTNN (Bayesian Spatio-Temporal Neural Network) introduced in Chapter 4:

- An Integrated Nested Laplace Approximation approach (**INLA** [5]), a well-suitable method for handling spatio-temporal data. INLA uses a combination of analytical approximation and numerical integration to obtain approximated posterior distributions of the parameters that can then be post-processed to compute quantities of interest like posterior expectations and quantiles. The class of models handled by INLA is wide and flexible ranging from generalized linear mixed to spatial and spatio-temporal mod-

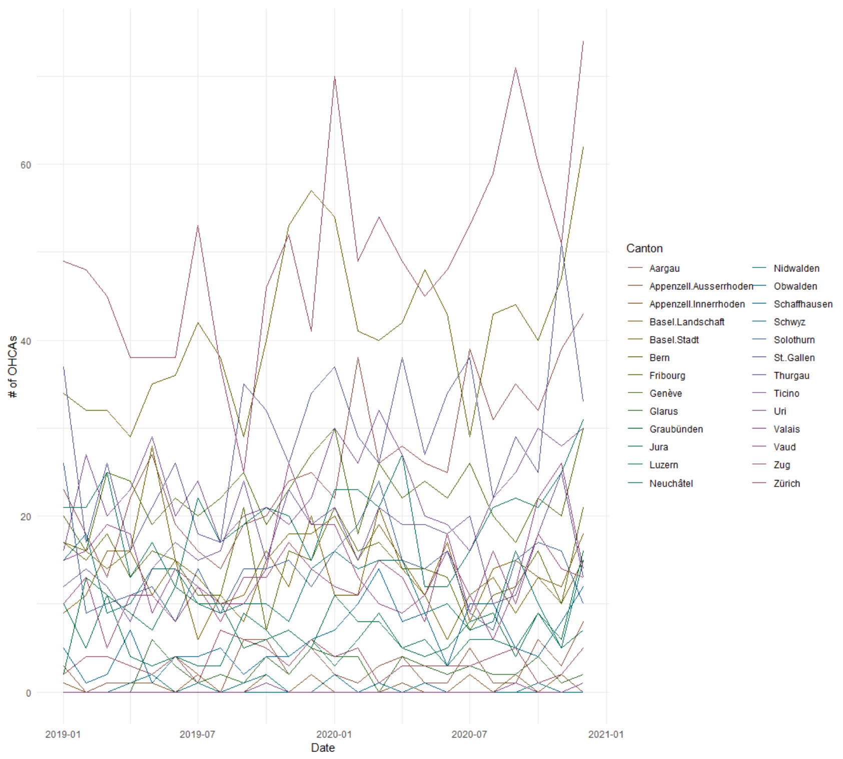


Figure 5.1 – Time Series of monthly OHCAs conditioned on the Canton

els.

- **LSTM** (already discussed in 4.2.1) applied to univariate time series of each region.

The Fig. 5.1 shows the time series of each canton. It is observed that there is a strong difference between cantons in terms of numbers of Out-Of-Hospital Cardiac Arrests, some are close to zero, while others can count more than 70 monthly. Since trends are very different from each Canton to others, it is useful to exploit representations that can provide information not only about the single Canton, but also on the "*context*" characteristics, i.e., how similar they are to other Cantons. The Fig. 5.2 shows the representation of the embeddings (generated through Entity Embedding) related to the different Cantons in the 2D space (the starting space is reduced from 13-Dimensional to 2-Dimensional through t-SNE). It is observed that cantons that share the same properties are close in the reduced space. These natural clusters make sense as can be seen from the adjacent time series plot in Fig. 5.2: *Bern* and *Zug* in the upper left are the cantons with the highest number of monthly cardiac arrests, while the group in the lower right (*Uri*, *Obwalden*, *Appenzell Innerrhoden*, *Schaffhausen* and *Nidwalden*) have values close to zero.

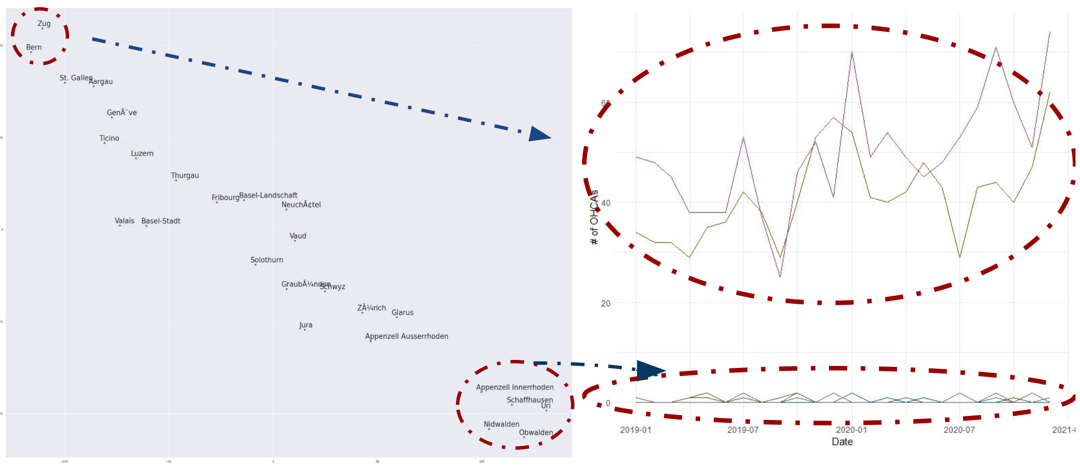


Figure 5.2 – On the left, the graphical representation of 2D Entity Embeddings after a dimensionality reduction operated by t-SNE. It can be observed that clusters of Cantons that lie together in the Embedding Space share similar behaviours.

5.1.1 Evaluation Metrics

The following metrics are used to evaluate the models on the test set:

- **Mean Absolute Error (MAE)**

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

  MAE measures the average magnitude of the errors in a set of forecasts, without considering their direction. It is chosen because it is easy to interpret.In simple terms, the MAE is the average over the test sample of the absolute values of the differences between forecast and the corresponding observation.

- **Root Mean Squared Error (RMSE)**

$$\sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

  RMSE is a quadratic scoring rule that also measures the average magnitude of the error. It's the square root of the average of squared differences between prediction and actual observation. Focusing on the square root of the average squared errors, it has some interesting implications for RMSE. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE should be more useful when large errors are particularly undesirable.

5.1.2 Results & Comparisons

Several models have been trained with the two architectures discussed in the previous chapter in which different embeddings have been fed as input:

- Both Entity Embedding representing a temporal statistical component (embeddings relating to months) and a spatial component (relating to the behavior of each region);

- Embeddings created with **Node2Vec** from a graph synthesizing geographical information. Specifically, a set of nodes representing several cantons, are connected with arcs whose weights represent the Canton's Capital distance. The embeddings generated by **Node2Vec** summarises the graph geographical information. It is expected that the more important the geographic information is, the more informative these embeddings are;

- Embeddings extracted from the VAE represent a synthesis of the sequential information at the cantonal level. For this reason they are only used in architecture that does not contain LSTM layers.

Firstly, to evaluate how the embeddings contribute to the prediction of the neural network it was decided to evaluate the performance of the BSTNNs with the single embeddings fed as input to the neural networks and evaluate the performance. Since the information from the embeddings represents different spatial characteristics of the cantons (thus, it is not redundant) it is expected that, if inputted simultaneously, the embeddings can contribute to better predictions scores.
Table 5.1 shows the scores (MAE and RMSE) on the test set of the models used for the forecasting task. It can be observed that BSTNN performs significantly better than the traditional models used in the literature, independently from the embeddings used. BSTNN without LSTM layers (see Fig. 4.2) + All the Embeddings is the model with the lowest MAE score, while BSTNN with LSTM layers (see Fig. 4.3) + Entity Embeddings + Node2Vec Embeddings is the model with the lowest RMSE score.
Fig. 5.3 shows the MAE scores of only four models (to make easy the plot visualization): the two best BSTNN models as shown by Table 5.1, INLA, and LSTM from a Canton point of view. It can be observed that BSTNN is able to better manage the cantons' predictions affected by high noise. While INLA and LSTM have extremely high errors, BSTNN manages to mitigate the high error (i.e., Aargau and Zug, respectively, first and last bars), introducing uncertainty in the forecasts as can be seen in Fig. 5.4.

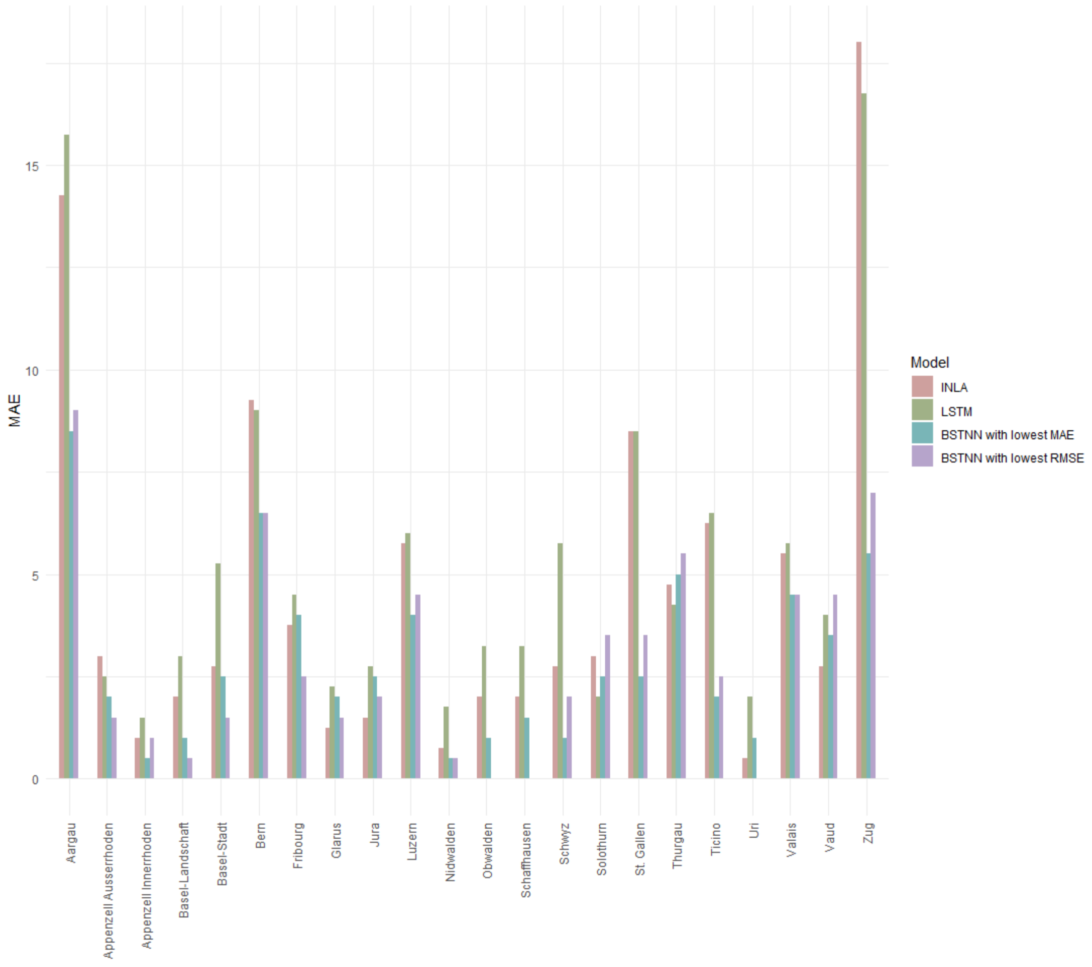


Figure 5.3 – The graph shows error scores (MAEs) for each Canton conditioned on the models used (INLA, LSTM, BSTNN without lowest MAE score, and BSTNN with lowest RMSE score.

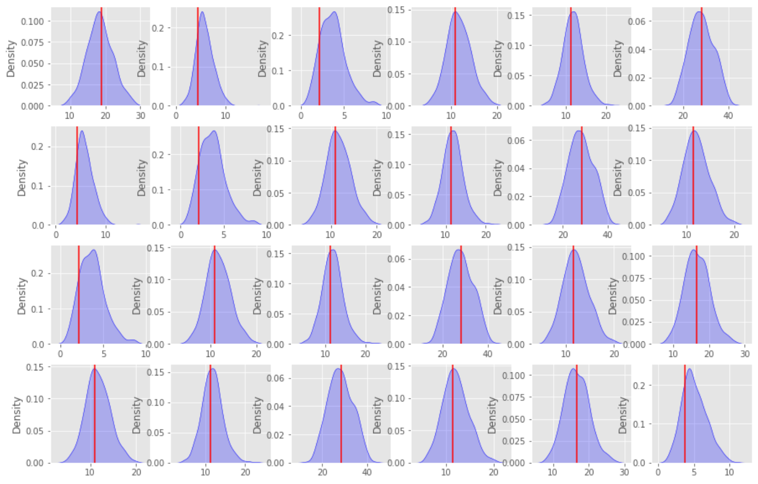


Figure 5.4 – The plot shows the first 20 prediction distributions sampled from the output probabilistic layer. The vertical red line represents the sample prediction media which coincides with the final prediction of the BSTNN model.

Models	MAE	RMSE
INLA	3.868	5.926
LSTM	4.168	6.226
BSTNN with Entity Embeddings	2.731	3.932
BSTNN with LSTM & Entity Embeddings	2.904	3.815
BSTNN with Node2Vec	3.269	4.86
BSTNN with LSTM & Node2Vec	4.019	5.227
BSTNN with VAE Embeddings	3.423	5.292
BSTNN with All Embeddings	**2.653**	3.901
BSTNN with LSTM & Entity Embeddings & Node2Vec	2.878	**3.799**

Table 5.1 – Two different types of scores are used to evaluate the different models (**MAE** and **RMSE**) on the test set (results have been rounded to three decimal places). The lower the score, the better the models perform. In **bold** the best results w.r.t. the score.

5.2 Forecasting Daily-Death Time Series of COVID-19

In January 7, 2020, a deadly new coronavirus strain, SARS-CoV-2, has been identified in China, and since then numerous scientific papers on the COVID-19 disease have been published, addressing global awareness (primarily at national levels) and covering a significant range of disciplines, from medicine to mathematics and social sciences.

5.2.1 Data

The data source considered for the analysis comes from the github repository of the Italian Civil Protection and is available at the following link https://github.com/pcm-dpc/COVID-19. These data are updated daily and exhaustively describe the progress of the pandemic (number of new positives, ICUs (Intensive Care Units) occupied, swabs made, deaths) from a national and regional point of view.
We focus on *New-Daily deceased*. This phenomenon is subject to high variability and to strong shocks, for this reason, capturing the signal and purifying it from the noise requires the use of complex models.
The Fig. 5.5 represents the historical time series of New-Daily Decesead in each Italian region (the plot is difficult to interpret given that the historical series of some regions tend to overlap). The plot below shows the trends of three regions in the 30 days starting from `2022-09-12` to `2022-10-11`, which exhibit different behavior: Lombardy (in blue) has a higher number of daily deaths compared with Emilia Romagna (in orange) and Molise (in violet), while Molise has an extremely low number of deaths.
The following preprocessing steps have been used:

- the daily data of the *Autonomous Province of Bolzano* and the *Autonomous Province of Trento* (reported separately in the reference dataset), were

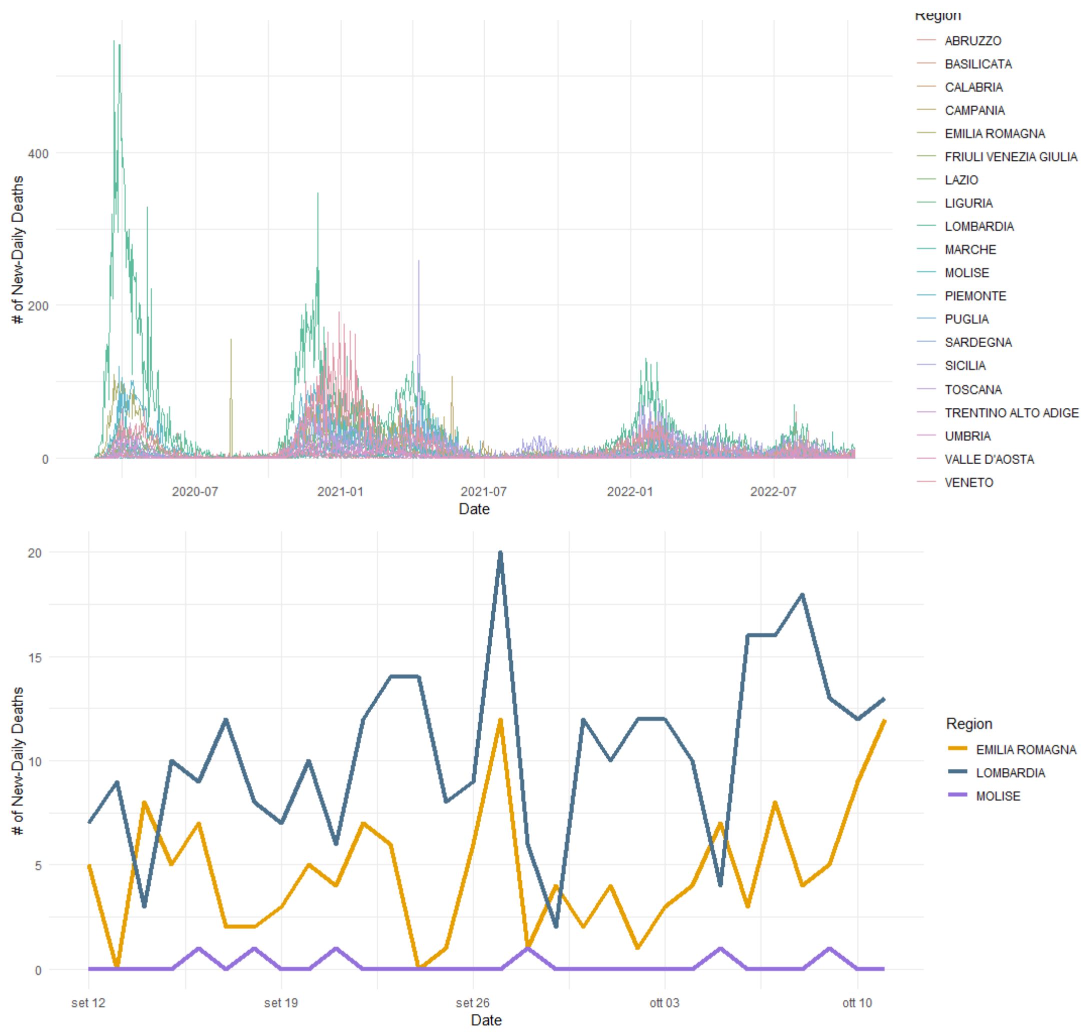


Figure 5.5 – Above, the historical time series of New-Daily Deaths from COVID-19 conditioned on each regions. Below the historical time series conditioned by the Emilia Romagna, Molise and Lombardy Regions starting from `2022-09-12` to `2022-10-11`

aggregated, renaming the entire area as *Trentino Alto Adige*;

- the cumulative sum of daily deaths in the original dataset was differentiated in order to obtain the target variable *"New-Daily Deaths"*;

5.2.2 Results & Comparison with Other State Of The Art Temporal and Spatio-Temporal Models

In this case the prediction task is more complex than the one considered in Subsec. 5.1: the temporal interval considered is much wider (19'200 observations compared to 572 of the previous task) and predictions are daily rather than monthly, therefore more subject to variability and noise (as can be seen from the Fig. 5.5).

Performances are evaluated over a variable forecast range (**7**, **14** and **21** days) and it is analyzed how these changes in forecasting ranges impact on the evaluation metric.
Test set is composed by observations from `2022-09-21` to `2022-10-11`.
For the comparison, INLA, LSTM and the two BSTNN architectures were considered with all the embeddings fed as input (architecture in Fig. 4.2 with no LSTM layers is referred to **BSTNN 1**, while architecture in Fig. 4.3 with LSTM + Attention layers is referred to **BSTNN 2**).
Performances are evaluated using **MAE** due to the fact that results are more interpretable. The models were evaluated both by calculating the MAE on all test observations (Fig. 5.7) and by calculating it at the regional level (Table 5.2 & Fig. 5.6).
Table 5.2 shows that the BSTNN models perform, for most regions, better than INLA and LSTM (the models that perform best for a particular region are highlighted in bold). Observing this comparison between models at a regional level from a graphical point of view in Fig. 5.6, it is clear that BSTNN tends to predict better where INLA and LSTM have excessively high errors, also in this case the Bayesian neural networks seem to capture better than the other two models the uncertainty in the data, the so-called *aleatoric uncertainty.*
Another interesting result can be seen in Fig. 5.7: as the forecast interval increases, the INLA and LSTM forecasts get worse significantly (blue and purple line), while BSTNNs performances, after **21** days, worsen, but much less pronounced (line green and purple). This tells us that, unlike the other two models, BSTNNs lose their predictive power less than other models.

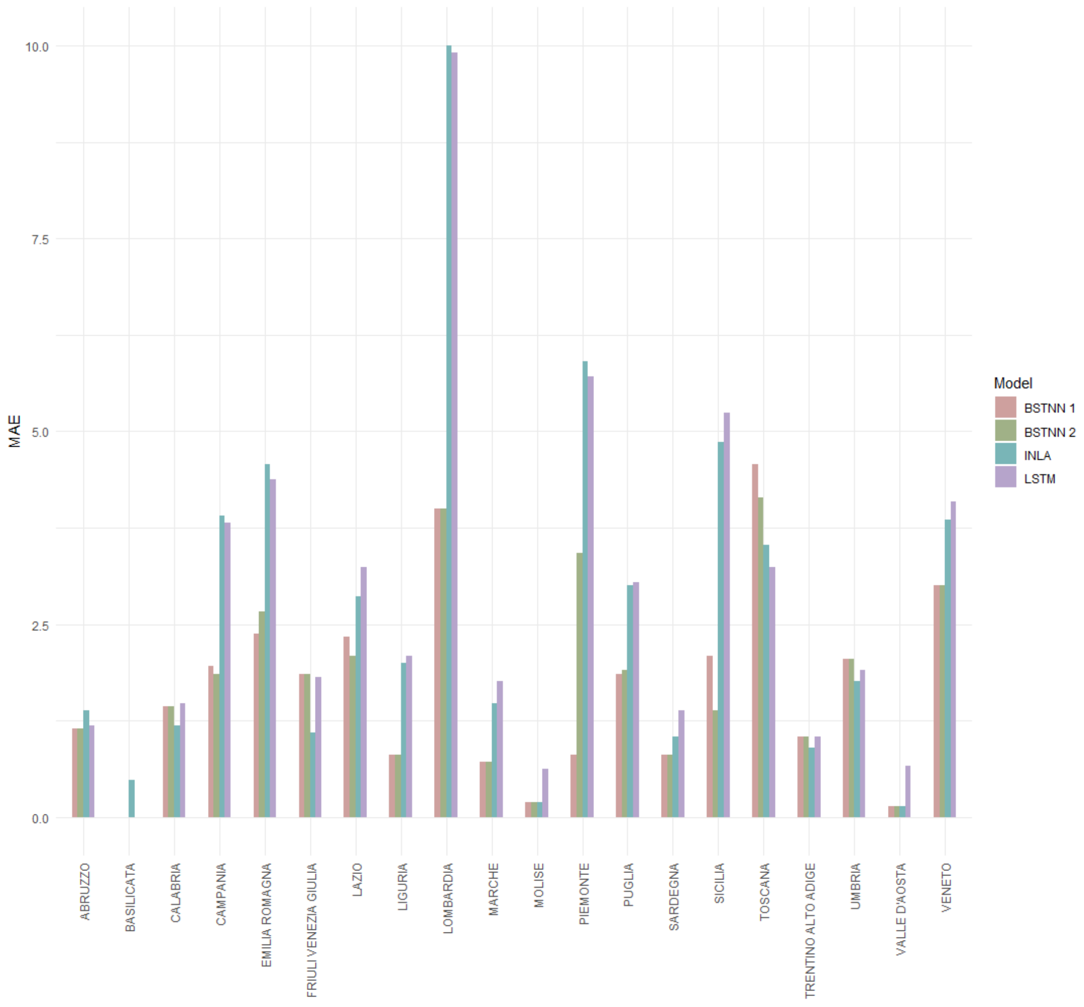


Figure 5.6 – Bars show the MAE scores for every model conditioned on each Region.

Region	BSTNN 1	BSTNN 2	INLA	LSTM
ABRUZZO	**1.14**	**1.14**	1.38	1.19
BASILICATA	**0.00**	**0.00**	0.48	**0.00**
CALABRIA	1.43	1.43	**1.19**	1.48
CAMPANIA	1.95	**1.86**	3.90	3.81
EMILIA ROMAGNA	**2.38**	2.67	4.57	4.38
FRIULI VENEZIA GIULIA	1.86	1.86	**1.10**	1.81
LATIUM	2.33	**2.10**	2.86	3.24
LIGURIA	**0.81**	**0.81**	2.00	2.10
LOMBARDY	**4.00**	**4.00**	10.00	9.90
MARCHES	**0.71**	**0.71**	1.48	1.76
MOLISE	**0.19**	**0.19**	**0.19**	0.62
PEDMONT	**0.81**	3.43	5.90	5.71
APULIA	**1.86**	1.90	3.00	3.05
SARDINIA	**0.81**	**0.81**	1.05	1.38
SICILY	2.10	**1.38**	4.86	5.24
TUSCANY	4.57	4.14	3.52	**3.24**
TRENTINO ALTO ADIGE	1.05	1.05	**0.90**	1.05
UMBRIA	2.05	2.05	**1.76**	1.90
AOSTA VALLEY	**0.14**	**0.14**	**0.14**	0.67
VENETO	**3.00**	**3.00**	3.86	4.10

Table 5.2 – The MAE scores for every model conditioned on each Region. The lower the score the better the model. In **bold** the best model for each Region.

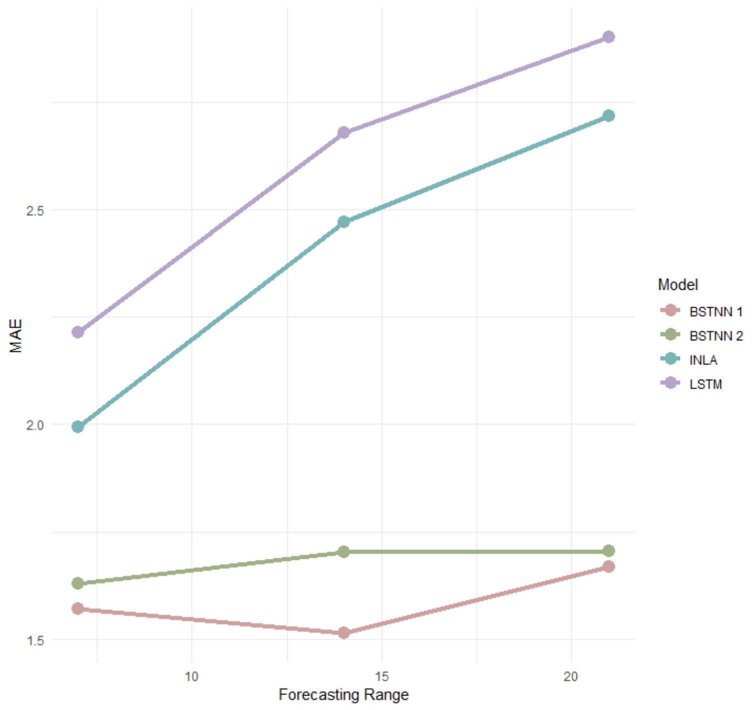


Figure 5.7 – Plot shows how MAE scores change as the forecast range increases (**7**, **14**, and **21** days) w.r.t. each model considered.

Chapter 6

Conclusions & Possible Future Developments

6.1 Goals Reached

This thesis presented a new deep learning architecture called BSTNN that is well suited for forecasting Spatio-Temporal data. Our architecture consists of several components, including Embeddings to model Spatial and Temporal components.

The effectiveness of the proposed architecture is demonstated on two real world datasets (a simple and a complex one) in Chapter 5, the proposed architecture outperforms state-of-the-art Statistics and Machine Learning methods.
Furthermore, the proposed architecture has several advantages over existing approaches namely, i.e. it is able to account for both the aleatoric and epistemic uncertainties, spatial and temporal components without the need to meet any strict statistical assumption.
Thanks to the satisfactory results obtained the desired goals were achieved, however there is no doubt that there is room for improvement both to ameliorate the results, to explore different perspectives, and to set new goals.
Overall, the presented work represents a contribution to the field of probabilistic deep learning.

6.2 Future Developments

While our architecture shows promising results, there are still some limitations to be addressed in future work.
Firstly, since we are dealing with spatio-temporal data, using multivariate input in a time series analysis can be beneficial because it allows to incorporate additional informations about the real data-generating distribution. Implementing a network that receives as input not only a univariate time-series, but a multivari-

ate one, can be particularly advantageous for predictions purposes. This is the case i.e., of the COVID-19 data: the time-series of New Daily-Positives, of the Intensive Care Units (**ICUs**) occupied and of vaccines' data can contribute to the explanation of the target variable New Daily-Deaths.
Another reason why multivariate input can be useful in time series analysis is that it can help to reduce the influence of noise or random fluctuations in the data. By including multiple variables in the analysis, this helps to filter out some of the noise and focus on the more important trends and patterns.
Secondly, the model actual implementation is not particularly efficient: for each instant of time a vector of N values (equal to the number of sites) is fed to the network N times. This means that the network sees the same information repeated N times and discriminates the response only and exclusively with the embeddings. For example, considering the time instant t, $t = 1, \dots, T$, the input that will be fed to the network will be the time sequence associated with each of the N sites and the embedding of site j, $j = 1, \dots, N$ will subsequently be concatenated; the next input will still be the time sequence associated with each of the N sites, and the embedding of site i, $i \neq j$ will subsequently be concatenated. This operation is repeated N times for each temporal instant. A possible solution to this problem could be the one implemented in Fig. 6.1 which shows a new architecture implemented. In the new BSTNN model, the time-series associated with each of the N sites are given as input. This new network shares with the proposed architecture the same layers till flattening, after which N different paths branch off, one for each of the N sites and N different predictions are made. From an intuitive point of view the two architectures implemented in Fig. 4.2 and Fig. 6.1 are equivalent, the second is more efficient both from a computational point of view (it avoids reiterating the same smoothing procedure through the convolutional layers N times) and is *"RAM-friendly"* since it avoids saving the same information N times within the input tensor to the network.
Additionally, A further possible development could be the better characterization of the graph from which the Node2Vec embeddings are extracted. For example, in the case of the geographical embeddings extracted from the COVID-19 dataset, instead of using only the distances between the provinces of each region, it could be useful to better characterize the graph by introducing new weights on the arcs including:

- The travel time using different types of vehicles (trains, planes, buses);
- The number of (travelling) flows from one region to another.

These can be key informations, especially for analyzing an epidemic phenomenon.

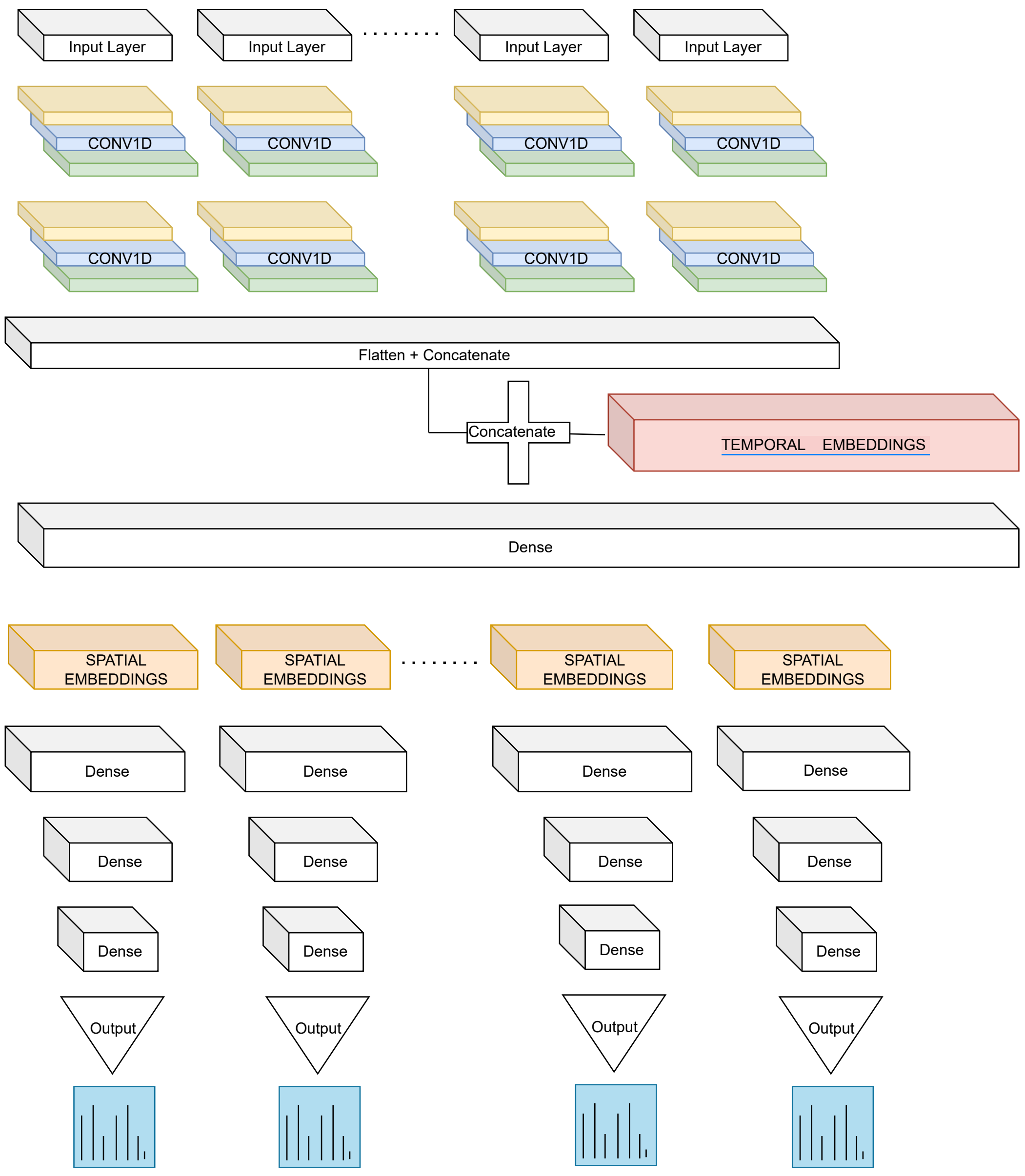


Figure 6.1 – An equivalent, but more efficient, representation of the BSTNN

Chapter 7

Appendix

Derivation of Negative Log-Likelihood Function

In a regression problem, the goal is to model the relationship between a dependent variable y and one or more independent variables x. A common way to evaluate the fit of a regression model is to use the negative log-likelihood function, which is defined as the negative logarithm of the likelihood function. The likelihood function is a measure of the probability of the observed data given the model parameters. In a regression problem, the likelihood function is typically expressed as a Gaussian distribution, where the mean of the distribution is given by the model prediction and the variance is a fixed constant:

$$likelihood = p(y|x, \theta) = N(y|f(x, \theta), \sigma^2)$$

where $N(y|\mu, \sigma^2)$ is the Gaussian distribution with mean μ and variance σ^2, $f(x, \theta)$ is the model prediction, θ is the vector of model parameters, and σ^2 is the variance of the noise. The negative log-likelihood (**NLL**) function is then given by:

$$negativelog - likelihood = -log(likelihood) = -log(N(y|f(x, \theta), \sigma^2))$$

Substituting in the expression for the Gaussian distribution and simplifying gives:

$$\begin{aligned} NLL = &= -log(N(y|f(x, \theta), \sigma^2)) = \\ &= -\frac{(\frac{1}{2}) \times log(2\pi) - (\frac{1}{2}) \times log(\sigma^2) - (\frac{1}{2}) \times (y - f(x, \theta))^2}{\sigma^2} \end{aligned} \tag{7.1}$$

This NLL function is often used as a loss function in a regression problem, with the goal being to minimize the negative log-likelihood by finding the optimal values for the model parameters θ. Note that in this derivation, it is assumed that the noise in the observed data is normally distributed with a fixed variance. In practice, other distributions and noise models may be used, in which case the NLL will have a different form.

How does BackPropagation Scheme Work?

Backpropagation [51] is an algorithm for training artificial neural networks, which are machine learning models inspired by the structure and function of the human brain. The goal of backpropagation is to adjust the weights and biases of the network so that it can accurately predict the output given an input.
The basic idea behind backpropagation is to use the chain rule of calculus to compute the gradient of the loss function with respect to the network weights and biases. The gradient is a vector that indicates the direction in which the weights and biases should be adjusted to minimize the loss.
Here is a brief overview of the steps involved in backpropagation:

1. The input is passed through the network (Forward propagation), and the output is computed by applying the activation function to the weighted sum of the inputs at each neuron.

2. The loss is computed by comparing the predicted output to the true output using a loss function, such as mean squared error.

3. The gradient of the loss function with respect to the output of the network is computed using the chain rule. This gradient is then propagated backwards through the network, and the gradients of the loss function with respect to the weights and biases at each layer are computed (Backward propagation).

4. The weights and biases are updated in the opposite direction of the gradient, using a learning rate hyperparameter to control the size of the update.

5. Steps 1-4 are repeated until the network has converged, or until a predetermined number of epochs has been reached.

Backpropagation is an efficient and effective algorithm for training neural networks, and it has played a crucial role in the success and widespread adoption of deep learning techniques.

Metropolis-Hastings - The Algorithm

The Metropolis-Hastings algorithm [24] is a Markov chain Monte Carlo (**MCMC**) method for sampling from a target distribution. It is used to estimate the parameters of a statistical model or to compute the probability of certain events. The algorithm works by constructing a Markov chain whose stationary distribution is the target distribution. The chain is constructed by generating a sequence of samples from the target distribution, with the property that each sample is dependent on the previous one. The Metropolis-Hastings algorithm consists of the following steps:

1. Initialize the Markov chain by choosing an initial state (also called a "current state") from the target distribution.

2. Generate a proposal for the next state of the chain by sampling from a proposal distribution. The proposal distribution should be chosen such that it is easy to sample from and has a high probability of proposing states that are similar to the current state.

3. Calculate the acceptance probability of the proposed state based on the target distribution and the proposal distribution. The acceptance probability is defined as the ratio of the target distribution evaluated at the proposed state to the target distribution evaluated at the current state, multiplied by the ratio of the proposal distribution evaluated at the current state to the proposal distribution evaluated at the proposed state.

4. Use a uniform random number generator to determine whether to accept the proposed state as the next state of the chain. If the random number is less than the acceptance probability, then the proposed state is accepted. If the random number is greater than the acceptance probability, then the current state is retained as the next state of the chain.

5. Repeat steps 2-4 until the desired number of samples have been obtained.

The Metropolis-Hastings algorithm has several desirable properties, including that it is easy to implement and that it can handle high-dimensional distributions. It is often used in Bayesian analysis and other statistical applications where sampling from the target distribution is difficult.

List of Figures

List of Tables

Bibliography

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in International Conference on Learning Representations, 2015.

[3] E. Hüllermeier and W. Waegeman, "Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods," Machine Learning, vol. 110, no. 3, pp. 457–506, 2021.

[4] Y. Kwon, J.-H. Won, B. J. Kim, and M. C. Paik, "Uncertainty quantification using bayesian neural networks in classification: Application to ischemic stroke lesion segmentation," 2018.

[5] H. Rue, S. Martino, and N. Chopin, "Approximate bayesian inference for latent gaussian models by using integrated nested laplace approximations," Journal of the royal statistical society: Series b (statistical methodology), vol. 71, no. 2, pp. 319–392, 2009.

[6] D. Khana, L. M. Rossen, H. Hedegaard, and M. Warner, "A bayesian spatial and temporal modeling approach to mapping geographic variation in mortality rates for subnational areas with r-inla," Journal of data science: JDS, vol. 16, no. 1, p. 147, 2018.

[7] E. Musenge, T. F. Chirwa, K. Kahn, and P. Vounatsou, "Bayesian analysis of zero inflated spatiotemporal hiv/tb child mortality data through the inla and spde approaches: Applied to data observed between 1992 and 2010 in rural north east south africa," International journal of applied earth observation and geoinformation, vol. 22, pp. 86–98, 2013.

[8] R. K. Pace, R. Barry, J. M. Clapp, and M. Rodriquez, "Spatiotemporal autoregressive models of neighborhood effects," The Journal of Real Estate Finance and Economics, vol. 17, no. 1, pp. 15–33, 1998.

[9] I. Nappi-Choulet Pr and T.-P. Maury, "A spatiotemporal autoregressive price index for the paris office property market," Real Estate Economics, vol. 37, no. 2, pp. 305–340, 2009.

[10] P. Chernyavskiy, M. P. Little, and P. S. Rosenberg, "Spatially varying age–period–cohort analysis with application to us mortality, 2002–2016," Biostatistics, vol. 21, no. 4, pp. 845–859, 2020.

[11] M. Ugarte, T. Goicoa, and A. Militino, "Spatio-temporal modeling of mortality risks using penalized splines," Environmetrics: The Official Journal of the International Environmetrics Society, vol. 21, no. 3-4, pp. 270–289, 2010.

[12] S. Wang, J. Cao, and P. Yu, "Deep learning for spatio-temporal data mining: A survey," IEEE transactions on knowledge and data engineering, 2020.

[13] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," arXiv preprint arXiv:1707.01926, 2017.

[14] N. G. Polson and V. O. Sokolov, "Deep learning for short-term traffic flow prediction," Transportation Research Part C: Emerging Technologies, vol. 79, pp. 1–17, 2017.

[15] Z. Wu, S. Pan, G. Long, J. Jiang, and C. Zhang, "Graph wavenet for deep spatial-temporal graph modeling," arXiv preprint arXiv:1906.00121, 2019.

[16] Z. Wu, S. Pan, G. Long, J. Jiang, and C. Zhang, "Graph wavenet for deep spatial-temporal graph modeling," in Proceedings of the 28th International Joint Conference on Artificial Intelligence, IJCAI'19, p. 1907–1913, AAAI Press, 2019.

[17] A. Kapoor, X. Ben, L. Liu, B. Perozzi, M. Barnes, M. Blais, and S. O'Banion, "Examining covid-19 forecasting using spatio-temporal graph neural networks," arXiv preprint arXiv:2007.03113, 2020.

[18] C. Zheng, X. Fan, C. Wang, and J. Qi, "Gman: A graph multi-attention network for traffic prediction," in Proceedings of the AAAI conference on artificial intelligence, vol. 34, pp. 1234–1241, 2020.

[19] H. Zhang, S. Li, Y. Chen, J. Dai, and Y. Yi, "A novel encoder-decoder model for multivariate time series forecasting," Computational Intelligence and Neuroscience, vol. 2022, 2022.

[20] S. Arlinghaus, Practical handbook of curve fitting. CRC press, 1994.

[21] D. Lambert, "Zero-inflated poisson regression, with an application to defects in manufacturing," Technometrics, vol. 34, no. 1, pp. 1–14, 1992.

[22] A. Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. Driessche, E. Lockhart, L. Cobo, F. Stimberg, et al., "Parallel wavenet: Fast high-fidelity speech synthesis," in International conference on machine learning, pp. 3918–3926, PMLR, 2018.

[23] P. Dellaportas, J. J. Forster, and I. Ntzoufras, "On bayesian model and variable selection using mcmc," Statistics and Computing, vol. 12, no. 1, pp. 27–36, 2002.

[24] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," 1970.

[25] O. Dürr, B. Sick, and E. Murina, Probabilistic Deep Learning: With Python, Keras and TensorFlow Probability. Manning Publications, 2020.

[26] D. T. Chang, "Probabilistic deep learning with probabilistic neural networks and deep probabilistic models," arXiv preprint arXiv:2106.00120, 2021.

[27] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," arXiv preprint arXiv:1312.6114, 2013.

[28] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, "Weight uncertainty in neural network," in International conference on machine learning, pp. 1613–1622, PMLR, 2015.

[29] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in international conference on machine learning, pp. 1050–1059, PMLR, 2016.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," The journal of machine learning research, vol. 15, no. 1, pp. 1929–1958, 2014.

[31] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in Proceedings of the thirtieth annual ACM symposium on Theory of computing, pp. 604–613, 1998.

[32] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," arXiv preprint arXiv:1301.3781, 2013.

[33] X. Rong, "word2vec parameter learning explained," arXiv preprint arXiv:1411.2738, 2014.

[34] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pp. 1532–1543, 2014.

[35] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," Transactions of the association for computational linguistics, vol. 5, pp. 135–146, 2017.

[36] C. Guo and F. Berkhahn, "Entity embeddings of categorical variables," arXiv preprint arXiv:1604.06737, 2016.

[37] Z. Liu, C. Xiong, M. Sun, and Z. Liu, "Explore entity embedding effectiveness in entity retrieval," in China National Conference on Chinese Computational Linguistics, pp. 105–116, Springer, 2019.

[38] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning. MIT press, 2016.

[39] K. Hornik, "Some new results on neural network approximation," Neural networks, vol. 6, no. 8, pp. 1069–1072, 1993.

[40] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," Journal of machine learning research, vol. 9, no. 11, 2008.

[41] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 701–710, 2014.

[42] D. Guthrie, B. Allison, W. Liu, L. Guthrie, and Y. Wilks, "A closer look at skip-gram modelling," in Proceedings of the fifth international conference on language resources and evaluation (LREC'06), 2006.

[43] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 855–864, 2016.

[44] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM journal on computing, vol. 1, no. 2, pp. 146–160, 1972.

[45] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–10, IEEE, 2012.

[46] D. P. Kingma, M. Welling, et al., "An introduction to variational autoencoders," Foundations and Trends® in Machine Learning, vol. 12, no. 4, pp. 307–392, 2019.

[47] Y. LeCun, Y. Bengio, et al., "Convolutional networks for images, speech, and time series," The handbook of brain theory and neural networks, vol. 3361, no. 10, p. 1995, 1995.

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

[49] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," arXiv preprint arXiv:1412.3555, 2014.

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

[51] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” nature, vol. 323, no. 6088, pp. 533–536, 1986.

RINGRAZIAMENTI

Ringrazio, in primo luogo, i miei due relatori.

Ringrazio il Prof. Cesarini che mi ha seguito, negli ultimi due anni, sia nella stesura della tesi triennale sia in quella magistrale. Mi ha sempre aiutato, ogniqualvolta ne avessi bisogno, sempre con immensa disponibilità, cortesia e bontà d'animo che lo contraddistingue come essere umano. Oggi, come due anni fa, lo ringrazio per essermi stato vicino in questo percorso.

Ringrazio il Prof. Peluso con cui ho collaborato nell'ultimo anno ad una serie di progetti che mi hanno fatto avvicinare al suo interessante mondo di ricerca e che sono stati lo spunto fondamentale per l'idea di questa tesi magistrale. Mi ha dato fiducia e coinvolto in tante attività senza mai sovrastarmi con le sue idee e sponsorizzandomi con parole gentili. Gliene sono grato.

Ringrazio i miei amici, quelli di Uni, di Milano, di Monza e di Brughi. Crescere insieme è una delle esperienze più preziose.

Ringrazio Mamma e Papà per avermi accompagnato in questo percorso senza farmi mai mancare il loro appoggio e supporto quando ho chiesto un confronto, gioendo sempre dei miei piccoli successi.

Ringrazio, infine, la mia super sorellina Sofia. La sua bellissima personalità, il suo spirito critico e la sua vivace curiosità sono stati/sono/saranno sempre per me punto di riferimento e ispirazione continua.