Acknowledgments

Questa tesi è il risultato dell'applicazione di competenze e conoscenze frutto del lavoro fatto durante il mio percorso di vita e perfezionte in questi ultimi due anni. Devo ringraziare le persone incontrate lungo questo percorso per lo studente che sono e la persona che sono diventato.
In particolar modo, voglio ringraziare la Professoressa Sonia Petrone che, oltre ad essere un'eccellente Professoressa e Relatrice di tesi, è una persona eccezionale. La ringrazio per la pazienza, il tempo, la disponibilità, l'interesse e l'entusiasmo con cui mi ha seguito durante la stesura di questa tesi, e per non avermi fatto sentire mai a disagio nel porre anche le domande più banali.
Ringrazio il Dott. Grillo, in rappresentanza dell' ISU Bocconi, per aver permesso a me e ai miei fratelli di ricevere il sostegno economico necessario per poter accedere all'insegnamento dell'Università Bocconi.
Ringrazio le donne e gli uomini del fantastico gruppo C.; le amiche e gli amici che mi sono stati accanto durante questi anni di studio a Milano; e le amiche che mi so(u)pportano da una vita: Chiara, Giulia e Cristiana.
Ringrazio Marta per la sua presenza costante, il suo affetto disinteressato, e per essermi stata compagna di vita durante questi anni.
Ringrazio i miei genitori: mia madre per l'amore incondizionato e puro, che è ciò che mi ha reso la persona che sono; mio padre per il supporto costante e per avermi insegnato a vedere sempre il sole oltre la tempesta. Li ringrazio per i sacrifici sopportati per permettere a me e ai miei fratelli di ricevere la migliore istruzione possibile e per essere stati lungimiranti: nonostante nessuno dei due sia laureato, hanno sempre creduto nel valore della cultura e della bellezza, permettendomi di apprendere le competenze necessarie per essere un "uomo in carriera" (cit.) e l'arte della musica per essere una persona migliore.
Infine, ringrazio i miei fratelli, Davide Marco e Andreuccio, e le mie sorelle, Valentina e Roberta, per essere la mia ricchezza, il mio sostegno e la mia compagnia.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others.

Pietro Lesci
February 2019

A mia Madre e a mio Padre

Contents

List of Figures

Chapter 1

Introduction

Artificial Intelligence (AI) is the buzz word of the current times. Most of what is being called “AI”, however, is what has been referred to as “Machine Learning” (ML) for the past several decades. ML is an algorithmic field that blends ideas from statistics, computer science and many other disciplines to design algorithms that make data-driven predictions and decisions.

The phrase “AI” was coined in the late 1950’s to refer to the aspiration of creating an entity possessing human-level intelligence. Nowadays, it is a thriving field of research that encompasses various disciplines such as computer science, engineering, statistics, neuroscience, biology, with applications that range from automating routine labour, interpreting images, understanding speech, to making diagnoses in medicine. Besides the recent hype for AI, the goal of creating artificial agents with human-like capabilities is for the most part cast as function approximation. Deep Neural Nets (DNN), coming in a variety of shapes and tastes, are the preferred tools for function approximation in the field of AI. Two questions promptly arise: what are DNN and why do they work so well?

After few searches on the internet, the answer to the first question would look something like: “Neural networks are computing systems inspired by the biological neural networks that constitute animal brains” or “A neural network is based on a collection of connected artificial neurons, each connection mimics synapses in a biological brain and transmits a signal from one artificial neuron to another which receives the signal and can process it” – paraphrasing Wikipedia *et similia.* Although neural nets are inspired by the human brain, this romanticized definition does not provide many insights on what they *really* are.

To shed light into their essence, we can think of neural networks as a particular kind of basis function in which the basis functions are defined as scalar-valued nonlinear functions applied to an affine transformation of the inputs. The flow of computations can be described as follows:

$$\underset{\text{(input)}}{x} \;\rightarrow\; \underset{\text{(affine transformation)}}{w \bullet +b} \;\rightarrow\; \underset{\text{(element-wise nonlinear transformation)}}{\phi(\bullet)}$$

where with $\bullet$ we refer to the result of the computation performed at the preceding step. A *deep* neural net, is simply a repeated application of the process defined above, that is

$$\underset{\text{(input)}}{x} \;\rightarrow\; \underset{\text{(affine-nonlinear transformation)}}{\phi^{(1)}(w_1 \bullet +b_1)} \;\rightarrow\; \cdots \;\rightarrow\; \underset{\text{(affine-nonlinear transformation)}}{\phi^{(L)}(w_L \bullet +b_L)}$$

where L stands for “layer” and refers to the number of times the affine-plus-nonlinear transformation is applied. Usually, any structure with more than two layers is considered “deep”.

The reason why deep neural nets are so widely used within the field of AI can be traced back to the late 1950’s. In 1957, the Russian mathematician Kolmogorov (Kolmogorov 1957) showed that any continuous function can be represented as a composition of simpler functions. Deep neural nets are an instance of the class of function approximators under the scope of Kolmogorov’s superposition theorem and are at the core of modern deep learning. The deeper the structure, the better the function approximation, the bigger the quantity of data needed, though. Deep neural nets are now the workhorse – in the context of deep learning – to learn complex data structures.

The term *deep learning* is used very broadly to refer to a wide class of ML techniques and architectures with the common trait of using many layers of nonlinear information processing. It has attracted tremendous attention from researchers in various sub-fields within AI, such as computer vision and language processing, but also more traditional sciences such as physics, biology, medicine, and geology. Neural Networks, image processing tools such as Convolutional Neural Networks, and sequence processing models such as Recurrent Neural Networks, are used extensively by practitioners and academics in many fields and are all examples of ML *algorithms* – this word is used, here, in its simplest sense: a sequence of actions.

Defining a chain of computations to solve a problem, by its nature is an ad-hoc procedure: even if many algorithms do work well in many different applications, seldom they can be used out of the box: they do require pretty sophisticated tuning. Indeed, tuning is the core of many research papers and practical applications and despite the fact that a sort of formalism exists, many results are found by trial and errors procedures – e.g. decision regarding the number of layers or which activation function to choose for each layer are still answered on a case-by-case base – or by applying rules of thumb. Usually, there is not a formal derivation from sound first principles. The lack of a general, shared, and coherent framework slows innovation and the spreading of knowledge.

Although, conceptually, Statistics and ML share a lot of commonalities, in the majority of cases the perspective is so antithetic that even the language is very different. A process of convergence between the two disciplines has not been well established yet. Why would such an endeavour be worth the toil? The statistical interpretation of ML algorithms has two important effects: first, it provides the necessary tools to quantify uncertainty; second, it provides a formal justification for otherwise ad-hoc applications. For instance, fields such as medicine (e.g. automated diagnoses) or engineering (e.g. self-driving cars) are ones in which representing model uncertainty is of crucial importance. Furthermore, providing a statistical framework for various, independently proposed, procedures allows for the creation of a *corpus* of knowledge which becomes institutionalized and broadly accessible to researchers outside the ML community, given the role of statistics as the *lingua franca* shared by academics and practitioners in different fields and with various backgrounds. This work is a contribution in this direction.

We would like to provide a fresh and aware *statistical view* of a subset of modern ML algorithms named Deep Learning. Without diminishing their importance, we would like to acknowledge the fact that many of the new "trendy" innovations in this context are a reframing of "classical" statistical problems.

In particular, in Chapter 2 we introduce problems that are discussed in more detail in the following chapters. Starting from the philosophical underpinnings of statistics, we discuss the two main approaches to solving problems that entail randomness: the data-modeling approach and the algorithmic approach. We argue that for each algorithm there exists, perhaps not unique, a statistical model for which the algorithm can be interpreted as performing inference. We focus on a specific family of algorithms: neural networks. The contribution of this chapter is to provide two different possible interpretations of neural networks: as a particular kind of basis function in the context of Bayesian nonparametric regression, and as particular instance of probabilistic graphical models. We conclude the chapter discussing Deep Learning and Deep Neural Networks.

In Chapter 3 we discuss the foundations of Deep Learning from the algorithmic perspective, gradually shifting to a more probabilistic interpretation that is discussed in detail in Chapters 4. We lay down some basic notation and terminology. We discuss optimization and regularization: the core of deep learning models. We present *Bayesian Deep Learning* and, in particular, Bayesian Neural Networks exploring the main preferred tool to perform Bayesian inference in a scalable way: Classical Variational Inference, a hot topic that will be developed in Chapter 4 and which is the bridge between deep learning and Bayesian inference.

In Chapter 4 we analyse very recent extensions of Classical (Mean-Field) Variational Inference. These innovations prompted the development of what is referred to as *Deep Bayesian Learning* which entails the use of deep learning architectures to parametrize probability density functions in the context of Bayesian inference. This differs from *Bayesian Deep Learning* since we are not only interested in making inference about the parameters of a neural network, but we want to use neural networks to facilitate Bayesian inference in complicated models in a computationally efficient way. Two examples are provided at the end of the chapter: Variational Autoencoders and Conditional Variational Autoencoders, two instances of the large class of Deep Latent Variable Models.

In Chapter 5 we present a new hot topic in the field of AI: *Meta-Learning*. Current AI systems excel at mastering a single skill, but when asked to do a variety of seemingly simple related activities they struggle. Meta-Learning aims at endowing AI with the capability of "learning to learn", that is leveraging past experience when learning a new skill. The objective is to reduce the need for training data and allow machines to gain human-like learning capabilities: imagine teaching a robotic arm to grab a new object by showing it how to perform the task only one or "few" times. Such a capability would make it easier for us to communicate new goals to artificial agents – we could simply show them (few times) what we want them to do.

In this chapter we explore the statistical underpinnings of meta-learning and we frame it as a statistical learning problem in hierarchical models via the application of empirical Bayes methods and – when this first solution is not viable – variational inference. Furthermore, we present a specific instance of the class of probabilistic models employed in this area: the *Neural Processes* (NPs) (Garnelo et al. 2018b; Kim et al. 2019). They have been recently (July 2018) introduced as an extension of a "pure" machine learning model called Generative Query Networks (Eslami et al. 2018) – a computer vision system which predicts how a 3D scene looks from any viewpoint after just a few 2D views from other viewpoints. In our view, there is still room for other statistical interpretations, even if, in the original paper, they have already been framed as stochastic process "approximators". By the time of writing, the two original papers (Garnelo et al. 2018a; Garnelo et al. 2018b) have been complemented with minor follow-up unpublished documents that start to explore their peculiarities, such as the various effects of different loss functions and their analogy with Gaussian Processes with deep kernels – a work still in its infancy, released on the internet in January 2019 (unpublished), but surely a promising step in the same direction of this thesis.

Furthermore, the authors have released the Python code which implements *Conditional Neural Processes* (the deterministic counterpart of the fully-fledged Neural Processes) using the `TensorFlow` library. With this thesis we also contribute a Python package, written in `PyTorch` which we called `NeuralProcesses`, that implements NPs. The aim of this package is to be user-friendly and flexible. Up to the best of our knowledge, although there exist two other implementations, this is the first code organized in the form of package that can be distributed as such and is not application-specific. To demonstrate its capabilities we perform two experiments on toy data in Chapter 6. This allows us to test the capabilities of the model in a controlled environment and to show-case the usage of the package.

Chapter 7 concludes.

Chapter 2

Algorithms and Data Models: The Anatomy of Learners

This chapter introduces problems that will be discussed in more detail in the following chapters. Starting from the philosophical underpinnings of statistics, it discusses the two main approaches to solving problems that entail randomness: the data-modeling approach and the algorithmic approach. We argue that for each algorithm there exists, perhaps not unique, a statistical model for which the algorithm can be interpreted as performing inference. We focus on a specific family of algorithms: neural networks. The contribution of this chapter is to provide two different possible interpretations of neural networks: as a particular kind of basis function in the context of Bayesian nonparametric regression, and as particular instance of probabilistic graphical models. We conclude the chapter discussing Deep Learning and Deep Neural Networks.

The desire to create machines capable of thinking accompanies the human kind since the first programmable computer was invented. Nowadays, artificial intelligence (AI) is a thriving field of research that encompasses various disciplines such as computer science, engineering, statistics, neuroscience, biology, with applications that range from automating routine labour, interpreting images, understanding speech, to making diagnoses in medicine.

Problems that are intellectually difficult for humans to tackle, namely those problems that can be described by a list of formal, mathematical rules, are among the easiest for computers to solve. The real challenge to AI is the resolution of tasks relatively easy for people to perform, but hard to describe in formal terms – problems that we, as human beings, solve intuitively, instinctively – such as recognizing objects in images. This instinctive "intuitions" are drawn from experience: collections of small facts (data) and personal judgments of facts (information)[1].

One solution to let machines mimic the process of deduction from experience is to hard-code the knowledge about the reality in formal language, allowing the computer to reason using logical inference rules. This approach, called *knowledge-based* (Goodfellow et al. 2016), is brute-force in nature and not viable in many practical applications.

Therefore, AI systems need to have the ability to "build" their own knowledge by extracting information from raw data. This field of research is known as *machine learning*. However, the performance of this approach crucially depends on the *representation* of data used – the usual maxim "garbage in, garbage out". Each bit of information included in the representation is called *feature*. Selecting or extracting features, that form representations, from raw data is a form of art of its own. Often, as an alternative to hand-designed features, machine learning algorithms are used to learn such *representations* besides learning the *mapping* from representations to outputs. This approach is called **representation leaning** (Goodfellow et al. 2016).

Regardless the methodology used to build such features, the aim is often to separate the *factors of variation* that explain the observed data. In many cases, these factors are not directly observed:

> "[...] They may exist as either unobserved objects or unobserved forces [...] constructs in the human mind [...] concepts or abstractions that help us make sense of the rich variability in the data" – Goodfellow et al. (2016), pp. 4-5

Therefore, extracting representations can be as difficult as solving the original problem. In these cases representation learning comes unhandy. **Deep**

[1]One research project that aims at producing completely automated analyses and reports is the *Automatic Statistician* [`https://automaticstatistician.com`]: the current version of the Automatic Statistician is a system which explores an open-ended space of possible statistical models to discover a good explanation of the data, and then produces a detailed report with figures and natural-language text. The system is based on reasoning over an open-ended collection of nonparametric models using Bayesian inference.

learning (DL) provides a solution by taking a constructionist approach: creating autonomously complex, expressive, representations of the world in terms of simple, more basic, ones. It can be defined as

> "[...] A form of machine learning that uses hierarchical abstract layers of latent variables to perform pattern matching and prediction"
> – Polson and Sokolov (2017), p. 1

In the statistical literature inputs are often called *predictors* or, more classically, *independent variables.* Outputs are usually referred to as *responses* or, more classically, *dependent variables.* Throughout the thesis these terms are used interchangeably. However, differences between statistics and machine learning are not limited to nomenclature. They truly can be defined as "two cultures" (Breiman 2001). In the next section we will discover how a different philosophical approach has been the cause of two, sometimes competing and opposite, visions of the world, that with this work we would like to bring closer.

2.1 The Two Cultures

In one of his most contended contributions (Breiman 2001), Leo Breiman states that

> "There are two cultures in the use of statistical modeling to reach conclusions from data. One assumes that the data are generated by a given *stochastic data model.* The other uses *algorithmic models* and treats the data mechanism as unknown. The statistical community has been committed to the almost exclusive use of data models. This commitment has led to irrelevant theory, questionable conclusions, and has kept statisticians from working on a large range of interesting current problems. Algorithmic modeling, both in theory and practice, has developed rapidly in fields outside statistics. It can be used both on large complex data sets and as a more accurate and informative alternative to data modeling on smaller datasets"

This words are extracted from the incipit of his provocative paper which prompted analogously stringent comments and wider discussions. These debates are a signal that there exist two completely separated fields of research, statistics and machine learning, with a common goal, but with two antithetic perspectives.

The approach we adopt throughout this thesis is that any specific algorithm, as much complicated as it can be, can be framed as inference in a specific statistical model. Therefore, we will argue that for any algorithm there exists a statistical model for which the algorithm performs inference. The goal is, thus, to construct a "story" around our data for which the algorithm is the synthesis. For example, Kulis and Jordan (2012) interpreted the classical clustering methods *k-means* algorithm from a Bayesian nonparametric viewpoint: starting from the asymptotic connection between k-means and mixtures of Gaussians, they show that a Gibbs sampling algorithm for the Dirichlet process mixture approaches a clustering algorithm in the limit so that the resulting algorithm monotonically minimizes the underlying k-means like clustering objective.

Philosophically, one can think of the world as a deterministic chain of causes and effects. Albeit complex, one can believe that there exist, unique, a cause-effect chain that leads to a specific outcome, manifests itself as a fact of our reality. If this is the case, then an all-mighty intellect could, in theory, be able to trace back each effect to its cause. Chance would not exist. Even if we admit that randomness pervades parts of out reality, we would still be bounded to reason about the complementary deterministic part. Our human nature, its finiteness, limits us twice: first, it limits the scope of investigation to the deterministic component of the real; second, it limits our possibilities to reason directly about it. We are bound to observe the deterministic part of our universe in an indirect way that, alone, creates an additional layer of randomness between us and the "truth". Fortunately, statistics provides us with mathematical tools to deal with this latter layer.

Broadly speaking, one of the most important roles of statistics is to try to discover and understand the functioning of our reality – via the study of dependencies, i.e. regression – starting from collections of facts. Facts are described by many source of information, the most important being data. The role of the statistician, generally speaking, is to organize these sources of information and extract insights and knowledge using the principles of statistical analysis. These are regarded as being generated by a black box in which a set of possible causes, interpreted as a vector of input variables x, enter on one side and an effect, a response variables y, comes out on the other: Inside the black box,

$$\begin{tikzcd} x \arrow[r] & \boxed{\text{Nature}} \arrow[r] & y \end{tikzcd}$$

nature associates the predictor variables with the response variables. We get to collect y and x as data, and the goal in analyzing them is twofold:

> *Understanding.* To extract some information about how nature is associating the response variables to the input variables
>
> *Prediction.* To be able to predict what the responses are going to be when future inputs will be fed in

Similarly, there are two approaches towards these goals (Breiman 2001):

- *The Data Modeling Culture.* The analysis starts by assuming a *stochastic data model* for the inside of the black box, e.g. assuming that data are generated by independent draws from $y = f(x, \text{random noise}, \text{parameters})$, and then the values of the parameters are estimated from the data; model validation is performed based on goodness-of-fit tests and residuals analysis

- *The Algorithmic Modeling Culture.* The analysis considers the inside of the box complex and unknown and the purpose is to find a function $f(x)$ – an algorithm, a rule that operates on x to predict the responses y; model validation is performed measuring predictive accuracy

These approaches differ in the logic used. The former starts by trying to impose on nature a set of models and verifies that at least one of them is compatible with the data observed. The latter, on the contrary, is model-agnostic. Its ultimate target is finding *a* rule that *approximates* the mechanism associating inputs and outputs, even if this algorithm is complex, inscrutable and unexpected, without imposing any a priori characteristics on the data-generating process.

Breiman (2001) argues that statisticians in applied research consider data modeling as the main, if not the only, means for statistical analysis:

"I started reading the *Annals of Statistics*, the flagship journal of theoretical statistics, and was bemused. Every article started with: *Assume that the data are generated by the following model: [...]*"

The underlying assumption of the data modeling approach is that the parametric class of models imagined by the statistician for the complex mechanism devised by nature is indeed reasonable. Parameters are then estimated and conclusions are drawn. However, the conclusions drawn are about the model's mechanism, and not about nature's inner functioning. Obviously, if the model is a poor approximation of nature, the conclusions may be wrong. On the other hand, an advantage of data modeling is that it produces a simple and understandable description of the relationship between inputs and responses. This simplicity comes at the cost of uniqueness: different models, although equally valid, may give different descriptions of this relation. The fact that models built using this approach are evaluated mainly using methods, that yield a yes-no answer, creates difficulties in ranking the various models by quality. The alternative proposed by the algorithmic culture is to measure the accuracy of the model's predictions: estimating the parameters in the model using the data and then using the model to predict the data checking how good the prediction is. The extent to which the model resembles nature's functioning is a measure of how well the model can reproduce the natural phenomenon producing the data.

Sometimes approaching problems by looking for an understandable data model imposes an a priori restriction to the ability of statisticians to describe a wide range of statistical problems that cannot be reproduced by using models that remain enough understandable. The algorithmic community rarely make use of data models. In fact, as we discussed above, their approach stems from the fact that the mechanics of how nature produces data are considered partly unknowable. They stop at the mere acceptance of the fact that what is observed is a set of x's that goes in and a set of y's that comes out. The problem is rephrased as finding an algorithm – a sequence of computations – a rule $f(x)$, such that for future observed x, $f(x)$ will be a good predictor of y. Theory, thus, is focused on the properties of algorithms. The implicit assumption made in the theory is that the data is drawn *i.i.d.* from an unknown multivariate distribution. The drawback of this approach is that models are seldom directly interpretable: usually, those that best emulate nature in terms of predictive accuracy are also the most complex, inscrutable, and unstable. Generally speaking, algorithms tend to have low bias, but very high variance.

This two – for the most part – opposite perspectives have not been reconciled yet, even though the possible gains stemming from a unified "statistical science" could be grand. This thesis goes in this direction, starting from the algorithmic approach and trying to extend it with concepts borrowed from the data modeling one.

2.2 Types of Learning

All models, methods, techniques, algorithms, implemented to discover and understand the mechanics of nature are informally considered to perform a kind of learning. This is true for any algorithm. Based on the observed data, the aim is to produce a statistical model of the underlying process from which the data are generated: informally, we have to create a "story" for the data. Once we know the plot of the story, we can fill in the missing bits that the narrator (nature) has avoided telling us: what we call inference. Following Hastie et al. (2009) and Jordan and Bishop (1996), we can distinguish three broad types of statistical modeling problems: *density estimation*, *classification* and *regression*.

Density estimation problems are also referred to as *unsupervised learning* problems in the machine learning literature. In this class of problems, only the predictors are observed, no measurements of the outcome is provided. The learner's task is to describe how the data are organized or clustered, rather than make predictions. The goal is to model the unconditional distribution of data described by some vector X. In the "classical" statistical language, this is referred to as standard multivariate analysis: given k variables, $(X_1, \ldots, X_k)$, we would like to study the properties of their joint distribution, without assigning a special role to any of the k variables.

Classification and **regression** problems are also referred to as *supervised learning* problems in the machine learning literature. In this class of problems we distinguish between *input* variables, which we denote by x, and *output* (or target) variables which we denote by the vector Y. Classification and regression only differ in the nature (discrete or continuous) of the target (output) variable. Using the canonical statistical language, we would distinguish supervised learning from unsupervised learning noting that, in this case, we can still have k variables, $(X_1, \ldots, X_k)$, but we assign a specific meaning to one of them, say the first one, since we may think that it has to be “explained” or “predicted” by the other $k-1$ variables. To highlight this special role, we usually change its name, e.g. X_1 is labeled Y: $(Y, X_2, \ldots, X_k)$.

Regression models and classification models both focus on the conditional density $p(y|x)$. Many regression techniques can be turned into classification methods by applying them to the problem of density estimation for the category probabilities. For example, if there are only two categories, we can denote them by $y \in \{0, 1\}$. The regression problem of modeling $E[y|x]$ can be directly turned into classification noting that $P(y = 1|x) = E[y|x]$, which follows from the definition of y.

Classification and regression problems can also be viewed as special cases of density estimation noting that the most general and complete description of the data is given by the probability distribution function $p(x, y)$ (Jordan and Bishop 1996). However, the usual goal is to be able to make good predictions for the target variables when presented with new values of the inputs. In this case it is convenient to decompose the joint distribution in the form:

$$p(x, y) = p(y|x)\, p(x)$$

and to consider only the conditional distribution $p(y|x)$. Seen in this form, it is easy to assess how density estimation models can often arise more as components of the solution to a more general classification or regression problem – since we may need an estimate of $p(x)$: to be able to estimate $p(x, y)$ or when there are missing data that we need to fill-in in a principled way.

In what follows, we will focus on regression problems only: that is, the estimation of $p(y|x)$.

2.3 The Anatomy of a Learner

An informative high-level definition of learner, or learning algorithm, from a machine learning perspective is provided in Mitchell (1997):

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"

Of course, the variety of experiences, tasks, and performance measures is wide. Below a high-level overview of the possible instances of these concepts is provided:

- Experiences E: supervised or unsupervised learning
- Tasks T: classification, classification with missing inputs, regression, anomaly detection, imputation of missing values, denoising, density estimation
- Performance measures P: accuracy, error rate, goodness-of-fit tests

The starting point is the definition of a function approximator (or model). The machine-learner/statistician chooses an appropriate loss function according to the problem at hand. Once the loss function is chosen, an optimization criterion is defined, e.g. maximum likelihood, and an optimization routine must be set up. Typically the optimization problem is not solved analytically deriving a closed-form formula to provide a symbolic expression for the correct solution. On the contrary, updated estimates of the solutions are found via iterative processes. Optimization refers to the task of either minimizing or maximizing some function $J(\theta)$ by altering θ. Since any maximization problem can be turned into a minimization problem, we will only consider minimization problems. Therefore, we can conclude that all learning algorithms share a common "anatomy":

1. A specification of a dataset
2. A model that defines a parametrised input-output mapping
3. A loss function that depends on the parameters of the model

4. An optimization procedure that finds the parameters values minimizing the loss function

These components are modular: replacing them independently we can obtain any kind of learning algorithm. The cost function typically includes at least one term that causes the learning process to perform statistical estimation. It may also include additional terms used for regularization.

This high-level description could be used for any algorithm as well as any statistical model! In the next section we will show that, in fact, they are two sides of the same coin.

2.4 Algorithms and Statistical models

Given a dataset of pairs $\{(x_i, y_i)\}_{i=1}^N$, the simplest example of learning algorithm that can learn how to predict y based on x is linear regression. The underlying assumption is that the natural mechanics associating inputs and outputs can be described by a linear function mapping, $y = wx$. For each input-output pair, the model is a linear transformation of the inputs. Usually, a modified version of the model is used: an intercept term is added to the equation. This still preserves the linear mapping from parameters to predictions, but the mapping from features to predictions is now an affine transformation

$$y_i = wx_i + b$$

The intercept, in the machine learning literature, is often called the *bias*[2] parameter (of the affine transformation) or *offset vector*. This nomenclature derives from the fact that, in the absence of any input, the prediction is biased towards b.

Different parameters (w, b) define different transformations. The goal is to find those values that minimize the errors in prediction, a procedure often referred to as “fitting” the model to data. To accomplish this task, the predictions are evaluated using a certain loss function. For continuous target variables the mean squared error (MSE) is often used: the values of w and b are chosen in a

[2]This term is not related whatsoever to the idea of *statistical bias*.

way that minimizes the residual sum of squares between the true outcomes and the predicted values, that is

$$\text{MSE}(\theta) \overset{def}{=} J(\theta) = \sum_{i=1}^{N} \|y_i - (wx_i + b)\|_2^2, \quad \theta = (w, b)$$

where $\| \cdot \|_2$ is the L^2-norm (euclidean norm). The function J will represent the *loss* function henceforth, also called *cost* or *objective* function. Minimizing the loss function is the purpose of each machine learning algorithm. In the machine learning literature, even more than in the statistical one, there exists a tight relationship between learning and optimization. Indeed, often they are used as synonyms. Optimizing a function requires being able to compute its derivative with respect to the parameters, a problem we will address in the next chapter.

Let's now build a story around the algorithm. From a statistical point of view, we still assume that the true data-generating process can be described by a linear function mapping. However, in this case, we also assume that we are not able to observe the mechanics of nature directly: what we get to observe is a blurred picture of it. We call this "blurring" ε. We can, thus, rewrite the algorithm above as a statistical model where "statistical" comes from the fact that we are dealing with randomness. We collapse the intercept b and w into $\beta \overset{def}{=} (b, w)$, adding 1 as the first component of the vector of features, $x_i \overset{def}{=} (1, x_i)$. We make explicit an assumption often hidden when defining algorithms: our observations are independent and identically distributed. Thus we write

$$y_i = \beta x_i + \varepsilon_i \qquad \varepsilon \sim WN(0,\ \sigma^2)$$

where WN stands for white noise. If we assume a specific parametric form for WN, e.g. a Gaussian distribution, we can write

$$y_i | x_i \overset{i.i.d.}{\sim} \mathcal{N}\left(\beta x_i,\ \sigma^2\right)$$

In this way we have defined a conditional probability $p(y|x)$. The toolbox of statistical inference contains various methods to estimate the parameter β (and possibly σ^2). If, for example, we use the maximum likelihood approach, we would set the parameter to a value that maximizes the likelihood of our sample: again, we reduce inference (learning) to an optimization problem. In the Bayesian approach we would assign a prior distribution to all uncertain quantities, in this case β, and use the Bayes' theorem to compute the posterior distribution of the parameters given the data.

We have now established a way to connect the two worlds and, strictly speaking, we have shown that what separates them is just the perspective from which we look at the problem. As soon as we assume that a certain degree of randomness exists, then we are in the realm of statistics. We thereby assert that for each algorithm there exists, perhaps not unique, a statistical model (interpretation).

2.5 Bayesian Nonparametric Regression

Generally, the relation between y and x need not be linear. Indeed, in many interesting cases characterized by big and complex data (e.g. speech recognition, image processing) the mapping is better described by a nonlinear function. The regression model can be written in a general way as the combination of a deterministic regression function and a random residual

$$y = \underbrace{f(x)}_{\text{deterministic part}} + \underbrace{\varepsilon}_{\text{random part}}, \qquad \varepsilon \sim p_\varepsilon(\varepsilon)$$

As long as both, the regression function, f, and the residual distribution, p_ε, are indexed by finitely many parameters, e.g. (β, σ^2), inference reduces to a traditional parametric regression problem. The regression problem becomes nonparametric when we relax the parametric assumptions on the model components. We regard as unknowns not the parameters of a function, but the function itself.

Bayesian nonparametrics concerns Bayesian inference methods for *nonparametric models*. A nonparametric model involves at least one infinite-dimensional parameter and hence may also be referred to as "infinite-dimensional model". Examples of infinite-dimensional parameters are functions or measures. The basic idea of nonparametric inference is to use data to infer an unknown quantity while making as few assumptions as possible. Usually, this means using statistical models that are infinite-dimensional. This characterization of nonparametric regression allows for three cases (Muller and Quintana 2004).

Nonparametric Residuals. The model can be generalized by relaxing parametric assumptions on the residual distribution, assuming $\varepsilon|G \overset{i.i.d.}{\sim} G$ with a nonparametric prior distribution $p(G)$ on G, while keeping the regression mean function parametric as $f(\cdot) = f_\theta(\cdot)$ indexed by a (finite-dimensional) parameter vector θ with prior $p(\theta)$. We refer to this case as a *nonparametric error model.* Essentially this becomes density estimation for the residual error. In principle, any model that is used for density estimation could be used, however, to maintain the interpretation of ε as residuals and to avoid identifiability concerns, it is desirable to center the random G at zero, for example, with $E(G) = 0$.

Nonparametric Mean Function. One could, instead, relax parametric assumptions on the mean function and complete the model with a nonparametric prior $f \sim p(f)$. We refer to this case as *nonparametric regression mean function.* Popular choices for $p(f)$ are Gaussian process priors (Rasmussen and Williams 2006) or priors based on basis expansions (Bishop 2006), regression trees (Breiman et al. 1984), wavelet (Vidakovic 2009), splines (Hastie and Tibshirani 1986; Denison 2002) or neural nets (Neal 1996). Such methods can potentially approximate a wide range of regression functions. This approach, however, is limited in the sense that it only allows for flexibility in the mean. Many datasets present non-normality or multi-modality of the errors, skewness, or tail behavior in different regions of the covariate space. To capture such behaviors, a flexible approach for modeling the conditional density that allows both the mean and error distribution to evolve flexibly with the covariates is required.

Fully Nonparametric Regression. One could go nonparametric on both components of the model. We refer to this case as *fully nonparametric regression.* The sampling model becomes $p(y_i|x_i) = G_x$, with a prior on the family of *conditional random probability measures*, $p(G_x, x \in \mathrm{X})$. Many commonly used prior distributions for $\mathcal{G} = \{G_x\}$ are variations of the dependent Dirichlet Process (MacEachern 1999).

Although limited for the reasons we mentioned above, the *nonparametric regression mean function* is a powerful tool, especially with the introduction of deep neural nets – possible thanks to the advent of modern computers and the development of fast numerical methods and software packages, e.g. `PyTorch` and `TensorFlow`. Usually, the residuals are assumed independent and identically normally distributed with mean zero and with common variance. If this is the case, i.e. ε has zero mean, then $f(x) = \mathrm{E}(y|x)$ becomes the *conditional mean function.* When the assumption of a straight line (or a hyperplane) fit is too restrictive we may think to use a richer class of functions. The typical nonparametric regression model is of the form

$$y_i = f(x_i) + \varepsilon_i$$

where $f \in \mathcal{F}$, some class of regression functions. Nonparametric regression models differ in the class of functions to which f is assumed to belong. There exist a variety of different ways to choose $\mathcal{F}$. Two classes of functions are very close to neural networks (which we will discuss in the next section): *local methods* and *basis functions.*

Local methods. They define one of the simplest classes of functions used in nonparametric regression problems. As the name suggests, they operate locally. One example of a member of this class, in one dimension, is a moving average with a fixed window size that results in a step function. Sophistication can be added in the choice of the window, the weighting of the averaging, or the shape of the fit over the window. One example of such sophistication is *kernel smoothing*: the idea is to use a moving weighted average, where the weight function is referred to as a kernel. A kernel is a continuous, bounded, symmetric function whose integral is one. Whatever the choice of the kernel, the resulting regression function estimate at a value x of a single explanatory variable is

$$\hat{y} = \frac{\sum_{i=1}^{n} y_i K(x - x_i)}{\sum_{i=1}^{n} K(x - x_i)}$$

The kernel regression approach is related to works on kernel density estimation (Silverman 1986; Loader 2006).

A completely different approach is to abandon the moving window in favour of partitioning the space into an exhaustive set of mutually exclusive regions. Using a constant function over the region gives rise to a step function. In one dimension, this is a histogram estimator (Gentle 2002, section 9.2). In higher dimensions, a tree is a useful method for representing the splitting of the space into regions (Breiman et al. 1984). For regression, the predicted value is the average value of the response variable in a region, and for classification the fitted probability of class membership is the empirical probability in that region.

Basis functions. A large class of nonparametric methods are those that use an infinite set of basis functions to span the space of interest, e.g. $\mathcal{F}$ – typically either the space of continuous functions or the space of square-integrable functions (Rasmussen and Williams (2006)). The regression model becomes of the form

$$y_i = \sum_{k=0}^{\infty} \beta_k \phi_k(x_i) + \varepsilon_i$$

where $\phi_1, \phi_2, \ldots$ is a set of basis functions and, usually, ϕ_0 is defined equal to 1 everywhere. An infinite number of terms would typically be required for a theoretical match to a continuous function, but in practice a finite number of terms is used to provide a close approximation to the infinite sum. We can obtain this imposing $k = 1, \ldots, K$ and then forming a linear combination of these functions, so that for a sufficiently large value of K, and for a suitable choice of the $\phi_k(x)$, such a model has the desired "universal approximation" properties (Kolmogorov 1957). Models of this form can be expressed as network diagrams in which there is a single layer of adaptive weights, also called **shallow learners** in the machine learning literature (Polson and Sokolov 2017).

An example of a basis set that spans the space of continuous functions is the standard basis of the vector space of all polynomials, $\{1, x, x^2, x^3, x^4, \ldots\}$. Any continuous function can be approximated arbitrarily closely with a linear combination of these functions. Also mixture models can be basis representations, depending on the particular form of the model (Jordan and Bishop 1996). One common form of a mixture model is the mixture of Gaussians: a collection of Gaussian densities is used as a basis set. For a univariate problem the model is

$$y_i = \sum_{k=0}^{\infty} \beta_k \ \phi\left(\frac{x_i - \mu_k}{\sigma_k}\right) + \varepsilon_i$$

where $\phi(u) = \frac{1}{\sqrt{2\pi}} \exp\{-u^2/2\}$ is the standard Gaussian density function, and

μ_k and σ_k are, respectively, the mean and the standard deviation of the Gaussian distributions in the mixture.

2.6 Neural Networks

We can now, naturally, introduce neural networks in the context of nonparametric regression. Neural networks are a particular case of basis function in which the basis functions are defined as $\phi_k^{w,b}(x) = \varphi(w_k x + b_k)$, where the scalar-valued nonlinear function, φ, is applied to the affine transformation of the inputs. The nonlinearity, also called *activation function*, is defined to be identical for each basis function. The most common activation functions used are presented in fig. 2.1. In the machine learning literature the basis functions are called *hidden nodes*. The overall number, K, of hidden nodes determines the *width* of the model. The collection of all hidden nodes is referred to as *hidden layer*. Usu-

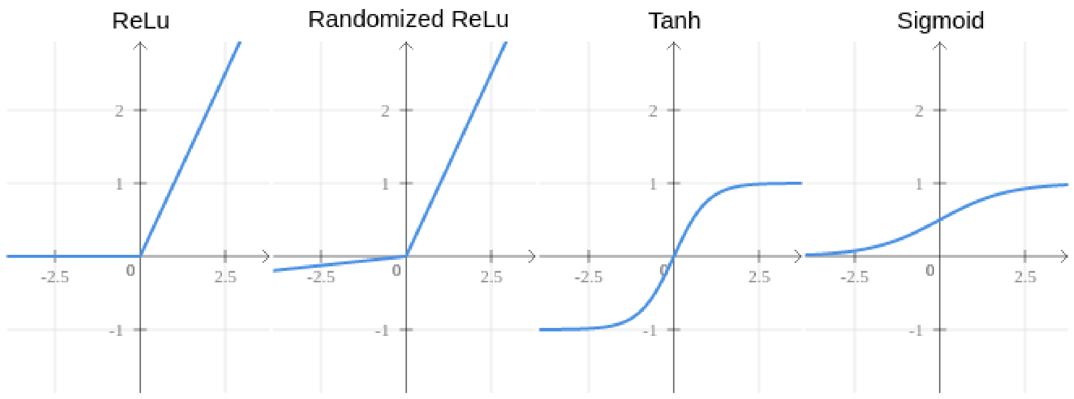

Figure 2.1: *Most used activation functions.*

ally, layers are not interpreted as vector-to-vector functions. Rather, they are thought of as consisting of computational units or neurons, that act in parallel, each representing a vector-to-scalar function. This view highlights the inspiration of neural networks to the human brain. The univariate regression model can be written as

$$y_i = \sum_{k=0}^{K} \beta_k \, \phi_k^{w,b}(x) + \varepsilon_i$$

This is the canonical example of a neural network, known also as fully-connected

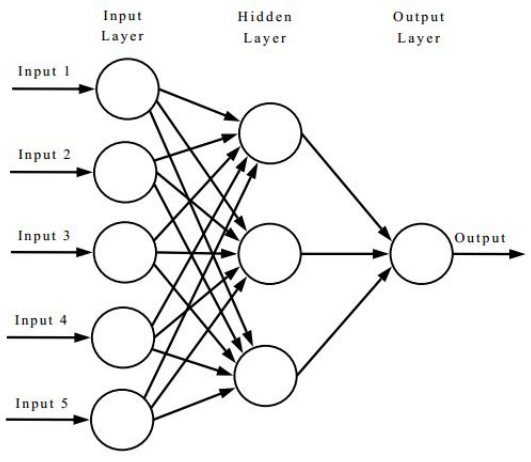

Figure 2.2: *A stylized example of a fully-connected feedforward neural network. The input layer has four units, corresponding to the components of the vector $x_i \in \mathbb{R}^4,\ i = 1, \dots, N$. Each circle is a neuron, or hidden node, and calculates a weighted sum of all incoming inputs and adds a bias, and then applies a nonlinear function element-wise to produce an output, which is averaged with the outputs of the other hidden nodes to produce the output $y_i,\ i = 1, \dots, N$.*

feedforward neural network (Rosenblatt 1958; Goodfellow et al. 2016). For a historical review of neural networks consult Schmidhuber (2015).

The neural network written above can be interpreted as a two-stage regression model: in the first stage the features are derived via a composition of an affine transformation of the inputs and a component-wise nonlinear function. These extracted features, in the second stage, are mapped to the output via a second affine transformation (Hastie et al. 2009). In fact, also linear regression can be interpreted as a very simple example of neural network where φ is an identity mapping. These models are associated with directed acyclic graphs, known as *network diagram.* The network diagram of the model described above is presented in fig. 2.2. To extend the above model to a multivariate y, we simply treat each dimension of y as a separate output and add a set of connections from each hidden node to each of the dimensions of y. We can also assign an observation-specific residual variance (Lee 2004).

It is clear from the equation above that a neural network is simply a nonparametric regression using a basis representation. In other words, the key to understanding a neural network model is to think of it in terms of basis functions.

From the equation above, it is also clear that a neural network is a standard parametric model, with a likelihood and parameters to be fit. We can, thus, claim that it is not a black box or purely an algorithm. It should now be apparent that neural nets are both algorithms *and* statistical models. By viewing neural networks as statistical models, we can apply many other ideas in statistics in order to understand, improve, and appropriately use these models (Lee 2004; Polson and Sokolov 2017).

The neural network described above is an example of a shallow learner: informally, every model that cannot be categorized as "deep" – which we discuss in the next section – is a shallow learner. Almost all shallow learners are data reduction techniques that consist of a low dimensional auxiliary variable Z and a prediction rule specified by a composition of functions

$$f(x) = f_2\big(f_1(x)\big) = f_2(z), \qquad f_1(x) := z$$

Linear regression, Logistic regression, Principal component analysis, Partial

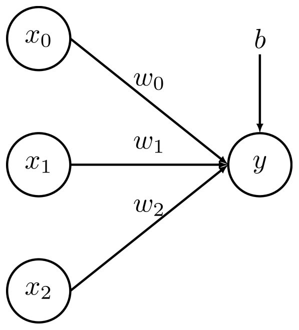

Figure 2.3: *The network graph of linear regression. For each observation, the predictor is a 3-D vector. The addition of the intercept has been made explicit.*

least squares, Reduced rank regression, Linear discriminant analysis and Project pursuit regression are all examples of shallow learners. Furthermore, they all can be represented as a neural network, for example a linear regression network diagram is shown in fig. 2.3.

2.7 Neural Nets as Graphical Models

How can we represent conditional and unconditional distributions with neural networks? One way to answer this question is to find a suitable graph representation of probability distributions. Probabilistic Graphical Models (PGM) use diagrammatic representations to describe random variables and relationships among them. Similar to a graph that contains nodes (vertices) and links (edges), PGM has nodes to represent random variables and links to express probabilistic relationships among them. Below we will provide a simple example.

Consider a case in which we want to model the unconditional distribution $p(x)$. A general approach to density estimation is to treat the density as being composed of a set of K simpler densities, where possibly $K = \infty$. This approach is a probabilistic form of clustering which involves modeling the observed data as a sample from a *mixture density*:

$$p(x|\pi) = \sum_{i=1}^{K} \pi_i \, p(x|i, \theta_i)$$

where the π_i are constants known as *mixing proportions*, the $p(x|i, \theta_i)$ are the *component densities*, generally taken to be from a simple parametric family, and θ_i are the parameters of the component densities. A common choice for component density is the multivariate Gaussian. In this case $\theta_i = (\mu_i, \Sigma_i)$. By varying this parameter, a wide variety of high-dimensional, multi-modal data can be modeled. Gaussian mixtures are representable as simple networks with one or more layers of adaptive weights, e.g. principal component analysis, kernel density estimation and factor analysis (Polson and Sokolov 2017), as shown in fig. 2.4.

Neural networks express relationships between variables by utilizing the representational language of graph theory. Variables are associated with nodes in a graph and transformations of variables are based on algorithms that "[...] propagate numerical messages along the links of the graph" (Jordan and Bishop 1996). The graphs is accompanied by probabilistic interpretations of the variables and their interrelationships. As we have seen, such probabilistic interpretations allow a neural network to be understood as a form of probabilistic model and

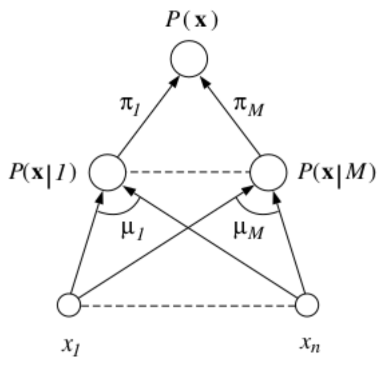

Figure 2.4: *A network representation of a Gaussian mixture distribution. The input nodes representing the components of x are in the lower level. Each link has a weight μ_{ij}, which is the j-th component of the mean vector for the i-th Gaussian. Each node in the intermediate layer contains the covariance matrix Σ_i and computes the Gaussian conditional probability $p(x|i, \mu_i, \Sigma_i)$. These probabilities are weighted by the mixing proportions π_i in the last layer and the output node calculates the weighted sum $p(x) = \sum_i \pi_i p(x|i, w_i)$.*

reduce the problem of learning the weights of a network to a problem in statistics. Hidden Markov models and Dynamic Linear models – also called Kalman filters in the machine learning literature – are all examples of graphical probabilistic models. There is a strong relationship between these models and neural networks: it is often possible to reduce one kind of model to the other. Indeed, neural networks can be seen as members of a general family of probabilistic graphical models (Goodfellow et al. 2016, ch. 20).

Although elegant, this interpretation of neural nets is limited since interpretability becomes easily an issue when the network grows deeper. That is why this interpretation will not be treated again in what follows. Again, a more useful interpretation, is to look at neural nets as a highly nonlinear nonparametric function approximator, that is as a particular case of basis function.

2.8 Deep Learners

Deep learning refers to a wide class of machine learning techniques and architectures, with the common trait of "[...] using many layers of nonlinear information processing" (Deng and Yu 2014). In general, a deep learner, is a representation learning methods with multiple levels of representation, obtained by composing simple but nonlinear transformations of the raw data, with each of these transforming the representation at one level (starting with the raw input) into a representation at a higher level of abstraction. Composing enough of such transformations, very complex functions can be learned (LeCun et al. 2015). A deep architecture can be thought as a multilayer stack of simple models.

A **Deep Neural Net** (DNN) simply applies a composition of functions ϕ, instead of a single one. Deep neural nets provide a flexible tool capable of rendering nonlinear mappings which can approximate any given function to arbitrary accuracy. Given an input, $X \in \mathbb{R}^Q$, and an output, $Y \in \mathbb{R}^D$, usually both high-dimensional, in a plain neural network the mapping $f : \mathbb{R}^Q \to \mathbb{R}^D$ is modeled via the superposition of univariate semi-affine functions where univariate activation functions are used to decompose the high-dimensional X. The particular characteristic of DNN is the depth of the networks, that is the number of hidden layers. Then, a deep prediction rule can be expressed, in matrix notation, as:

$$\begin{aligned}
h^{(1)} &= \varphi_1\left(W^{(0)}X + b_0\right)\\
h^{(2)} &= \varphi_2\left(W^{(1)}h^{(1)} + b_1\right)\\
&\dots\\
h^{(L)} &= \varphi_L\left(W^{(L-1)}h^{(L-1)} + b_{L-1}\right)\\
f(X) &\approx W^{(L)}h^{(L)} + b_L
\end{aligned}$$

where $h^{(l)}$ identifies the layer. DNN are, perhaps, the most famous instance of deep learning models. However, recently, other models characterized by deep architectures have been developed: e.g. deep gaussian processes (Damianou and Lawrence 2013; Lee et al. 2018) and neural processes (Garnelo et al. 2018b) – analysed in Chapter 5.

How a series of simple operations can represent complicated mappings? In other words, why this approach works? The formal roots of DL can be traced back in Kolmogorov's representation of a multivariate response surface as a superposition of univariate activation functions applied to an affine transformation of the input variables (Kolmogorov 1957). In 1957 the Russian mathematician

Kolmogorov showed that *any* continuous function f of many variables can be represented as a composition of simpler functions of one variable. The original version of this theorem can be expressed as follows[3]:

> **Theorem (Kolmogorov-Arnold superposition theorem).** *Let $f : [0,1]^d \to \mathbb{R}$ be an arbitrary multivariate continuous function defined on the identity hypercube. Then it has the representation*
>
> $$f(\mathbf{x}) = f(x_1, \ldots, x_d) = \sum_{q=0}^{2d} \phi_q \left(\sum_{p=1}^{d} \psi_{q,p}(x_p) \right)$$
>
> *with continuous one–dimensional inner and outer functions ϕ_q and $\psi_{q,p}$, defined on the real line. The inner functions $\psi_{q,p}$ are independent of the function f.*

Starting from the 2000s, Kolmogorov's superposition theorem found the attention of the machine learning community interest in providing a theoretical justification of neural networks. Leni et al. (2011) try to apply the theorem to a feed-forward network with an input layer, one hidden layer and an output layer (Hecht-Nielsen 1987). However, the inner functions, ψ, in this application of the theorem are highly nonsmooth and thus not useful in optimization. Sprecher (1965) contributed to this research topic providing an explicit method to construct the univariate functions ψ, corresponding to the activation functions of the network. Bryant (2008) implements Sprecher (1972) algorithm to estimate the inner link function. In his application of the theorem, deep layers allow for smooth activation functions to provide "learned" hyperplanes which find the underlying complex interactions and regions without having to see an exponentially large number of training samples.

To summarize, rather than manually engineering the transformations to be applied to the input data to create more abstract concepts, deep learning architectures provide a way to learn them: the behaviour of the hidden layers is not directly specified in the data, it is learned.

[3]We propose a version of the theorem deducted from Braun and Griebel (2009) and Leni et al. (2011).

Chapter 3

The Learning Process

In this chapter we discuss the foundations of deep learning from the algorithmic perspective, gradually shifting to a more probabilistic interpretation that will be discussed in detail in the next chapters. We lay down the basic notation and terminology. We discuss optimization and regularization: the core of deep learning models. We present Bayesian Deep Learning and, in particular, Bayesian Neural Networks exploring the main preferred tool to perform Bayesian inference in a scalable way: Classical Variational Inference, a hot topic that will be developed in the next chapter and which is the bridge between deep learning and Bayesian inference. In brief, Bayesian deep learning consists in interpreting optimization and/or regularization procedures as performing some kind of variational approximations; two examples are provided at the end of the chapter.

The goal of the learning process of any learning algorithm is to minimize the expected generalization error, known as **risk**, i.e. to maximize its accuracy when new inputs are provided. This is achieved via two paths: **optimization**, which is the process of finding parameters values that minimize the loss function, and **regularization** which is the process of reducing model capacity avoiding overfitting on the observed data and improve generalization performances. To use Mitchell's codification (Mitchell 1997), we care about some performance measure P defined with respect to new unseen observations.

To measure the generalization performance of the model we would compute the loss function, e.g. the MSE, of the model with respect to unseen data. For this purpose, often in practical applications the dataset is divided into a *training* *set*, used to perform the learning process, and a *test set*, used to measure out-of-sample performances. Sometimes, especially for DL, a third partition is created called *validation set* and is used to learn the hyperparameters of the model, if not calibrated ex-ante. In what follows, if not explicitly declared otherwise, N will refer to the number of observations in the training set while N^{test} and N^{val} will refer, respectively, to the dimension of the test and validation set.

To improve generalization performances means to reduce the loss on the test set, J_{test}. However, only the training data are used to train the model. One can intuitively minimize the loss, J_{train} on the training set, simply solving for where its gradient is $\nabla_\theta J_{train} = 0$, and hope this will lead to improvements on the test set prediction performance. What we actually care about is the generalization error, computed on the unobserved data. How can we affect the performance on the test set when we can observe only the training set? Statistical theory comes handy in justifying this approach. We are able to do so thanks to the implicit assumption (§2.1) that the training and test set are produced by the same data-generating process and each observation is independent of any other.

Now that we have justified why it is sensible to minimize the loss function on the observed data, how do we actually minimize it? In other words, how do we choose the value for the parameter θ which minimizes our loss function producing "good" estimates $\hat{y}(x)$?

In the machine learning literature, learning (or inference) is always cast as an optimization problem. In the next section we will give an overview of the procedure.

3.1 Optimization

The aim of optimization is to find values for the parameter θ in the parameter space Θ that minimize the loss function $J(\theta)$, which usually includes a term specifying a performance measure evaluated on the training set and a regularization term to improve the generalization error and avoid overfitting.

Note that for the purpose of this thesis, we will treat optimization in a supervised learning context, i.e. we are provided with the outcome variable Y as well as the inputs X. Furthermore, throughout this section, we will refer to the unregularized optimization case, i.e. the loss function will not contain any regularization term. The development of the regularized cases will be addressed in the next section.

It is common in machine learning applications to use loss functions that decompose as a sum over per-observation loss functions. Therefore, up to a multiplicative constant, the cost function can be seen as an expectation (Goodfellow et al. 2016). In principle, we would like to minimize the objective function where the expectation is taken over the data-generating distribution, p_{data}:

$$J^{\star}(\theta) = \mathrm{E}_{(X,Y)\sim p_{data}} L\big(f_\theta(x),\ y\big) \tag{3.1}$$

that defines the risk, i.e. the expected generalization error, while $f_\theta(x)$ defines the predicted output when the input is x.

However, in practise we only have a finite training set. Therefore, we are bound to use a reduced form of $J^{\star}(\theta)$, namely the **empirical risk** defined as

$$J(\theta) = \mathrm{E}_{(X,Y)\sim \hat{p}_{data}} L\big(f_\theta(x),\ y\big) = \frac{1}{N}\sum_{i=1}^{N} L\big(f_\theta(x_i),\ y_i\big) \tag{3.2}$$

where $\hat{p}_{data}$ is the empirical distribution on the training set. Therefore, rather than optimizing the risk directly, the empirical risk is minimized with the hope that the risk decreases as well – a procedure called empirical risk minimization.

The most important mathematical tool used for optimization is the derivative operator, denoted as $J'(\theta)$ or $dJ/d\theta$. The concept of derivative is crucial for optimization algorithms because it specifies how to scale a small change in the parameters θ to obtain the corresponding change in the loss function evaluation

$$J(\theta + \lambda) \approx J(\theta) + \lambda J'(\theta)$$

It tells us that $J(\theta)$ can be reduced by moving θ in small steps with the opposite sign of the derivative. The type of optimization techniques employing the concept of derivative as a tool is called **gradient-based** optimization. In machine learning this is the most used class of optimization methods and will be the object of this section.

How does gradient-based optimization work? Since, as said, often loss functions decompose as a sum over some per-example loss functions, the computation of the gradient is performed using the linearity of the derivative operator

$$\begin{aligned} g = \nabla_\theta J(\theta) &= \nabla_\theta \mathrm{E}_{(X,Y)\sim\hat{p}_{data}} L\big(f_\theta(x),\ y\big) \\ &= \nabla_\theta \frac{1}{N}\sum_{i=1}^{N} L\big(f_\theta(x_i),\ y_i\big) \\ &= \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta L\big(f_\theta(x_i),\ y_i\big) \end{aligned} \tag{3.3}$$

The most basic example of gradient-based optimization algorithm is **gradient descent** (Cauchy 1847; Bishop 2006). It proposes a new parameter value according to the updating rule

$$\theta^{new} \leftarrow \theta - \lambda g \tag{3.4}$$

where $\lambda \in \mathbb{R}$ is an hyperparameter called the **learning rate**, a positive scalar determining how the gradient affects the proposed value[1].

One issue in applying gradient descent is that, when the N is big, computing the expectation in eq. (3.3) is computationally very expensive since before an update is proposed, the model must be evaluated on the entire dataset in order to compute the per-example loss and calculate the average. Optimization methods that use the entire training set to compute the gradient are called **batch** gradient methods. One way to solve the issue is to resort to statistical estimation. Instead of computing the exact gradient, it is approximated by randomly sampling a small number of examples from the training set, computing the per-example loss, and taking the average over them only. This type of optimization methods is called **minibatch** or **stochastic** gradient methods. Of course, to compute an unbiased estimate of the expected gradient, minibatches must be selected truly randomly. It can be shown[2] that when the stochastic gradient is computed on different observations – i.e. no observation is resampled – the approximate expected gradient follows the gradient of the true generalization error, eq. (3.1). Each time the training set is fully sampled, the procedure

[1]Its determination has posed serious challenges to both practitioners and researchers. In subsection (§3.1.1) we will review ways to set the parameter. Usually for "vanilla" gradient descent it is kept fixed during the entire optimization process.

[2]Goodfellow et al. (2016) pp. 273-274.

restarts again. Each of this iterations is usually called **epoch**.

The basic example of a stochastic optimization (Ruder 2016) method is **stochastic gradient descent**. The update rule for the parameters is the same as in eq. (3.4) but instead of an exact g, an approximate $\hat{g}$ is used. The learning rate usually is not held fixed, but it is gradually decreased over time. This is necessary since the estimation of g introduces noise, via the random sampling, that does not vanish even when a minimum is reached. Adaptive learning rate methods are discussed below. On the contrary, batch optimization methods, such as gradient descent, can hold λ fixed over time[3].

Another example of stochastic optimization is the **momentum** method (Qian 1999). This algorithm adds a variable v, called *velocity*, that stores the value of an exponentially decaying moving average of the past gradients and suggests the algorithm to move in the same direction. A memory parameter $\alpha \in [0, 1)$ determines how much past information to store. The update rule becomes, denoting the size of the minibatch with m,

$$\begin{aligned} v^{new} &\leftarrow \alpha v - \lambda \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\big(f_\theta(x_i),\ y_i\big) \right), \quad m < N \\ \theta^{new} &\leftarrow \theta + v^{new} \end{aligned}$$

Given λ – that should be modeled to vary in some ways – the size of the step now depends on how large and aligned the sequence of gradients is.

A slightly modified version of the momentum algorithm is the **Nesterov momentum** (Sutskever et al. 2013). The gradient is evaluated after the velocity is applied. The update rule in this case is

$$\begin{aligned} v^{new} &\leftarrow \alpha v - \lambda \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\big(f_{\theta+\alpha v}(x_i),\ y_i\big) \right) \\ \theta^{new} &\leftarrow \theta + v^{new} \end{aligned}$$

It is important to note that given the common non-convexity of loss functions in deep learning, an important factor impacting the convergence of optimization algorithms is parameter initialization, i.e. the starting point of the path towards

[3]For more details on how to tackle the issue, the reader is redirected to Goodfellow et al. (2016) ch. 8.

the minimum. This is the reason why, sometimes optimization is brought about using a completely different approach: training a simpler model and use the trained parameters as initial values of the more complex model. This technique is called *supervised pretraining* (Goodfellow et al. 2016).

3.1.1 The Learning Rate Problem

The convergence speed of SGD depends on the variance of the gradient estimates which, in turn, depends on the size of the mini-batch: by the law of large numbers, increasing the mini-batch size reduces the stochastic gradient noise. Smaller gradient noise allows for larger learning rates and leads to faster convergence. For batch gradient descent where we use the entire dataset to evaluatethe loss function, the learning rate can be relatively bigger with respect to using stochastic gradient descent. Therefore, there is a trade-off between the computational overhead associated with processing a mini-batch of a bigger size, and the computational cost of performing more gradient steps due to the smaller learning rate.

To accelerate the learning procedure, one can either optimally adapt the mini-batch size for a given learning rate, or optimally adjust the learning rate to a fixed mini-batch size. The latter approach is the preferred in the machine learning literature.

How does the learning parameter need to be chosen? This question has multiple answers, each corresponding to a different algorithm. Here we briefly review the most important that will be useful in actual applications. The basic idea is that in each iteration, the empirical gradient variance can guide the adaptation of the learning rate which is inversely proportional to the gradient noise. Popular optimization methods that make use of this idea include

Delta-bar-delta. The delta-bar-delta (Jacobs 2008) algorithm was one of the first heuristic method to adapt the learning rate of batch optimization methods. The rule is simple: if the partial derivative of the loss with respect to a given model parameter remains the same sign, then the learning rate should increase, otherwise the learning rate should decrease.

AdaGrad. A similar heuristic approach is AdaGrad (Duchi et al. 2011) that scales the individual model parameters inversely proportional to the square root of the sum of all the historical squared values of the gradient.

RMSProp. A modification of AdaGrad is provided by the RMSProp algorithm (Hinton 2012). It changes the gradient accumulation into an exponentially weighted moving average controlled by an hyperparameter ρ determining the algorithm's memory.

Adam. The Adam algorithm combines RMSProp to set the learning rate with the momentum method (Kingma and Ba 2015).

This methods are referred to as *adaptive subgradient methods* and have been recently given a statistical interpretation (Salas et al. 2018) in the context of Bayesian Deep Learning.

3.2 Regularization

The aim of a learning algorithm is twofold: minimizing the training error and the the gap between training and test error. In other words, avoiding

- **Underfitting**: high error rate on the training set
- **Overfitting**: large gap between training error and generalization error

We can control whether a model is more likely to overfit or underfit by altering its *capacity*, that is its ability to fit a certain variety of functions – e.g. linear regression as a lower capacity than polynomial regression – and one way to control the capacity of a learning algorithm is by choosing its *hypothesis space*, the set of functions that the learning algorithms is able to approximate, appropriately.

Regularization refers to strategies designed to reduce the generalization error, often paying the price of higher training error. They can all be seen as a way to reduce the capacity of the model. In general such strategies can be categorized into two classes:

1. Strategies that put extra constraints on a learning model (e.g. adding restrictions on parameters' values or on the activation functions)

2. Strategies that add extra terms in the objective function

These constraints and penalties can be generally interpreted as encoding some specific kind of prior knowledge or expressing preferences for simpler model classes in order to improve generalization reducing the threat of overfitting.

In the context of deep learning, most regularization strategies are based on regularizing estimators that have the effect of increasing bias while reducing variance. As usual, the bias of an estimator is defined as $\text{Bias}(\hat{\theta}) = \text{E}_{\hat{p}_{data}}\big(\hat{\theta} - \theta\big)$ which defines the expected error of the estimator, while the variance $\text{V}(\hat{\theta})$ defines the error in the estimation stemming from the sensitivity to small fluctuations in the sample observed. High bias can cause an algorithm to mistake the mapping between features and target outputs (underfitting), while high variance can cause an algorithm to model the random noise in the training data rather than the intended outputs (overfitting). Therefore, an effective regularizer is one that is able to find the right trade-off, reducing variance significantly while not overly increasing the bias.

Via regularization we want to restrict at minimum the capacity of the model to include the true data-generating process ruling out other possible candidate data-generating processes in the scope of the model capacity (reducing the variance). This conditions of the model capacity should not be too restrictive: the risk is to rule out of the scope of the model capacity the true data-generating process (bias). Unfortunately, we almost never have access to the true data-generating process and thus we can never be completely sure that the class of models estimated includes the true generating process.

3.2.1 Parameter Norm Penalties

A commonly used example of regularization approach belonging to the second class of strategies are the *parameter norm penalties*. These approach entails limiting the capacity of the model by adding a penalty in the loss function $J(\theta)$ proportional to the *parameter norm* $\Omega(\theta)$. Note that usually for neural networks the norm is computed only on the weights W, while the biases b are left unregularized. The regularized cost function can be written as

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta) \tag{3.5}$$

where $\alpha \in [0, \infty)$ weights how the norm penalty affects the regularized objective function relative to the standard loss function. In neural networks, usually, the penalties are separated per layer and a different α is used for each layer (Bishop 2006). In practical applications, due to computational costs, the definition of Ω used is often the same for each layer. Different definitions of Ω result in different solutions being preferred. The two widely used definitions are:

L^2-norm. In this case we have $\Omega(\theta) = \frac{1}{2}\|\theta\|_2^2$, commonly known as **weight decay**. Therefore the regularized loss becomes

$$\begin{aligned} \tilde{J}(\theta) &= \frac{\alpha}{2}\theta^\top\theta + J(\theta) \\ \nabla_\theta \tilde{J}(\theta) &= \alpha\theta + \nabla_\theta J(\theta) \end{aligned}$$

defining the update rule

$$\theta^{new} \leftarrow \theta - \lambda\big(\alpha\theta + \nabla_\theta J(\theta)\big)$$

The effect of the regularization term is to shrink θ by a constant factor on each step, just before performing the usual gradient update.

L^1-norm. In this case we have $\Omega(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$, obtaining

$$\begin{aligned} \tilde{J}(\theta) &= \alpha\|\theta\|_1 + J(\theta) \\ \nabla_\theta \tilde{J}(\theta) &= \alpha sign(\theta) + \nabla_\theta J(\theta) \end{aligned}$$

In this case, the gradient does not scale linearly as with the L^2-norm definition, but it is a constant factor with a sign equal to the sign of θ. The gradient so defined has no longer an algebraic solution and must be approximated. It can be shown (Bishop 2006; Goodfellow et al. 2016) that L^1 regularization results in a solution that is more sparse, i.e. parameters will tend to have optimal value of zero. This can be interpreted as a form of variable selection mechanism. The LASSO is an example of this mechanism employed in training linear models.

The parameter norm penalties regularization can also be interpreted as MAP Bayesian inference: e.g. L^2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on θ.

3.2.2 Early Stopping

Often when training neural network with sufficient representational capacity, a signal that the model is overfitting is that training error decreases steadily over time, but validation set error begins to rise again. The validation set is used to evaluate a given model, like the test set, frequently during training. It corresponds to a small portion of the data that are mainly used to fine-tune the model hyperparameters. Hence the model "occasionally" sees the validation data, but never learns from them.

In principle, we can expect that minimizing the validation set error would also lead to better test error. Therefore, rather than stopping the optimization algorithm when the training error is minimum, it is possible to stop it earlier, when the validation error is the least, i.e. we use the parameters values in correspondence of the least validation error, rather than the latest values returned by the optimization algorithm.

This strategy is known as *early stopping*. In practise, the optimization algorithm is instructed to stop when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. In deep learning, this is the most commonly used regularization strategy. In this way, we control how many times the optimization algorithm can run to fit the training set, limiting the threat of overfitting. This strategy does not require any change in the underlying procedure, loss function, or parameter set.

3.2.3 Sparse Representations

Another approach to regularize neural networks is to impose a penalty on the activation functions of the units by introducing sparseness. Essentially, the model is brought to prefer configurations in which more hidden units are set to zero; this reduces the capacity of the model. Indirectly, it is like imposing a complex penalization on the model parameters. As mentioned in subsection (§3.2.1), also L^1-penalization introduces sparseness in the parameters values.

Sparseness is induced in the representations, i.e. components of the hidden layer h_l are set to zero, not in the components of the parameters W_l or b_l. In practice, this regularization is applied to the model via adding a parameter norm penalty over the dimension of the hidden units in the loss function

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(h), \quad \alpha \in [0, \infty)$$

Note how the argument of Ω is now the h vector and not θ.

3.2.4 Bagging and Ensemble Methods

The underlying idea is to reduce generalization error combining different models. In practise it is performed by training different models separately and for each input, x_i, each of these models collects the proposal $f_j(x_i)$, for $j \in \{1, 2, \dots\}$ denoting the number of models considered. Techniques using this strategy, known as *model averaging*, go under the label of *ensemble methods*.

On average the ensemble performs at least as well as any of its members (Goodfellow et al. 2016). In case the errors made by the individual models are independent, the ensemble performs significantly better than any single one of them.

A particular example is the *bagging* method (Bishop 2006), short for "bootstrap aggregating". This approach allows the very same model to be reused, instead of fitting different models. In practise k datasets with the same number of observations as the training set are constructed by sampling with replacement from the training set. The model is trained on dataset $i \in \{1, \dots, k\}$. Given that the exact same model is used, the difference among the resulting model outputs is due to the different k "artificial" datasets on which the model is trained.

3.2.5 Dropout

Dropout is a type of ensemble method, but for the importance it has acquired in practical applications it deserves a dedicated space.

Bagging entails training the same model multiple times on different training sets. For large neural networks the computational cost soon skyrockets. Dropout provides a computationally cheap approximation to train and evaluate a "bagged" ensemble. In particular, dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from the underlying base network. The number of submodels, thus, grows exponentially in the number of units. Switching off a unit – in the input layer or hidden layers – is performed by multiplying it by zero. Dropout applied to input units serves as a form of variable selection (Hinton and Salakhutdinov 2006; Srivastava et al. 2014), interpretable as Bayesian ridge regression (Polson and Sokolov 2017). When applied to hidden layers it regularizes the choice of the number of hidden units in a layer. Once a variable from a layer is dropped, all terms above it in the network also disappear.

In practise, dropout removes units in the input layer and/or in the hidden layers randomly with a given probability p, different for each layer. Training with dropout is performed using stochastic optimization methods like SGD. Each time a minibatch is created, for each layer, a binary vector ζ_l is randomly sampled, where each component is distributed as a Bernoulli(p_l). The number of components in the binary vector matches the number of units in the layer. This vector is usually called **mask**. The components of ζ are sampled independently of each other. The probabilities p_l are hyperparameters fixed before training begins. Therefore the original input layer becomes $\tilde{x} = x \odot \zeta_0$, where $\odot$ represents the Hadamard (or element-wise) product. Similarly, each hidden layer becomes $\tilde{h}_l = \zeta_l \odot h_l,\ l = 1, \dots, L$. The same values of the binary vector are used during optimization, namely when computing the derivatives.

Let $J(\theta, \zeta)$ define the cost of the model defined by parameters θ and the mask ζ. Marginalizing over the randomness, the objective becomes

$$\mathrm{E}_{\zeta \sim \mathrm{Ber}(p)} J(\theta, \zeta)$$

The expectation contains exponentially many terms, but an unbiased estimate of its gradient can be obtained by sampling and using only certain values of $\{\zeta_0, \dots, \zeta_L\}$. This differentiate dropout from bagging: in the latter case, each model is trained until convergence on its respective training set; in the former case, most models are not explicitly trained at all, but only a tiny fraction of the possible subnetworks are trained for a single step, and the parameter sharing, across models with the same active units, causes the remaining subnetworks to arrive at good settings of the parameters.

To conclude, we have to highlight that optimization, per se, does not provide an uncertainty measure. This is solved by taking a Bayesian approach.

3.3 Bayesian Neural Networks

Bayesian Neural Networks (BNN) were first suggested in the '90s and studied extensively since then (MacKay 1992; Neal 1996), they are now living a period of huge fame. They offer a probabilistic interpretation of neural networks models by inferring distributions over the weights of the model. The model offers robustness to overfitting, uncertainty estimates, and can easily learn from small datasets. As the name suggests, a Bayesian neural network is a neural network equipped with a prior distribution over its weights θ.

Consider an *i.i.d.* data set of $\{(x_i, y_i)\}_{i=1}^N$. For illustration purposes, we consider a canonical regression model of the form

$$y_i = \mathrm{NN}_\theta(x_i) + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$$

where σ^2 is assumed known. Therefore, the distribution for each data point is Gaussian, with a mean dependent on the inputs via a neural networks whose parameters are denoted by θ. The likelihood – i.e. the probabilistic model by which the inputs generate the outputs given some parameter setting θ – can be, thus, written as

$$p(y|x,\theta) = \prod_{i=1}^N \mathcal{N}\big(\mathrm{NN}_\theta(x_i), \sigma^2\big)$$

Following the Bayesian approach we would specify a prior distribution for the unknown parameters, $p(\theta)$. This distribution represents our prior belief as to which parameters are likely to have generated our data before we observe any data point. Once some data are observed, this distribution will be transformed to capture the more likely and less likely parameters given the observed data points. We define a Gaussian prior distribution over the weights θ, as is usually done for conjugacy reasons

$$p(\theta) = \mathcal{N}\big(0, I\big)$$

By applying the Bayes's theorem, the resulting posterior distribution is

$$p(\theta|y,x) = \frac{p(y|x,\theta)\ p(\theta)}{p(y|x)}$$

where $p(y|x) = \int p(y|x,\theta)\ p(\theta)\ d\theta$. As a consequence of the nonlinear dependence of $\mathrm{NN}_\theta(x)$ on θ, it will be non-Gaussian.

Once we compute the posterior distribution, we can predict an output for a new input point x^* by integrating out the parameters

$$p(y^*|x^*) = \int p(y^*|x^*,\theta)\ p(\theta|x,y)\ d\theta$$

Performing this integration is also referred to as *marginalising* the likelihood over θ, which explains the alternative name for the model evidence: *marginal likelihood.* Marginalisation can be done analytically for simple models such as Bayesian linear regression. In such models the prior is conjugate to the likelihood, and the integral can be solved with known tools of calculus. Marginalisation is the one of core parts in Bayesian modelling, and ideally we would want to marginalise over all uncertain quantities – i.e. averaging with respect to all possible model parameter values θ, each weighted by its plausibility $p(\theta)$. But with more interesting models this marginalisation cannot be done analytically. In such cases an approximation is needed.

3.4 Approximation Methods

Modern research in BNN often relies on either *variational inference*, or *sampling based techniques* like MCMC. Each approach has its merits and its limitations. MCMC methods tend to be more computationally intensive than variational inference, but they also provide guarantees of producing (asymptotically) exact samples from the target density (Robert and Casella 2005). Variational inference does not enjoy such guarantees: it can only find a density close to the target, but tends to be faster than MCMC. Because it rests on optimization, variational inference easily takes advantage of methods like stochastic and/or distributed optimization. Some MCMC methods can exploit these innovations (Welling and Teh 2011) as well, but they still are not as well established as variational inference.

To provide an overview of the method, let's setup the general problem. Consider a joint density of latent variables $z = (z_1, \ldots, z_m)$ and observations $x = (x_1, \ldots, x_n)$. For example, in the case of a Gaussian distribution with unknown mean and variance, $z = (\mu, \sigma^2)$. In the case of a neural network, z is equal to all weights in the network. We can write

$$p(x, z) = p(z)\ p(x|z)$$

The generative process of a typical Bayesian model is the following: draw the latent variables from a prior density $p(z)$ and then relate them to the observations through the likelihood $p(x|z)$. Inference, thus, amounts to conditioning on data and computing the posterior distribution $p(z|x)$. In complex Bayesian models, like Bayesian neural nets, this computation often requires approximate inference, as we have seen at the end of the previous section.

As discussed above, until the rise in popularity of Bayesian neural networks the dominant paradigm for approximate inference has been MCMC. In MCMC, we first construct an ergodic Markov Chain on z whose stationary distribution is the posterior distribution $p(z|x)$. Then, we sample from the chain to collect samples from the stationary distribution. Finally, we approximate the posterior distribution with an empirical estimate constructed from a subset of the collected samples.

The key characteristic of variational inference, on the other hand, is that it transforms inference problems into optimization. Thus, variational inference is suited for large data sets and scenarios where we want to quickly explore many models; MCMC performs well on smaller data sets and scenarios where we are willing to pay a heavier computational cost for more precise samples.

In what follows we will focus on variational inference (VI) which is the workhorse for Bayesian inference in modern Bayesian Deep Learning (Gal and Ghahramani 2016; Gal 2016; Blundell et al. 2015; Polson and Sokolov 2017; Blei et al. 2017; Kim et al. 2019).

3.4.1 Preliminaries

How do we implement variational inference? First, we assume a family of approximate densities $\mathcal{Q}$, that is a set of densities over the space of the latent variables. Then, we try to find the member of that family that minimizes the Kullback-Leibler (KL) (Kullback and Leibler 1951) divergence from the exact posterior distribution,

$$q^\star(z) = \operatorname*{arg\,min}_{q(z)\in\mathcal{Q}} \mathrm{KL}\big[q(z)\|p(z|x)\big] \tag{3.6}$$

Finally, we approximate the posterior with the optimized member of the family $q^*(z)$. In other words, the goal in VI is to approximate the posterior by a simpler distribution. To this end, one minimizes the KL divergence between the approximating distribution and the posterior. One of the key ideas behind variational inference is to choose $\mathcal{Q}$ to be flexible enough to capture a density close to $p(z|x)$, but simple enough for efficient optimization.

Unfortunately, the objective, (3.6), is not computable because it requires computing the evidence $\log p(x)$. In fact $p(z|x) = p(x,z)/p(x)$. To appreciate this recall that the KL divergence is defined as

$$\mathrm{KL}\big[q(z)\|p(z|x)\big] = \mathrm{E}\big[\log q(z)\big] - \mathrm{E}\big[\log p(z|x)\big]$$

where all expectations are taken with respect to $q(z)$. Expanding the conditional distribution,

$$= \mathrm{E}\big[\log q(z)\big] - \mathrm{E}\big[\log p(z,x)\big] + \log p(x) \tag{3.7}$$

and the dependence on the log-evidence is revealed.

Since we cannot compute this KL divergence directly, we optimize an alternative objective that is equivalent to the KL divergence up to an *added* constant,

$$\mathrm{ELBO}(q) = \mathrm{E}\big[\log p(z,x)\big] - \mathrm{E}\big[\log q(z)\big]$$

This function is called the **evidence lower bound** (ELBO). The ELBO is the negative KL divergence in eq. (3.7) plus $\log p(x)$, which is a constant with respect to $q(z)$. Maximizing the ELBO is equivalent to minimizing the KL divergence in eq. (3.7). To make explicit the dependence of the ELBO on the conditional log-likelihood of the data given the latent variables, we can rewrite $p(z,x) = p(x|z)\ p(z)$, thus

$$\begin{aligned}\mathrm{ELBO}(q) &= \mathrm{E}\big[\log p(x|z)\big] + \mathrm{E}\big[\log p(z)\big] - \mathrm{E}\big[\log q(z)\big] \\ &= \mathrm{E}\big[\log p(x|z)\big] - \mathrm{KL}\big[q(z)\|p(z)\big]\end{aligned} \tag{3.8}$$

The first term is the expected log-likelihood: it encourages densities that place their mass on configurations of the latent variables that explain the observed data well. The second term is the negative divergence between the variational density and the prior distribution: it favours densities close to the prior distribution. Thus the variational objective mirrors the usual balance between the likelihood and prior distribution. The KL divergence term can also be interpreted as a form of regularization that favours simpler models.

Furthermore, rewriting eq. (3.7) as

$$\log p(x) = \mathrm{KL}\big[q(z) \| p(z|x)\big] + \mathrm{ELBO}$$

and considering that $\mathrm{KL}(\cdot) \geq 0$, we can assess that the ELBO is a lower-bound for the marginal log-evidence, $\log p(x) \geq \mathrm{ELBO}(q)$ for any $q(z)$. This was originally shown via the Jensen's inequality in Jordan et al. (1999). To sum up, this approach circumvents computing intractable normalization constants.

3.4.2 Classical Variational Inference

We slightly modify the simplified notation of the previous section making explicit the parametrization of both p_θ and q_ϕ. To simplify the exposition, for now we assume that θ and ϕ are random variables that do not call for a Bayesian treatment, they are estimated by maximum likelihood. This assumption will be relaxed in section (§4.2.2).

As in the previous section, the model has observations x and latent random variables z whose distributions are both parametrized by θ. The joint probability density is of the form

$$p_\theta(x, z) = p_\theta(x|z)\; p_\theta(z)$$

This kind of model is also called *latent variable model.* It is a probability distribution over two sets of variables, x and z, where the former are observed and the latter are never observed.

Our goal is to compute the maximum likelihood estimate of the parameters

$$\hat{\theta}_{MLE} = \arg\max_\theta \; \log p_\theta(x)$$

as well as the posterior over the latent variables

$$p_{\hat{\theta}}(z|x) = \frac{p_{\hat{\theta}}(x,z)}{\int p_{\hat{\theta}}(x,z)\ dz}$$

Variational inference offers a scheme for calculating $\hat{\theta}_{MLE}$ and computing an approximation to the posterior $p_{\hat{\theta}}(z|x)$. As said in the previous section, even with θ known, the integral would be intractable. Hence, we apply variational inference introducing the *variational distribution* $q_\phi(z)$ parametrized by the variational parameters ϕ. The members of the family of approximate densities $\mathcal{Q}$ on z is, thus, indexed by ϕ. Equation (3.6) can be rewritten as

$$\phi^* = \arg\min_\phi \mathrm{KL}\big[q_\phi(z) \| p_\theta(z|x)\big] \tag{3.9}$$

Following the same reasoning of eq. (3.8), we can rewrite the ELBO as a function of the variational parameters

$$\begin{aligned}\mathrm{ELBO}(\phi) &= \mathrm{E}\big[\log p_\theta(x,z)\big] - \mathrm{E}\big[\log q_\phi(z)\big] \\ &= \mathrm{E}\big[\log p_\theta(x|z)\big] - \mathrm{KL}\big[q_\phi(z) \| p_\theta(z)\big] \end{aligned} \tag{3.10}$$

where all the expectations are with respect to q_ϕ. Therefore, we can now state that VI turns Bayesian inference into an optimization problem over variational parameters. To complete the specification of the optimization problem, we have to define the family of variational distributions $\mathcal{Q}$.

In traditional VI, computing the ELBO amounts to analytically solving the expectations over q. This restricts the class of tractable models and thus restrict our choice regarding the family $\mathcal{Q}$. The **mean-field variational family** is a class of distributions whose members are of the form

$$q_\phi(z) = \prod_i q_{\phi_i}(z_i)$$

The latent variables in z are mutually independent, each one is described by a different factor in the variational density, and each factor is governed by its own variational parameter. In principle, each of the factors can take on any parametric form appropriate to the corresponding random variable. A fully factorized variational distribution allows one to optimize the ELBO via simple iterative updates. A common optimization algorithm for this case is the *coordinate ascent variational inference* (CAVI) (Blei et al. 2017).

The mean-field family, though, is limited in multiple ways when it comes to modern applications. The three major drawbacks are the following:

- Although it is expressive because it can capture any marginal density of the latent variables, it cannot capture correlations among them
- It is not scalable to big datasets
- To use CAVI we have to be able to express the ELBO analytically

These three limitations are huge constraints especially in a deep learning context where the ELBO cannot be evaluated analytically. All these limitations will be addressed in the next chapter, where we will show that VI can be extended to accommodate all the needs of deep learning applications.

3.5 Bayesian Deep Learning

With Bayesian deep learning we refer to the application of Bayesian methods to deep learning models, in particular deep neural nets. Most of the research in this field is focused on the statistical interpretation of regularization or optimization procedures customarily used in training neural networks. Bayesian estimation is usually brought about via variational inference. The aim is to demonstrate that the cost functions minimized by applying variational methods actually coincides, or can be interpreted, as the cost functions minimized using practical regularization techniques.

For example, Gal and Ghahramani (2016) and Gal (2016) proved that training deep neural nets with dropout corresponds, under mild assumptions on the shape of the variational distribution, to approximate inference in deep neural nets. Dropout, randomly eliminating some nodes of the network, injects noise into the feature space. In these two seminal works, the authors proved that this noise can be transformed into noise on the parameter space. This procedure is referred to as **MC dropout**.

Furthermore, Salas et al. (2018) provide a probabilistic interpretation of adaptive subgradient methods such as AdaGrad and Adam. They set out a framework in which it is possible to perform inference on the posterior distribution of the weights of a deep neural network. They start by applying a second-order expansion to the cost function around the current iterate of the stochastic optimization, that is around the value of the parameters at the current iteration of the stochastic gradient descent process. By imposing an improper prior on the network weights (constant), they recover an approximate posterior distribution for the network weights.

In the next chapter we will discuss Deep Bayesian Learning which refers to the application of deep learning to Bayesian methods. In particular, we will discuss how using deep neural networks to parameterize distributions can improve inference.

Chapter 4

Deep Bayesian Learning

How do we perform Bayesian inference in complex, intractable models in a computationally efficient way? In this chapter we discuss very recent extensions of Classical (Mean-Field) Variational Inference. These innovations prompted the development of what is referred to as Deep Bayesian Learning which entails the use of deep learning architectures to parametrize probability density functions in the context of Bayesian inference. This differs from Bayesian Deep Learning since we are not interested in making inference about the parameters of a neural network, but we want to use neural networks to facilitate Bayesian inference in complicated models in a computationally efficient way. Two examples are provided at the end of the chapter: Variational Autoencoders and Conditional Variational Autoencoders, two instances of the large class of Deep Latent Variable Models.

In the previous chapter we reviewed the application of Bayesian methods to solving inference problems in neural networks. In this chapter we approach the problem from a different perspective.

Deep Bayesian Learning refers to the application of deep learning to Bayesian methods. In practical terms, deep neural nets can be used to parameterize distributions, allowing to capture richer characteristics of the data improving inference. To make this approach feasible, a series of innovations in the context of approximate inference have been proposed in recent years and the literature in this field is flourishing.

To apply deep learning to Bayesian inference techniques, in particular to variational inference, a series of improvements (Kingma and Welling 2013;

Rezende et al. 2014; Blundell et al. 2015; Gal and Ghahramani 2016; Gal 2016; Hoffman et al. 2013; Ranganath et al. 2014) have been proposed to overcome the limitations outlined in the previous chapter.

As we discussed at the end of the chapter 3, classical VI – also known as *mean-field VI* – suffers three major drawbacks that can summarized as follows:

- It does not scale well to large datasets
- It is restricted to a limited class of variational distributions
- It is unable to account for correlations among latent variables

These issues will be addressed in the next sections. Notice that the KL term of the ELBO is usually not a concern in practical applications and, often, the distributions involved are chosen in a way to make this term have an analytical expression. The focus is, thus, on the log-likelihood term of the ELBO, which requires a specific treatment.

4.1 Scalable Variational Inference

To make VI tractable for more complex models like deep neural nets, VI is combined with stochastic optimization. **Stochastic variational inference** (SVI) exploits stochastic gradient descent (SGD) to scale VI to large datasets. It was originally proposed by Hoffman et al. (2013).

As we have seen in chapter 2 for general loss functions, for many models of interest also the variational objective has a special structure, namely, it is the sum over contributions from all N individual data points

$$\mathrm{ELBO}(x, \theta, \phi) = \sum_{i=1}^{N} \mathrm{ELBO}(x_i, \theta, \phi)$$

Problems of this type can be solved efficiently using stochastic optimization. Therefore, to apply stochastic optimization we have to assess that both the model and the variational distribution have some conditional independence structure that we can take advantage of. The most common case is the one in which the observations are conditionally independent given the latent variables, hence the log-likelihood term in the ELBO can be approximated with

$$\text{ELBO}(x, \theta, \phi) \approx \frac{N}{M} \sum_{\mathcal{I}_M} \text{ELBO}(x_i, \theta, \phi)$$

where I_M is a mini-batch of indices in $I = \{1, \ldots, i, \ldots, N\}$ of size M with $M < N$. In other words, we can cheaply obtain noisy estimates of the gradient by subsampling the data and computing a scaled gradient on the subsample. If we sample independently then the expectation of this noisy gradient is equal to the true gradient. The problem is now cast has a pure optimization problem: all properties discussed in section (§3.1) apply.

To maximize the ELBO using this approximation, we would like to take gradient steps in the parameter space $\{\theta, \phi\}$, that is, we need to be able to compute unbiased estimates of

$$\nabla_{\theta,\phi}\text{ELBO}(x, \theta, \phi) = \nabla_{\theta,\phi}\text{E}_{z\sim q_\phi(z)}\left[\log p_\theta(z, x) - \log q_\phi(z)\right]$$

where we have made explicit that the expectations are computed with respect to q. Computing this gradient could be difficult if the ELBO does not have a "nice" analytical form. Variational inference was originally limited to conditionally conjugate models, for which the ELBO could be computed analytically before it is optimized (Hoffman et al. 2013). In the next section, we introduce methods that relax this requirement and simplify inference. Central to this section are stochastic gradient estimators of the ELBO that can be computed for a broader class of models without requiring its the direct evaluation.

4.2 Generic Variational Inference

In classical VI the ELBO is first derived analytically, and then optimized. For many models, including Bayesian deep learning architectures or complex hierarchical models, the ELBO contains intractable expectations with no known or simple analytical solution. Even if an analytic solution is available, the analytical derivation of the ELBO often requires time and mathematical expertise.

Black-Box variational inference (BBVI) removes the need for an analytic expression of the ELBO – this explains the origin of its name. It proposes a generic inference algorithm for which only the generative process of the data has to be specified.

The main idea is to represent the gradient as an expectation, and to use Monte Carlo techniques to estimate this expectation. The key insight of BBVI is that one can obtain an unbiased gradient estimator by sampling from the variational distribution without having to compute the ELBO analytically (Paisley et al. 2012). The goal of using Monte Carlo (MC) estimation in variational inference to estimate the expected log-likelihood is only useful if we are then able to compute the expected log-likelihood derivative with respect to θ and ϕ.

There exist two main techniques for MC estimation of the derivative of the ELBO in the VI literature[1]:

- Score function gradient (Ranganath et al. 2014; Mnih and Gregor 2014)
- Reparameterization gradient (Kingma and Welling 2013; Rezende et al. 2014)

These have very different characteristics; notably, the variances of the two resulting estimators of the gradient of the ELBO differ significantly. A thorough analysis of the variances of this two estimators can be found in Gal (2016). In general, reparametrization gradients have been proved empirically to have lower variance, although no formal proof has been given in the literature.

A question spontaneously arises: how do we apply variational inference to problems entailing general stochastic functions – one which we are not able to evaluate analytically, but we are able to sample from – and general variational distributions? We review in the next subsections the two main approaches.

4.2.1 Score Function Gradient

Adopting the score function gradient approach, we rewrite the derivative of the objective as an expectation with respect to the variational distribution, and then we sample from the variational approximation to get noisy but unbiased

[1]A brief survey of the literature can be found in Schulman et al. (2015) and in Zhang et al. (2018)

gradients that we use to update our parameters. For each sample, our noisy gradient requires evaluating

- The joint distribution of the observed and latent variables
- The variational distribution
- The gradient of the log of the variational distribution

This is a black box method since it can be derived once for each type of variational distribution and reused for many models and applications.

This MC approximation method is also known as a *likelihood ratio estimator* or *REINFORCE* in the literature. It relies on the identity $\frac{\partial}{\partial\nu} f_\nu(u) = f_\nu(u) \frac{\partial}{\partial\nu} \log f_\nu(u)$. In what follows, to have a clearer notation, we denote

$$f_{\theta,\phi}(z) \overset{def}{=} \log p_\theta(z,x) - \log q_\phi(z) \tag{4.1}$$

so that ELBO $= \mathrm{E}[f_{\theta,\phi}(z)]$. For clarity, we discard all subscripts in the derivation below. We begin by expanding the gradient of interest as

$$\begin{aligned}\nabla \mathrm{E}[f(z)] &= \nabla \int q(z) f(z)\, dz \\ &= \int \Big(\nabla q(z)\Big) f(z) + q(z) \Big(\nabla f(z)\Big)\, dz\end{aligned}$$

The problem is that, although we know how to generate samples from $q(z)$, we have troubles computing the expectation with respect to $\nabla q(z)$, which need not even be a proper probability density function. We use the identity presented above, eq. (4.1), to write

$$\begin{aligned}&= \int q(z) \Big(\nabla \log q(z)\Big) f(z) + q(z) \Big(\nabla f(z)\Big)\, dz \\ &= \int q(z) \left[\Big(\nabla \log q(z)\Big) f(z) + \Big(\nabla f(z)\Big) \right]\, dz\end{aligned}$$

to finally obtain

$$= \mathrm{E}\left[\Big(\nabla \log q(z)\Big) f(z) + \Big(\nabla f(z)\Big) \right]$$

Crucially, the gradient has been moved inside the expectation. Therefore, the gradient with respect to the model parameters θ is straightforward

$$\nabla_\theta \text{ELBO} = \text{E}\big[\nabla_\theta \log p_\theta(z, x)\big]$$

The corresponding gradient with respect to the variational parameters is somewhat more involved and requires the use of the identity above[2]

$$\nabla_\phi \text{ELBO} = \text{E}\big[\nabla_\phi \log q_\phi(z) \, \big(\log p_\theta(x, z) - \log q_\phi(z)\big)\big]$$

As both gradients involve expectations which are intractable, we can use Monte Carlo approximation using samples from the variational distribution. Generating S samples $z^{(1)}, \ldots, z^{(s)}$ from $q_\phi(z)$, we can compute

$$\nabla_\theta \text{ELBO}(x, \theta, \phi) \approx \frac{1}{S} \sum_{s=1}^{S} \nabla_\theta \log p_\theta \left(x, z^{(s)}\right) \tag{4.2}$$

$$\nabla_\phi \text{ELBO}(x, \theta, \phi) \approx \frac{1}{S} \sum_{s=1}^{S} \nabla_\phi \log q_\phi \left(z^{(s)}\right) \, \left(\log p_\theta \left(x, z^{(s)}\right) - \log q_\phi \left(z^{(s)}\right)\right)$$

In this way, we have specified a Monte Carlo estimator that does not require to compute evaluate the ELBO explicitly. We only need to be able to generate from the variational distribution and the joint ditribution of latent variables and observations.

4.2.2 Reparametrization Gradient

An alternative to the score function gradients are the so-called reparametrization gradients. These gradients are obtained by representing the variational distribution as a *deterministic parametric* transformation of a noise distribution. Empirically, reparametrization gradients are often found to have lower variance than score function gradients, however, they are less generally applicable since we have to be able to reparametrize the variational distribution.

The **reparameterization trick** – as this approach is also known in the machine learning literature – allows to estimate the gradient of the ELBO by Monte Carlo samples by representing random variables, z, as deterministic functions of noise distributions. This allows to compute stochastic gradients for a large

[2]For a complete derivation consult Ranganath et al. (2014) appendix A.

class of models without having to compute analytic expectations. In particular, if we are able to rewrite a random variable z distributed according to $q_\phi(z)$ as a transformation of a random variable $\varepsilon \sim p(\varepsilon)$, a noise distribution such as uniform or Gaussian, we can compute any expectation over z as an expectation over ε[3].

More generally, to apply this method we need to be able to write $z = g(\varepsilon, \phi)$ for a deterministic parametric and differentiable function g. The noise distribution $p(\varepsilon)$ must not depend on the variational parameters. Therefore $q_\phi(z)$ and $g(\varepsilon, \phi)$ share the same parameters ϕ. This allows to compute any expectation over z as an expectation over ε by the theory behind the change of variables in integrals since, following this process, generating z from g is equivalent to directly drawing it from the original distribution.

Using the same notation as in eq. (4.1), we can rewrite the the gradient as follows

$$\begin{aligned}\nabla_{\theta,\phi}\text{ELBO} &= \nabla_{\theta,\phi}\text{E}_{q_\phi(z)}\big[f_{\theta,\phi}(z)\big]\\ &= \text{E}_{p(\varepsilon)}\big[\nabla_{\theta,\phi}f_{\theta,\phi}\big(g(\varepsilon,\phi)\big)\big]\end{aligned}$$

Now the expectation is with respect to $p(\varepsilon)$. Again, crucially, the gradient has been shifted inside the expectation. Therefore, the Monte Carlo estimator for the ELBO can be computed generating S samples $\varepsilon^{(1)}, \dots, \varepsilon^{(s)}$ from $p(\varepsilon)$, and using the following approximation

$$\nabla_{\theta,\phi}\text{ELBO}(x,\theta,\phi) \approx \frac{1}{S}\sum_{s=1}^{S}\nabla_{\theta,\phi}\left[\log p_\theta\big(x, g(\varepsilon_s,\phi)\big) - \log q_\phi\big(g(\varepsilon_s,\phi)\big)\right]$$

Again, in this way we get around the obstacle of evaluating the ELBO explicitly, we only need to be able to generate from p and q. In other words, the estimator only depends on samples from $p(\varepsilon)$ which is not influenced by ϕ, therefore the estimator can be differentiated with respect to ϕ.

An fully Bayesian extension of this approach regards the estimation of the parameters θ via variational inference, not just maximum likelihood. We assign

[3]For example, if $z \sim \mathcal{N}(\mu, \sigma^2)$, then $z = \mu + \sigma\varepsilon$, for $\varepsilon \sim \mathcal{N}(0, I)$.

a hyperprior to θ as well. The model becomes

$$\begin{aligned}\theta &\sim p_\alpha(\theta)\\ x_i, z_i|\theta &\sim p(x, z|\theta)\end{aligned}$$

where the distribution of θ is governed by the hyperparameters α. We want to compute the posterior of θ given the data, that is $p(\theta|x) = p(x|\theta)p(\theta)/p(x)$, as well as the posterior of z given the data, that is $p(z|x, \theta)$. We use variational inference to approximate both posteriors. It can be shown (Kingma and Welling 2013) that the resulting ELBO is the following

$$\mathrm{ELBO}(x, \theta, \phi) = \log p(x|z, \theta) - \mathrm{KL}\big[q_\phi(z)\|p(z|\theta)\big] - \mathrm{KL}\big[q_\phi(\theta)\|p_\alpha(\theta)\big]$$

The reparametrization trick can be applied to both z and θ, following the same path showed above.

4.3 Structured Variational Inference

The methods we have exposed above usually addresses the standard mean-field variational inference (MFVI) setup and employs the KL divergence as a measure of distance between distributions.

Mean-field methods were first adopted in neural networks by Anderson and Peterson in 1987 (Zhang et al. 2018), and later gained popularity in the machine learning community (Jordan et al. 1999). The main limitation of mean-field approximations is that they explicitly ignore correlations between different latent variables, indeed, MFVI assumes a fully-factorized variational distribution. Fully factorized variational models have limited accuracy, especially when the latent variables are highly dependent such as in models with hierarchical structures. Allowing a structured variational distribution to capture dependencies between latent variables is a modeling choice; different dependencies may be more or less relevant and depend on the model under consideration.

For many models, the variational approximation can be made more expressive by maintaining dependencies between latent variables, but these dependencies make it harder to estimate the gradient of the variational bound.

In order to capture dependencies between latent variables, one starts with a mean-field variational distribution $\prod_i q(z_i|\phi)^4$, but instead of assuming independence, one assumes conditional independence. Consider the most general case in which each data point x_i depends on a *local* latent variable z_i whose distribution is governed by the parameters ϕ_i. In turn, the local latent variables z_i's are *conditionally independent* given a *global* latent variable v. We can, thus, write

$$q_\phi(z) = \int p(v) \prod_i q(z_i|v, \phi_i)\, dv$$

In such a way, we can allow dependencies among the latent variables z_i's.

4.4 Amortized VI

The term *amortized* "inference" refers to utilizing inferences from past computations to support future computations (Ritchie et al. 2016). In the context of variational inference, amortized inference refers to inference over local variables. Usually, for each data point x_i a *local* latent variable z_i is defined, whose parameters are ϕ_i. This is true even when a structured VI approach is taken, the only difference being that such local latent variables are *conditionally* independent. Traditional VI makes it necessary to optimize a ϕ_i for each data point x_i, which is computationally expensive, in particular when this optimization is embedded a global parameter update loop.

Instead of approximating separate variables for each data point, amortized VI assumes that the local variational parameters can be predicted by a parameterized function of the data. Once this function is estimated, the latent variables can be acquired by passing new data points through the function. Deep neural networks used in this context are also called *inference networks* (Kingma and Welling 2013). Amortized VI with inference networks thus combines probabilistic modeling with the representational power of deep learning.

The basic idea behind amortized inference is to use a powerful predictor to predict the optimal z_i based on the data x_i, i.e. $z_i = h_\xi(x_i)$. This way, the local variational parameters are replaced by a function of the data whose parameters are shared across all data points, i.e. inference is *amortized.*

[4]We made clear that $q(\cdot)$ is the conditional distribution of z given ϕ.

Consider, again, a model with global and local latent random variables and local variational parameters:

$$p(x, z, v) = p(v) \prod_{i=1}^{N} p(x_i|z_i)p(z_i|v) \qquad q(z, v) = q(v) \prod_{i=1}^{N} q(z_i|v, \phi_i)$$

For small to medium-sized N using local variational parameters like this can be a good approach. If N is large, however, the fact that the space we are doing optimization over grows with N is a crucial problem. One way to avoid the growth of variational parameters induced by the size of the dataset is to use *amortization.*

Instead of introducing local variational parameters, ϕ, we learn a single parametric function $h(\cdot)$ and work with a variational distribution that has the form

$$q(v) \prod_{i=1}^{N} q(z_i|h(x_i))$$

The function $h(\cdot)$ maps a given observation to a set of variational parameters tailored to that data point and needs to be sufficiently rich to capture the posterior accurately, but now we can handle large datasets without having to introduce one variational parameter for each data point: the number of parameters involved is only equal to the number of parameters xi parameterizing h. This approach has other benefits too: learning $h(\cdot)$ effectively allows us to share statistical power among different data points.

4.4.1 Example: Deep Latent Variable Models

The model employs a multivariate normal prior from which we draw a latent variable z from

$$p(z) = \mathcal{N}(\mu_z, \Sigma_z)$$

even though this could be an arbitrary prior. The likelihood of the model is

$$p_\theta(x|z) = \prod_{i=1}^{N} \mathcal{N}\left(\mu(z_i), \sigma^2(z_i)\right)$$

where $\mu(\cdot)$ and $\sigma^2(\cdot)$ are two nonlinear functions, usually neural nets. In this case, θ refers to the parameters of the network.

Density Estimation: Variational Autoencoders

In the machine learning literature, variational autoencoders (VAE) refer to the application of amortized variational inference for making inference in deep latent Gaussian models. It is, perhaps, the simplest setup that realizes deep probabilistic modeling.

VAEs employ two deep sets of neural networks: a top-down generative model as described above, mapping from the latent variables z to the data x, and a bottom-up inference model which approximates the posterior $p(z|x)$. Commonly, the corresponding neural networks are referred to as the *generative network* and the *recognition network*, or sometimes as *decoder* and *encoder* networks.

In order to approximate the posterior, VAEs employ an amortized mean-field variational distribution:

$$q_\phi(z|x) = \prod_{i=1}^{N} q_\phi(z_i|x_i)$$

Notice that if we were not making use of amortization, we would introduce variational parameters $\{\phi_i\}$ for each data point x_i. Via amortization rather than introducing variational parameters, we instead learn a function that maps each x_i to an appropriate ϕ_i. Since we need this function to be flexible, we parameterize it as a neural network. Therefore, the conditioning on x_i indicates that the local variational parameters associated with each data point are replaced by a function of the data. This amortized variational distribution is typically chosen as:

$$q_\phi(z_i|x_i) = \mathcal{N}\left(\mu_z(x_i), \sigma_z^2(x_i)I\right)$$

Similar to the generative model, the variational distribution employs nonlinear mappings μ_z and σ_z^2 of the data in order to predict the approximate posterior distribution. The parameter ϕ summarizes the corresponding neural network parameters.

Given this setup of the variational inference problem, to perform stochastic optimization we apply the reparametrization trick outlined in section (§4.2).

This setup was proposed in two independent seminal paper Kingma and Welling (2013) and Rezende et al. (2014). The process is the following.

For each data point, we sample S times $\varepsilon_{i,s}$ from a noise distribution $p(\varepsilon)$. We reparametrize the latent variable as $z_i = \mu_z(x_i) + \sigma(x_i)\varepsilon_{i,s}$, with μ_z and σ_z parametrized by ϕ. Therefore, we would obtain an S-dimensional vector of samples of the same z_i where each component is denoted $z_{i,s}$. We can approximate the per-data point ELBO

$$\text{ELBO}(x_i, \theta, \phi) = \log p_\theta(x_i|z_i) - \text{KL}[q_\phi(z_i|x_i)\|p(z_i)]$$

using the following Monte Carlo estimate

$$\text{ELBO}(x_i, \theta, \phi) \approx \frac{1}{S}\sum_{s=1}^{S} \log p_\theta(x_i|z_{i,s}) - \text{KL}[q_\phi(z_i|x_i)\|p(z_i)]$$

This stochastic estimate of the ELBO can subsequently be differentiated with respect to θ and ϕ to obtain an estimate of the gradient. Notice that the KL divergence, in this setup, can be integrated analytically being computed between two Gaussian distributions.

Regression: Conditional Variational Autoencoders

The conditional variational autoencoder (CVAE) is a conditional directed graphical model used for regression tasks. Given an input x and an output y, the aim is to create a model $p(y|x)$ which maximizes the probability of the observed data. Unlike VAE, there are three types of variables involved: input variables x, output variables y, and latent variables z. The setup is the following

$$\begin{aligned} z &\overset{i.i.d.}{\sim} p(z) \\ y_i|x_i, z &\overset{ind}{\sim} p(y_i|x_i, z) \end{aligned}$$

The outputs are assumed normally distributed, as well as the latent variable. The location parameter of the output distribution is a deterministic function m of x and z that we can learn from data and, in particular, it is a neural network.

Hence, the generative model is

$$p(z, y_{1:N}|x_{1:N}) = p(z)\prod_{i=1}^{N}\mathcal{N}\left(m(x_i, z), \sigma^2\right)$$

Performing Bayesian inference, our aim is to compute the posterior over the latent variables $p(z|x, y)$ as well as the conditional marginal distribution $p_\theta(y|x)$. In doing so we perform variational inference. Therefore, we define a variational distribution, $q_\phi(z)$, and we minimize

$$\mathrm{KL}[q_\phi(z)\|p(z|x, y)]$$

As above, it is possible to deduce the ELBO

$$\mathrm{ELBO} = \log p(y|x, z) - \mathrm{KL}[q_\phi(z)\|p(z|x)]$$

where we assume $p(z|x) = p(z)$, that is z is independent of x when y is unknown. To optimize this objective we use the reparametrization trick, expressing $z = \mu + \sigma\varepsilon$, for $\varepsilon \sim p(\varepsilon)$. Furthermore, we use amortization, that is

$$q_\phi(z_i|x_i, y_i) = \mathcal{N}\left(\mu_z(x_i, y_i), \sigma_z^2(x_i, y_i)I\right)$$

For each data point, we sample S times $\varepsilon_{i,s}$ from a noise distribution $p(\varepsilon)$. Once we have S z_i's for each data point we can approximate the per-data point ELBO using the following Monte Carlo estimate

$$\mathrm{ELBO}(x_i, y_i) \approx \frac{1}{S}\sum_{s=1}^{S}\log p_\theta(y_i|x_i, z_{i,s}) - \mathrm{KL}[q_\phi(z_i|x_i, y_i)\|p(z_i|x_i)]$$

This stochastic estimate of the ELBO can subsequently be differentiated with respect to θ and ϕ to obtain an estimate of the gradient.

Once we have the approximate posterior, we can compute the predictive marginal distribution of a new observation y^* given a new input x^*

$$p(y^*|x^*) = \int p(y^*|x^*, z)\ q(z|x, y)\ dz$$

Chapter 5

Meta-Learning and Neural Processes

A key aspect of intelligence is versatility – the capability of doing many different things. Current AI systems excel at mastering a single skill, but when asked to do a variety of seemingly simple activities they struggle. In contrast, a human can act and adapt intelligently to a wide variety of new, unseen situations. How can we enable our artificial agents to acquire such versatility? There are several techniques being developed to solve these sorts of problems. In this chapter we will review meta-leaning, framing it as a statistical learning problem in hierarchical models via the application of empirical Bayes methods and – when this first solution is not viable – variational inference.

Learning a new skill by observing another individual, the ability to imitate, is a key part of intelligence in humans and animals. Can we enable a robot to do the same, that is, learning to manipulate a new object by simply watching a human manipulating the object few times? Such a capability would make it dramatically easier for us to communicate new goals to robots – we could simply show robots what we want them to do. Many researchers in the past have investigated how well a robot can learn from an expert. However, this kind of vision-based skills usually required a huge number of demonstrations of an expert performing the action.

To enable a robot to imitate skills performed in video by a human, for example, we can allow it to incorporate prior experience, rather than learn each skill completely from scratch. By incorporating prior experience, the robot should also be able to quickly learn to manipulate new objects while being invariant to differing conditions, such as the person providing the demonstration, the varying background scene, or different viewpoints. The methods presented in this chapter aim to achieve both of these abilities, namely *few-shots* learning – i.e. imitation based on few examples – and *domain invariance*, by "learning to learn" from demonstration data: while learning a specific task, the robot is capable of learning a sort of attitude towards leaning in general. This field of research is commonly labelled **meta-learning** in the machine learning community. We will prove that this kind of statistical learning is not new. We bridge this new evolving and exciting field, which is exploiting deep learning techniques, with well-known statistical models. Meta-learning is, and will be, the key to equip robots with the ability to imitate by observing a human.

In this chapter we explore the statistical underpinnings of meta-learning and we present a specific instance of the class of probabilistic models employed in this area: the **Neural Processes**. They have been recently (July 2018) introduced as an extension of a "pure" machine learning model called Generative Query Networks (Eslami et al. 2018) – a computer vision system which predicts how a 3D scene looks from any viewpoint after just a few 2D views from other viewpoints. In our view, there is still room for other statistical interpretations, even if, in the original paper, they have already been framed as stochastic process "approximators". By the time of writing, the two original papers (Garnelo et al. 2018b; Garnelo et al. 2018a) have been complemented with minor follow-up unpublished documents that start to explore their peculiarities, such as the various effects of different loss functions and their analogy with Gaussian Processes with deep kernels – a work still in its infancy, released on the internet in January 2019 (unpublished), but surely a promising step in the same direction of this thesis.

Furthermore, the authors have released the Python code which implements Conditional Neural Processes (the deterministic counterpart of the fully-fledged NPs) using the `TensorFlow` library. With this thesis we also contribute a Python package, written in `PyTorch` which we called `NeuralProcesses`, that implements NPs. The aim of this package is to be user-friendly and flexible.

The code is presented in the Appendix, while its use is show-cased in the next chapter.

Neural processes can jointly consider different conditions and enable efficient generalization to new scenarios. This class of models is based on Gaussian Processes (GPs) augmented with additional latent variables and deep neural nets. The model is able to represent the variance related to different tasks in the latent space. This allows the model to efficiently and robustly generalize to a new task. An efficient Bayesian inference scheme, developed by deriving an amortized variational lower bound, and a particular training regime render this model more scalable than Gaussian Processes. especially in high-dimensional problems.

5.1 Meta-Learning

Current AI systems can master a complex skill from scratch, using an understandably large amount of time and experience. But if we want our artificial agents to be able to acquire many skills and adapt to many environments, we cannot afford to train each skill in each setting from scratch. In fact, deep learning models, unfortunately, usually need millions of labeled samples to be trained. Instead, we need our agents to learn how to learn new tasks faster by reusing previous experience, rather than considering each new task in isolation. This approach of learning to learn, or meta-learning, is a key stepping stone towards versatile agents that can continually learn a wide variety of tasks throughout their lifetimes. Early approaches to meta-learning date back to the late 1980s and early 1990s (Schmidhuber 1987; Bengio and Bengio 1990). Recently meta-learning has become a hot topic with a myriad of research papers being published each year (Finn et al. 2017; Yoon et al. 2018; Grant et al. 2018; Finn et al. 2018).

In machine learning, meta-learning is formulated as "[...] the extraction of domain-general information that can act as an inductive bias to improve learning efficiency in novel tasks" (Grant et al. 2018). Meta-learning attempts to endow machine learning models with the ability to learn from small data leveraging past experience by training a meta-learner to perform well on a *set* of training tasks. The meta-learner is then applied to an unseen task, assumed

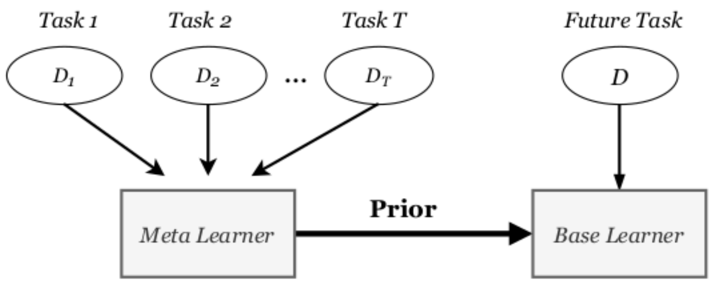

Figure 5.1: *The meta-learner uses the datasets of the observed tasks $D_1, \ldots, D_T$ to infer prior knowledge which in turn can facilitate learning in future tasks.*

to be similar to the one used for training, with the hope that it can learn to solve the new task efficiently. Therefore, for problems in which data are insufficient, meta-learning is a good solution if there are multiple related tasks.

In particular, in meta-learning (or learning-to-learn or inductive transfer) a *meta-learner* extracts knowledge from several observed tasks to facilitate the learning of new tasks by a *base-learner*, see fig. (5.1). In this setup the meta-learner must generalize from a finite set of observed tasks. The performance is evaluated when learning related new tasks which are unavailable to the meta-learner. In other words, meta-learning is a way to *learn a prior*. The goal is, thus, to train a model on a variety of learning tasks, such that it can solve new learning tasks using only a small number of training samples – also called *few-shots learning* (Li et al. 2006; Lake et al. 2017). To summarize, meta-learning allows an intelligent agent to leverage prior learning episodes as a basis for quickly improving performance on a novel task.

Bayesian hierarchical modeling provides a theoretical framework for formalizing meta-learning as inference for a set of parameters and latent variables that are shared across tasks.

Formally, a set of T tasks is considered, and within each task a dataset of size N_t is observed, that is $\{(x_{it}, y_{it})\}_{i=1}^{N_t}, t = 1, \ldots, T$. The observed data are divided into training and test set, as usual. However, in the case of meta-learning these sets are not fixed, they can change and be recomposed multiple times. Thus, a different nomenclature is often used in this setting. The data from which the algorithm learns are called *context set*, $\mathcal{C}$, while the data on which the algorithm is tested are called are defined *target set*, $\mathcal{T}$. The division between context and target set, as we will assess below, is not rigid; on the contrary, for each task it can change at each training iteration.

Meta-learning systems are trained by being exposed to a large number of tasks and are then tested in their ability to learn new tasks. This differs from many standard machine learning techniques, which involve training on a single task and testing on held-out examples from that specific task. There are two optimization procedures involved: the learner, which learns new tasks, and the meta-learner, which trains the learner.

The main approaches to meta-learning fall into one of the following five categories: point-estimate probabilistic meta-learning, amortized MAP inference, gradient-based meta-learning, conditional probability modelling, metric-based meta-learning (Gordon et al. 2019). Neural Processes lie between conditional probability modeling and gradient-based meta-learning. In the following section, we formalize meta-learning as an inference problem in hierarchical Bayesian models. In section (§5.3) we introduce Neural Processes formally.

5.2 A Probabilistic Interpretation

Meta-learning can be formalized, from a statistical point of view, as hierarchical Bayesian modelling. We introduce this section with a practical example which highlights a common use-case of hierarchical Bayesian modelling. This lets us lay down the notation and gives us the possibility to introduce statistical concepts that will be used in the following sections to interpret Neural Processes probabilistically.

Probabilistic models

We often collect data as a combination of multiple experiments, or "scenarios" or "tasks" – in the machine learning parlance. This different scenarios can be, for example, images taken from different models of cameras. We only have some labels to identify these scenarios, or tasks, in our data, e.g. we can have the specifications of the used cameras. These labels themselves do not represent the full information about these scenarios. Therefore, we could wonder how to use them in a supervised learning task. A common practice in this case would be to ignore the differences among scenarios, but this will result in low accuracy of modeling because all the variations related to the different scenarios are considered as observation noise, as different scenarios are not distinguishable anymore in the inputs. Alternatively, we can either model each scenario separately, which often suffers from too small training data, especially if we want to train a deep model. In both of these cases, generalization to new scenarios (or tasks) is not possible: we need some sort of dependence among tasks to be able to generalize. Let's consider the following example inspired by Dai et al. (2017):

> *Consider a situation in which we wish to model the braking distance of a car in a completely data-driven way, that is, assuming no knowledge about physics. We can treat it as a nonparametric regression problem, where the input is the initial speed and the output is the distance from the location where the car starts to brake to the point where the car is fully stopped. We know that the braking distance depends on the friction coefficient, which varies according to the conditions of tyres and road, which in turn depend on many other conditions.*

To measure the friction coefficient we can conduct experiments with a set of different tyres and road conditions, each associated with a condition identifier. We can think of something like ten different conditions, each has five experiments with different initial speeds. How can we model the relation between the speed and braking distance in a data-driven way, so that we can extrapolate to a new condition with only one experiment? To formalize the problem we denote the speed to be x_t and the observed braking distance to be y_t and the condition identifier to be t – for "task". We use y and X to denote all the outputs and all the inputs, respectively, without distinguishing the various tasks.

One modeling choice is to ignore the difference in conditions assuming a single regression function, m, linking inputs and outputs. Since we do not know the shape of the function, we consider the entire space of regression functions, $\mathcal{M}$, and we define a distribution, F, over this space. Then, the relation between the speed and distance can be modeled simply as

$$\begin{aligned} y &= m(X) + \varepsilon \\ m &\sim F \end{aligned}$$

where ε represents measurement noise and the function m can be modeled, for example, as a Gaussian Process, that is we can impose a GP prior in place of F. As noted above, the drawback of this modeling choice is that the variations caused by different conditions are modeled as measurement noise and thus accuracy results very low.

Alternatively, we can model each condition separately, that is, we assume T task-specific functions m_t. Hence,

$$\begin{aligned} y_t &= m_t(x_t) + \varepsilon_t \\ m_t &\sim F \qquad t = 1, \dots, T \end{aligned}$$

where T denotes the number of considered conditions, or tasks. In this case, the relation between speed and distance for each condition can be modeled only if there are sufficient data in that condition. If for each condition we only have few data, estimating m_t can be problematic. Even if this is possible, the model is not able to generalize to new conditions because it does not consider the correlations among conditions.

Our goal is to model the relation between inputs and outputs in each condition in a way that satisfies the following two requirements:

- To maximize predictive performance on each task
- To leverage shared statistical structure between tasks

In other words, being as much task-specific as possible while getting around the obstacle of having few data points for the specific condition introducing dependence across the various tasks. One way to fulfill these requirements is to assume the existence of a fixed known function g – that we will assume unknown later – shared across tasks. In addition, we assume that the task-specific information is contained in a vector of latent variables, z_t, for $t = 1, \dots, T$, generated by an unknown distribution. Assuming that the set $\{z_t\}_{t=1}^{T}$

is an exchangeable set, therefore *conditionally independent* rather than i.i.d., is consistent with the assumption that the experiments are similar in a broad sense, so they share many common elements.

The standard way to generate exchangeable random variables parametrically (Ghosh et al. 2006) is to introduce a vector of hyperparameters η and assume that z_t's are i.i.d. conditionally on η, where η parameterizes the probability density function defined on z_t. Therefore, the model can be written as[1]

$$\begin{aligned} y_{it}|x_{it}, z_t, g &\overset{ind}{\sim} p(\cdot|x_{it}, z_t, g) \\ z_t|\eta &\overset{i.i.d.}{\sim} p(\cdot|\eta) \\ \eta &\sim p(\eta) \end{aligned} \tag{5.1}$$

The problem is cast as a Bayesian hierarchical regression problem. In this way all tasks are modeled jointly letting the correlation among different conditions to be correctly captured. This allows for generalization to new conditions (or tasks).

The joint probability of the outputs and the task specific latent variables, for T tasks, given the inputs and the hyperparameter η, is given by

$$p\left(y_{1:T}, z_{1:T}|x_{1:T}, \eta, g\right) = \prod_{t=1}^{T} p\left(z_t|\eta\right) \prod_{i=1}^{N_t} p\left(y_{it}|x_{it}, z_t, g\right)$$

This is often referred to as the *generative* model. We have, thus, cast the meta-learning setting as a hierarchical model.

Furthermore, we can assume g fixed, but unknown. We can assume that for each task there are different optimal parameters value, that is let θ_t be the parameters of g for task t. As we did for the latent variable z, we can *borrow* statistical across task introducing dependence among the task specific θ_t's. We assume that $\{\theta_t\}_{t=1}^{T}$ is an exchangeable set, i.e. given a vector of hyperparameters α, the θ_t's are conditionally independent. The resulting model

[1]Even if X and g are assumed known, we condition on them as well for clarity.

can be, thus. written as

$$
\begin{aligned}
y_{it}|x_{it}, z_t, \theta &\overset{ind}{\sim} p(\cdot|x_{it}, z_t, \theta) \\
z_t|\eta &\overset{i.i.d.}{\sim} p(\cdot|\eta) \\
\theta_t|\alpha &\overset{i.i.d.}{\sim} p(\cdot|\alpha) \\
\eta &\sim p(\eta) \\
\alpha &\sim p(\alpha)
\end{aligned}
\tag{5.2}
$$

We will see that in the case of NPs, θ represents the vector of parameters parametrizing a neural network.

Probabilistic Inference

A fully Bayesian treatment of the models defined above is usually computationally involved and in practice approximation methods are employed. Inference methods used by machine learning practitioners in meta-learning problems can be reconciled with Empirical Bayes techniques (Grant et al. 2018; Finn et al. 2018).

Empirical Bayes

The term empirical Bayes is used in various contexts (Petrone et al. 2014):

- Compound experiments, where a latent distribution driving experiment-specific parameters formally acts as a prior on each one such parameter and is estimated from the data, usually by maximum likelihood
- Bayesian inference when hyperparameters of a subjective prior distribution are selected from data
- Bayesian inference when nuisance parameters are selected in a data-driven way

Inference for meta-learning can be straightforwardly connected with empirical Bayes applications in compound experiments, that is with “classical” empirical Bayes methods (Ghosh et al. 2006).

The introduction of empirical Bayes methods is commonly associated with Robbins' article (Robbins 1956) on compound sampling problems. In such settings, T values $z_1, \dots, z_T$ are drawn at random from a latent distribution G. Then, conditionally on $z_1, \dots, z_T$, observable random variables $y_1, \dots, y_T$, where $y_t = (y_{1t}, \dots, y_{N_t t})$ for each t, are drawn independently from probability distributions $p(\cdot|z_1), \dots, p(\cdot|z_T)$, respectively. Therefore, the framework can be described as

$$\begin{aligned} &y_t|z_t \overset{ind}{\sim} p(\cdot|z_t) \\ &z_t|G \overset{i.i.d.}{\sim} G(\cdot), \qquad t = 1, \dots, T \end{aligned}$$

where the index t refers to the t-th experiment, or task. The goal is to estimate the experiment-specific latent variables z_t using all observations $y_1, \dots, y_T$. In a fully Bayesian framework, the latent distribution G formally would plays the role of a prior distribution on z_t. When G is known, inference on z_t is carried out through the Bayes' rule, computing the posterior distribution of z_t given y_t. In general G is unknown and the posterior distribution is not computable directly. In the Bayesian approach one would introduce probabilistic dependence across experiments by regarding $(z_1, \dots, z_T)$ as exchangeable and assigning a hyperprior probability law to G, interpreted as a formalization of subjective information, and then performing Bayesian inference.

However, we can use an estimate of G based on all the available observations $y_1, \dots, y_T$. This fact originated the term "empirical Bayes". In the empirical Bayes approach to compound problems, although G formally acts as a prior distribution on a single parameter, its introduction is motivated in a frequentist sense: simply as the common distribution of the random sample $(z_1, \dots, z_T)$. In fact, G is estimated by frequentist methods: maximum likelihood, usually. The empirical Bayes estimate of z_t makes efficient use of the available information because all data are used when estimating G. In other terms, empirical Bayes techniques involve learning from the experience gathered from similar experiments, that is "borrowing strength" (Petrone et al. 2014).

To summarize, in a broad sense, the term empirical Bayes is used to denote a data-driven selection of prior hyperparameters in Bayesian inference where the prior distribution is interpreted in terms of subjective probability on an unknown, but fixed, parameter.

Considering the problem formalized in eq. (5.2), we can apply empirical Bayes techniques in the following way. To share statistical strength among tasks, the set of latent variables z is assumed to be generated by a common distribution $p(z|\eta)$ whose parameters are shared among the various latent variables: this is the key to borrow strength. From a fully Bayesian point of view we would have assumed exchangeability, which is reasonable since the tasks are related to each other in some ways, introducing a vector of hyperparameters η and assuming conditional independence of the z_t's given η. Inference is performed computing the posterior

$$p(z|y,X) = \int \prod_{t=1}^{T} p(z_t|y_t, x_t, \eta)\; p(\eta|y,X)\; d\eta$$

If the number of tasks is large, we can invoke the theorem on posterior normality. Thus, $p(\eta|y,X)$ is nearly Gaussian with mean $\hat{\eta}_{MLE}$ computed maximizing the conditional marginal likelihood across tasks $\prod_{t=1}^{T} p(y_t|x_t,\eta)$, and variance of order $O(1/T)$ (Ghosh et al. 2006). Thus, the posterior distribution of η is approximately degenerate at $\hat{\eta}_{MLE}$. This implies that for a specific task

$$\begin{aligned} p(z_t|y,X) &= \int p(z_t|y_t, x_t, \eta)\; p(\eta|y,X)\; d\eta \\ &\approx p(z_t|y_t, x_t, \hat{\eta}) \end{aligned}$$

Many meta-learning models are estimated using empirical Bayes methods, even if the application of such methods is not made explicit, being hidden below specific training regimes or inference algorithms. In particular, for a large class of meta-learning models called Model-Agnostic Meta-Learning, or MAML, a one-to-one relationship between gradient-based learning (that is the optimization procedures analysed in Chapter 3) and empirical Bayesian inference in hierarchical models such as eq. (5.2) has been formulated (Finn et al. 2018; Gordon et al. 2019).

However, for Neural Processes this bridge with empirical Bayes methods is not feasible. NPs can be cast has hierarchical models similar to eq. (5.2). Unfortunately, the presence of multiple neural nets prevents the derivation of this one-to-one connection with empirical Bayes methods. In principle the connection still holds: the parameters θ of the function g, that in the case of NPs is a deep neural net, are estimated by maximum likelihood, and for the latent variables, z, a distributional estimate is provided. However, inference is performed using variational inference methods.

5.3 Neural Processes

Neural Processes have been introduced by the two recent papers Garnelo et al. (2018b) and Garnelo et al. (2018a), and discussed in some unpublished material (Kim et al. 2019). The authors successfully manage to blend two apparently distant worlds: deep neural nets and Gaussian processes.

A neural network is a parameterised function that can be tuned via gradient descent to approximate a labelled collection of data with high precision. A Gaussian process (GPs), on the other hand, defines a distribution over possible functions, and is updated when new data are available via the rules of probabilistic inference. GPs are probabilistic, data-efficient and flexible, however they are also computationally intensive and thus limited in their applicability.

Neural Processes (NPs) combines the best of both worlds. Like GPs, NPs define distributions over functions, are capable of rapid adaptation to new observations, and can estimate the uncertainty in their predictions. Like neural nets, NPs are computationally efficient during training and evaluation but also learn to adapt their priors to data. Crucially, NPs generate predictions in a computationally efficient way: prediction with a trained NP corresponds to a forward pass in a deep neural net which is a cheap operation with modern tools and computer hardware.

Regression tasks are usually cast as modelling the distribution of a vector-valued output y given a vector-valued input x via a deterministic function, such as a neural network, taking x as an input. In this setting, the model is trained on a single dataset of input-output pairs, and predictions of the outputs are independent of each other given the inputs. An alternative approach to regression involves using the training data to compute a distribution over functions that map inputs to outputs, and using draws from that distribution to make predictions on test inputs. This approach allows for reasoning about multiple functions consistent with the data, and can capture the variability in outputs

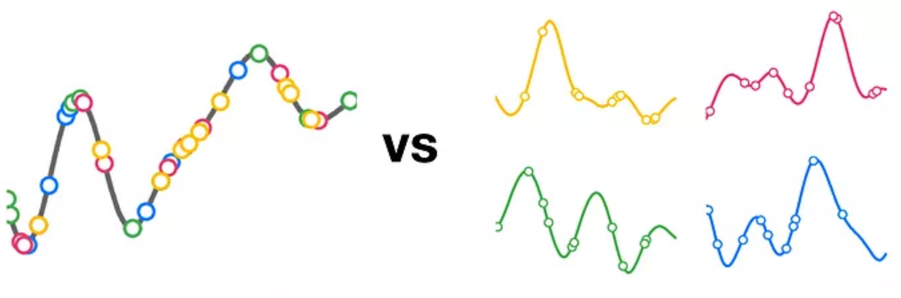

Figure 5.2: *Training regimes of Neural Processes (right) and Gaussian Processes (left).*

given inputs. In the Bayesian machine learning literature, non-parametric models such as Gaussian Processes are popular choices of this approach.

In the context of Neural Processes, the training data are usually called *context set* and the test data *target set*. Also here we employ this nomenclature. In particular, consider a set of T tasks is observed, and within each task a dataset of size N_t is observed, and each of these datasets is divided into a context set and a target set. Neural Processes offer an efficient method to modelling a distribution over regression functions. Once trained, they can predict the distribution of an arbitrary target output conditioned on a set of context input-output pairs of an arbitrary size. This flexibility of NPs enables them to model data that can be interpreted as being generated from a stochastic process. It is important to note however that NPs and GPs have different training regimes. NPs are trained on samples from multiple realisations of a stochastic process (i.e. trained on many different functions), whereas GPs are usually trained on observations from one realisation of the stochastic process (a single function) – a graphical depiction is given in fig. 5.2. Hence a direct comparison between the two is usually not plausible.

Neural processes find suitable application in supervised problems involving compound experiments that we would like to model as *probabilistically dependent*. In statistical terminology, this entails *sharing* or *borrowing statistical strength* among multiple experiments that, as noted in the introduction to the chapter, in machine the learning parlance is referred to as meta-learning. Neural Processes were introduced formalizing the compound experiments problem as a problem of estimating the law governing the stochastic process $\{Y_x, x \in \mathcal{X}\}$.

Each experiment, or task, is considered a sample from *one* realization, or trajectory, of the underlying stochastic process (Gikhman and Skorokhod 1969). The distribution over datasets, therefore, allows us to learn the law governing the stochastic process. We label this approach, the **stochastic process view**. We, on the other hand, conceptualize neural processes as a specific class of hierarchical Bayesian models for compound experiments estimated using empirical Bayes and variational inference. We label this approach **hierarchical model view**.

5.3.1 Stochastic Process View

A stochastic process (Lamperti 1977) f is typically defined as a collection of random variables, on a probability space (Ω, Σ, P). The random variables, indexed by some set $\mathcal{X}$, all take values in the same mathematical space $\mathcal{Y}$. In other words, given a probability space (Ω, Σ, P), a stochastic process can be simply written as $\{Y_x\}_{x \in \mathcal{X}}$ or $\{f(x) : x \in \mathcal{X}\}$, defining $Y_x = f(x)$. For any point $\omega \in \Omega$, $f(\cdot, \omega)$ is a sample function mapping index space $\mathcal{X}$ to space $\mathcal{Y}$. These functions are called the trajectories of the stochastic process.

Furthermore, when $\mathcal{X}$ is a singleton, the process $\{Y_x\}_{x \in \mathcal{X}}$ is just a single random variable. When $\mathcal{X}$ is finite (e.g., $\mathcal{X} = \{1, 2, \ldots, n\}$), we get a random vector. Therefore, stochastic processes are generalizations of random vectors. In contrast to the case of random vectors or random variables, it is not easy to define a probability density (or a probability mass function) for a stochastic process. Loosely speaking, the main issue is that we have to deal with infinity. One usually considers a family of finite-dimensional distributions, i.e., the joint distributions of random vectors $(Y_{x_1}, \ldots, Y_{x_n})$ for all $n \in \mathbb{N}$ and all choices $x_1, \ldots, x_n \in \mathcal{X}$ and requires that these finite-dimensional distribution are consistent with each other.

For any finite index set $x_{1:n} = \{x_1, \ldots, x_n\}$, we can define the finite-dimensional marginal joint distribution over function values $\{f(x_1), \ldots, f(x_n)\}$. For example, Gaussian Processes have marginal distributions as multivariate Gaussians. Kolmogorov Extension Theorem (Øksendal 2003) shows that a stochastic process can be characterized by marginals over all finite index sets.

In viewing Neural Processes as stochastic processes approximators helps us in interpreting the role of the multiple datasets in a multi-task context. We assume that the different datasets are particular realizations, or trajectories, of the true underlying stochastic process. In building NPs, the authors describe a random function using a latent random variable z to parameterize a *deterministic* function g, as in section (§5.2). In this way they can model different realizations of the data generating stochastic process: each sample of z corresponds to one realization of the stochastic process. In this sense z is a *global* latent variable: once sampled, an entire function is defined, not just one data point. In other words, given x, $g(x, z)$ is random since z is random: the randomness in g is conveniently moved to z.

A Neural Process, therefore, describes a stochastic process. It learns to approximate a stochastic process exploiting the information derived from multiple datasets, interpreted as a single instantiation, or trajectory, of the stochastic process.

Neural Processes as Implicit Processes

We can reframe this stochastic approximation view, provided in the original papers, by looking at the problem from a Bayesian nonparametric point of view, exploiting the definition of *implicit stochastic process* developed in a recent paper by Ma et al. (2018).

The concept of implicit processes, and its distinction from prescribed models, was firstly introduced by Diggle and Gratton (1984). A *prescribed* statistical model is a parametric specification of the distribution of a random vector, whilst an *implicit* statistical model is one defined in terms of a generating stochastic mechanism. In their paper, the authors developed methods of inference for implicit statistical models whose distribution is intractable. Following their traces, recently Ma et al. (2018) developed an inference scheme for possibly very complicated data-generating processes such as deep neural nets. Their paper explores applications of implicit models to Bayesian nonparametric regression, which brings together the best of both worlds: elegance of inference and the well-calibrated uncertainty of Bayesian nonparametric methods, with the strong representational power of implicit methods. They justify the introduction of implicit processes generalizing the definition of Gaussian processes. To grasp the idea behind implicit processes it is instructive to follow their reasoning.

Like GPs, implicit stochastic processes (IPs) assign implicit distributions over any finite collections of random variables. GPs defines the distribution of a random function f by placing a multivariate Gaussian distribution $\mathcal{N}(m(\cdot), k(\cdot))$ over any finite collection of function values $f = (f(x_1), \ldots, f(x_N))$ evaluated at any given finite collection of input locations $\{x_n\}_{n=1}^N$. Following Kolmogorov consistency theorem (Itô 1984), the mean and covariance functions are shared across all such finite collections. Alternatively the construction of GPs can be defined via the following sampling process as

$$f = Bz + m, \quad \text{where } z \sim \mathcal{N}(0, I), \text{ with } K = BB^T$$

the Cholesky decomposition of the covariance matrix. Observing this, the authors propose a generalization of the generative process by replacing the linear transform of the latent variable z with a nonlinear one. The formal definition of implicit stochastic process is the following:

> **Definition (noiseless implicit processes).** *An implicit stochastic process is a collection of random variables $f(\cdot)$, such that any finite collection $f = (f(x_1), \ldots, f(x_N))$ has a joint distribution implicitly defined by the following generative process*
>
> $$z \sim p(z), \quad f(x_n) = g_\theta(x_n, z), \ \forall x_n \in X$$
>
> *A function distributed according to the above IP is denoted as*
>
> $$f(\cdot) \sim \mathcal{IP}\left(g_\theta(\cdot, \cdot), p_z\right)$$

Neural processes are thus a specific instance of the class of implicit processes where $p(z) = \mathcal{N}(0, I)$, and $g_\theta(\cdot, \cdot) = \mathrm{NN}_\theta(\cdot, \cdot)$ is a neural net. However, inference is conducted in the latent variable z space using the variational auto-encoder approach (§4.4.1), with the inference network parameterized by a neural net, in a way we will explore in section (§5.3.2). By contrast, the variational IP approach applies to any implicitly defined process, and performs inference in function space.

We can return to the problem in section (§5.2). The model can be rewritten as a Bayesian regression model

$$y = m(X) + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2)$$
$$m(\cdot) \sim \mathcal{IP}\left(\mathrm{NN}_\theta(\cdot,\cdot), p_z\right), \qquad z \sim \mathcal{N}(0, I)$$

Given an observational dataset $D = \{X, y\}$, where all the observation have been grouped regardless the task to which they belong, and a set of test inputs y^*, Bayesian predictive inference over y^* involves computing the predictive distribution $p(y^*|X^*, X, y, \theta)$, which implies computing the posterior $p(m|X, y, \theta)$. Besides prediction, we may also want to learn the prior parameters θ and noise variance σ^2 by maximizing the log marginal likelihood: $\log p(y|X, \theta) = \log \int_{m_X} p(y|m_X)p(m_X|X, \theta)dm_X$, with m_X the evaluation of m on X. Unfortunately, both the prior $p(m_X|X, \theta)$ and the posterior $p(m_X|X, y, \theta)$ are intractable as the implicit process does not allow point-wise density evaluation, let alone the integration tasks. Therefore, to address these tasks, we must resort to approximate inference methods. In the context of NPs this approximation is performed using variational inference methods, implementing the same approach adopted for variational autoencoders.

Hierarchical Model View

We derive structure of the Neural Process from that of a Bayesian hierarchical model. Following the reasoning presented in a very recent paper by Lacoste et al. (2019) we then show that the Neural Process structure can be derived by a full hierarchical model. Consider the generative model of the NPs

$$p(y, z|X) = p(z) \prod_{i=1}^{N} \mathcal{N}\left(g_\theta(x_i, z), \sigma^2\right)$$

We start our analysis with the goal of learning a prior $p(w|\eta)$, over the weights w of a generic neural networks NN_w, across multiple tasks parameterized by η. We use a hierarchical Bayesian approach across T tasks, with hyper-prior $p(\eta)$. Each task has its own parameters w_t, with $w = \{w_t\}_{t=1}^{T}$. We can, thus, express the generative process as follows

$$\begin{aligned} y_{it}|x_{it}, w_t &\overset{ind}{\sim} p(y_{it}|x_{it}, w_t) \\ w_t|\eta &\overset{i.i.d.}{\sim} p(w_t|\eta) \\ \eta &\sim p(\eta) \end{aligned}$$

Using all datasets $D = \{D_t\}_{t=1}^{T}$, we have the following posterior distribution

$$p(w, \eta|D) \propto p(D|w)\ p(w|\eta)\ p(\eta) = \prod_t \prod_i p(y_{it}|x_{it}, w_t)\ p(w_t|\eta)\ p(\eta)$$

Following the reasoning developed in section (§5.2), for the posterior distribution, $p(\eta|D)$, we assume that the large amount of data available across multiple tasks will be enough to overcome a generic prior $p(\eta)$. Hence, we consider a point estimate of the posterior distribution $p(\eta|D)$ computed via maximum likelihood methods.

Focusing on $p(w_t|\eta)$, since w_t is potentially high dimensional with intricate correlations among the different dimensions, we cannot use a simple Gaussian distribution. As for VAE (Kingma and Welling 2013) we can use an auxiliary variable $z \sim \mathcal{N}(0, I)$ and a deterministic function projecting the noise z to the space of w, i.e. $w = h_\eta(z)$. Considering a point estimate for η, and the newly introduced latent variable z, the posterior distribution becomes

$$\prod_t p(w_t, z_t|\hat{\eta}, D_t) \propto \prod_t p(w_t|z_t, \hat{\eta})p(z_t) \prod_{i \text{ of task } t} p(y_{it}|x_{it}, w_t)$$

The goal is to jointly learning a function h_η common to all tasks and a posterior distribution $p(z_t|\hat{\eta}, D_t)$ for each task. The authors, in Lacoste et al. (2019), provide a method which allows to recast this inference problem into a simpler architecture in the context of multi-task learning.

They show that conditioning on η, the posterior factorizes independently for all tasks. This reduces the joint ELBO to a sum of individual ELBO for each task. Given a family of distributions $q_{\phi_t}(z_t|D_t, \eta)$, the ELBO for task t is

$$\text{ELBO}_t = \text{E}_{q_{\phi_t}} \sum_{i \text{ of task } t} \log p(y_{it}|x_{it}, h_\eta(z_t)) - \text{KL}_t$$

where the KL term is shown to simplify to $\mathrm{KL}_t = [q_{\phi_t}(z_t|D_t,\eta)\|p(z_t|\eta)]$. Notice that after simplification, the task-specific KL term is no longer over the space of w_t but only over the space z_t. This amounts to "[...]finding a low dimensional manifold in the parameter space where the different tasks can be distinguished" (Lacoste et al. 2019). Then, the posterior $p(z_t|D_t,\eta)$ only has to model which of the possible tasks are likely, given observations D_t instead of modeling the high dimensional $p(w_t|D_t,\eta)$. Since we no longer need to explicitly calculate the KL on the space of w, we can simplify the likelihood function to $p(y_{it}|x_{it},z_t,\theta)$, which can be a deep network parameterized by θ, taking both x_{it} and z_t as inputs. In the case of Neural Processes, this neural network is the decoder $g_\theta(x,z)$.

Empirical Bayes View

We can now explore how Neural Process estimation could have been approached from an Empirical Bayes point of view, even though, as said above, the implementation of such approximate method is infeasible due to the presence of neural networks.

Consider the regression problem of section (§5.2). We assumed the existence of a fixed, but unknown, regression function g and that the latent variable z encodes all the peculiar information of a specific task

$$\begin{aligned} y &= g(X,z)+\varepsilon, \qquad \varepsilon \sim p(\varepsilon) \\ z|\eta &\sim p(z|\eta) \\ \eta &\sim p(\eta) \end{aligned}$$

In section (§2.5) we referred to this kind of models as *nonparametric mean function regression*. Since g is unknown, we should impose a prior distribution on it, from a fully Bayesian perspective. However, for NPs no prior distributions are defined, and a powerful function approximator (neural nets) is used instead. In fact, the peculiarity of NPs is that g is defined to belong the class of neural networks $\{g_\theta : \theta \in \Theta\}$, whose generic elements, g_θ, are indexed by a parameter vector $\theta \in \Theta$ representing the weights and biases of the network. This modeling approach liberates the practitioner from having to specify a prior for g, e.g. a Gaussian Process, since it will be inferred from the data. The error term is assumed to be normally distributed with mean zero and diagonal covariance matrix: this guarantees that g_θ can be interpreted as the conditional mean function. Furthermore, in line with the literature about variational autoencoders, section (§4.4.1), $p(z|\eta)$ is assumed to be a multivariate Gaussian.

To define a Neural Process we can rewrite the equation above according to these assumptions

$$\begin{aligned} y|x, z, g &\sim \mathcal{N}\left(g(x,z), \sigma^2 I\right) \\ z|\eta &\sim \mathcal{N}\left(\eta_1, \eta_2\right) \\ \eta_1, \eta_2 &\sim p(\eta) \end{aligned}$$

Since, given a specific z, g is deterministic and the covariance matrix is diagonal, the outputs y's are *i.i.d.* Gaussian random variables. Given the presence of the neural network g, it is not possible to proceed further with the empirical Bayes estimation of the model and we have to rather use amortized variational inference.

In particular, instead of assigning a prior $p(\eta)$ to the hyperparameters of the prior distribution on z, a maximum likelihood estimate is used for both η_1 and η_2. They are expressed as a the output of a neural network, whose parameters are learned maximizing the log-likelihood of the observed outputs. This procedure can be equivalently justified applying amortized variational inference. The term amortized "inference" refers to utilizing inferences from past computations to support future computations. As we detailed in section (§4.4), a neural network is used to infer the optimal value, in a maximum likelihood sense, for the parameters from the data. Statistical strength is shared since the parameters of the network are the same for each task: every task contributes to pin down the best values for the parameters of the prior distribution. This links Empirical Bayes procedure to Neural Processes.

In practise, this is obtained in the following way. In fact, given N input-output pairs (across all tasks) we can write the generative model as

$$\begin{aligned} p(y, z|x) &= p(z|\hat{\eta})\, p(y|z, x) \\ &= \mathcal{N}\left(\mu_z, \sigma_z^2 I\right) \prod_{i=1}^{N} \mathcal{N}\left(g_\theta(z, x_i), \sigma^2\right) \end{aligned}$$

where η_1 and η_2 have been replaced by two neural networks: μ_z and σ_z^2. Bayesian inference for this model implies the computation of the posterior distribution $p(z|y, x)$. To approximate this posterior, as for VAE (sec. §4.4.1), we can use amortized variational inference. We define the variational distribution to be

$$q_\phi(z|x) = \mathcal{N}\left(\mu_z(x), \sigma_z^2(x) I\right)$$

as for conditional variational autoencoders, where the conditioning on x indicates that the variational parameters are replaced by functions of the data. Specifically, these functions, $\mu_z(\cdot)$ and $\sigma_z^2(\cdot)$ are neural networks with a peculiar structure that makes them invariant to the order of the inputs, and ϕ represents the parameters of the networks. Order invariance is motivated in the following sections.

To obtain an estimate of the posterior, we minimize $\mathrm{KL}[q_\phi(z|x) \| p(z|y, x)]$. Since this involves the computation of intractable distributions, as in section (§4.2), we minimize another tractable objective: the ELBO, that following the same rationale explained in that section, can be written as

$$\mathrm{ELBO}(\phi) = \mathrm{E}\big[\log p(y|z, x)\big] - \mathrm{KL}\big[q_\phi(z|x) \| p(z)\big] \tag{5.3}$$

where we know that $\log p(y|x) \geq \mathrm{ELBO}(\phi)$ from section (§3.4). Thus, maximizing the ELBO is approximately equal to maximizing the joint distribution of the outputs given the inputs.

In the following section we analyse in more detail the network involved in the estimation of the hyperparameters of the distribution of z.

5.3.2 Encoder Specification

As we defined it in section (§4.4.1), the approximating distribution q_ϕ is also called *encoder*. The authors, in building the NP (Garnelo et al. 2018b; Garnelo et al. 2018a), designed the encoder in such a way to accommodate *invariance to the order* of context points. This is important because we would like to accommodate for variable numbers of observation per task without retraining. Rapid inference comes automatically with the use of a deep neural network to amortize the approximate posterior distribution q. However, it typically comes at the cost of flexibility: amortized inference is usually limited to a single specific task. Below, we discuss design choices that enable us to retain flexibility.

Inference with sets as inputs requires particular care. The amortization networks, in the case of Neural Processes, take data sets of variable size as inputs whose ordering they should be invariant to. We use permutation-invariant instance-pooling operations, like averaging, to process these sets, as formalized in Zaheer et al. (2017). The instance-pooling operation ensures that the network can process any number of training observations.

Therefore, the neural nets μ_z and σ_z^2 have both a peculiar structure. The architecture used can be boiled down to three core components

- A neural net h, that the authors refer to as **encoder** – even if the encoder is usually used to refer to the entire approximating distribution – that maps the input space into the representation space; it takes pairs $(x_i, y_i)_{i \in C}$ in the context set and produces a representation $r_i = h(x_i, y_i)$ for $i \in C$

- An **aggregator** a, that summarizes the encoded inputs in an order-invariant global representation $r = a(r_i)_{i \in C}$; they implement the aggregator as a mean function, perhaps the simplest operation that ensures order-invariance

- Two neural nets μ and σ^2 that map the global representation to the mean and variance of the global latent variable; in practice they have the same architecture, but a transformation that ensures positive outcomes is added at the end of the σ^2 network

Therefore, $\mu_z \overset{def}{=} \mu(a(h(x_i, y_i)))$ and $\sigma_z^2 \overset{def}{=} \sigma^2(a(h(x_i, y_i)))$, for $i \in C$.

To summarize, NPs have three main desirable properties:

- *Scalability*: since they rely on amortization

- *Flexibility*: since they define a wide family of distributions and one can condition on an arbitrary number of context points to obtain an arbitraty number of targets
- *Permutation invariance*: given by the construction of the encoder they can handle input sets of arbitrary dimension

Therefore, like Gaussian Processes, NPs define distributions over functions, are capable of adaptation to new observations, and can estimate uncertainties in their predictions. Like neural networks, they are computationally efficient during both training and evaluation, and in addition learn to adapt their priors to the data.

5.3.3 A Peculiar Training Regime

NPs are designed to learn distributions over functions from distributions over datasets. Consider a set of datasets (or tasks), D. For each dataset, $D_t \in D$ we have N_t input-output pairs $\{(x_{it}, y_{it})\}_{i=1}^{N_t}$, where N_t is the size of dataset t, for $t \in \{1, \dots, T\}$. The total number of training data is $N = N_1 + \cdots + N_T$.

At each iteration of learning, one dataset $D_t = \{(x_{it}, y_{it})\}_{i=1}^{N_t}$ is chosen. A subset, referred to as *context set*, $\mathcal{C}_t = \{(x_{it}, y_{it})\}_{i=1}^{M_t}$ is randomly sampled. Symmetrically, the entire dataset is called *target set*, that is, $\mathcal{T}_t = D_t$. Usually $M_t \leq N_t$ and is also randomly chosen, that is, $M_t \sim \text{Unif}(0, N_t)$. For brevity, once a dataset is chosen amongst all datasets, let $x_\mathcal{C}, y_\mathcal{C}, x_\mathcal{T}, y_\mathcal{T}$ denote the inputs and outputs of the context and target set respectively.

The NP during training can learn from the context points only and must predict the target points, i.e. the the entire dataset (or function, or realization of the stochastic process). The ELBO in eq. (5.3) does not reflect this division into context and target set. In fact, we are requiring the NP to maximize the joint distribution of all outputs (the target set, as we have defined it) given no context – which is the standard variational lower bound. We label this kind of objective function as $\text{ELBO}_{[\mathcal{T}|\varnothing]}$.

To better reflect the training regime of the NPs, instead of maximizing the joint distribution of all outputs, i.e. the target set, given no context, we can maximize the conditional distribution of the target given the context. This lead to the objective

$$\text{ELBO}_{[\mathcal{T}|\mathcal{C}]}(\phi) = \text{E}[\log p(y_{\mathcal{T}}|z, y_{\mathcal{C}}, x_{\mathcal{C}})] - \text{KL}[q_\phi(z|y_{\mathcal{T}}, x_{\mathcal{T}}) \| p(z|y_{\mathcal{C}}, x_{\mathcal{C}})]$$

where we know that $\log p(y_{\mathcal{T}}|x_{\mathcal{T}}, x_{\mathcal{C}}, y_{\mathcal{C}}) \geq \text{ELBO}_{[\mathcal{T}|\mathcal{C}]}$. Note that $p(z|y_{\mathcal{C}}, x_{\mathcal{C}})$ is a *conditional prior* [Garnelo et al. (2018a)], that can be interpreted as a less informed posterior distribution. Furthermore, since also $p(z|y_{\mathcal{C}}, x_{\mathcal{C}})$ is intractable, an approximation is used, therefore the ELBO becomes

$$\text{ELBO}_{[\mathcal{T}|\mathcal{C}]}(\phi) = \text{E}[\log p(y_{\mathcal{T}}|z, y_{\mathcal{C}}, x_{\mathcal{C}})] - \text{KL}[q_\phi(z|y_{\mathcal{T}}, x_{\mathcal{T}}) \| q_\phi(z|y_{\mathcal{C}}, x_{\mathcal{C}})] \quad (5.4)$$

Going through this generative mechanism step-by-step:

- The context points $(x_i, y_i)_{i \in \mathcal{C}}$ are mapped through a neural network h to obtain a latent representation r_i
- The vectors $\{r_i\}_{i \in \mathcal{C}}$ are aggregated (in practice: averaged) to obtain a single value r with the same dimensionality of a single r_i
- The aggregated representation r is used to parametrise the distribution of z, i.e. $p(z|x_{\mathcal{C}}, y_{\mathcal{C}}) = \mathcal{N}(\mu_z(r), \sigma_z^2(r))$
- To obtain a prediction at a target $x_t, t \in \mathcal{T}$, we sample z and concatenate this with x_t, and map (z, x_t) through a neural network g to obtain a sample from the predictive distribution of y, i.e. $p(y_t|z, x_t)$

Training

Inference for the Neural Processes is carried out in the variational inference framework. Specifically, we introduce two approximate distributions: $q(z|context)$ to approximate the conditional prior $p(z|\text{context})$ and $q(z|\text{context}, \text{target})$ to approximate the respective $p(z|\text{context}, \text{target})$ where we have denoted context $:= \{(x_i, y_i)\}_{i \in \mathcal{C}}$ and target $:= \{(x_i, y_i)\}_{i \in \mathcal{T}}$. The approximate posterior $q(z|\cdot)$ is chosen to have the specific form as illustrated in fig. 5.3. That is, we use the same h to map both the context set as well as the target set to obtain the aggregated r, which in turn is mapped to μ_z and σ_z^2. These parameterize the approximate posterior $q(z|\cdot) = \mathcal{N}(\mu_z, \sigma_z^2)$

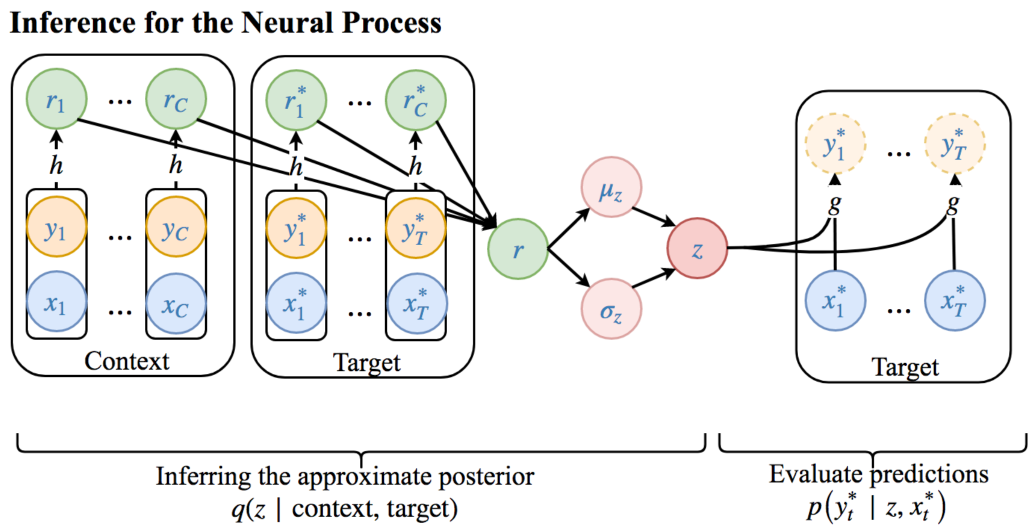

Figure 5.3: *Observed values (blue) and true data generating process (red) for the single task.*

5.3.4 Stochastic Approximation of the ELBO

Since the ELBO contains (possibly deep) neural networks, an analytic evaluation of it is hard to obtain. In such cases, as we have seen in section (§4.2), it is possible to use black-box variational inference. In particular, we use the reparametrization trick to obtain an estimate of the ELBO by Monte Carlo samples.

We reparametrize z as a transformation of a random variable $\zeta \sim \mathcal{N}(0, I)$, that is

$$z = \mu_z(x) + \sigma_z(x)\zeta$$

The expectations involved in the ELBO are, thus, computed with respect to the distribution of ζ. As for the conditional variational autoencoder, for each data point, we sample S times $\zeta_{i,s}$ from a noise distribution $p(\zeta) = \mathcal{N}(0, I)$. Once we have S instantiation of the latent variable corresponding to one dataset, $z_s = \mu_z(x) + \sigma_z(x)\zeta_{i,s}$, for each data point, we can approximate the per-data point ELBO using the following Monte Carlo estimate

$$\text{ELBO}_{[\mathcal{T}|\mathcal{C}]}(x_i, \phi, \theta) \approx \frac{1}{S}\sum_{s=1}^{S} \log p(y_i|x_i, z_s) - \text{KL}[q_\phi(z|x_i, y_i)\|p(z|x_i)]$$

As shown in section (§4.2.2), the gradient of the ELBO can be easily computed, given the new parametrization. However, note that in this case we do not have the subscript i in z. This is motivated by the fact that z in NP represents global uncertainty, that is, it is sampled once for all points in a dataset (or task), it is not sampled for each single data point.

Chapter 6
Applications

In this chapter we test Neural Processes in two different settings. First, we perform a single 1-dimensional regression task which allows for a direct comparison with Gaussian Processes. Second, we extend the 1-dimensional regression problem to a meta-learning setting which allows for a concrete test of the multi-tasking capabilities of Neural Processes in a controlled environment.

The aim of this chapter is to show-case the capabilities of Neural Processes performing two different experiments.

In the first experiment we employ a Neural Process to learn a single 1-dimensional regression function (single task). This will give us the possibility to make a direct comparison with Gaussian Processes. We have to notice that we are forcing the Neural Process to adapt to a single task (regression function), even if they were designed to deal with many tasks at once. Consequently, given the fact that the variability of the estimated regression function is learned from the variability of the various datasets describing the different tasks, we expect underestimated uncertainty of the prediction. To mimic the multi-task setting, we apply the standard training regime of Neural Processes in which the training set is divided into context and target set at each training iteration. We will assess that the Neural Process prediction is very good, even compared with that of the Gaussian Process, but the uncertainty estimates are less accurate.

In the second experiment we test Neural Processes in a meta-learning setting. We consider a set of 30 regression functions (tasks). We apply the standard training regime defined for Neural Processes. At meta-test time, that is when a new unseen task (regression function) is considered, we assume that we can only observe 10 noisy input-output pairs ("few-shots"). We perform few training iterations on these observed data, using the standard training regime of Neural Processes. As we will assess in the next sections, the Neural Process is able to capture the true data-generating process and estimate uncertainty coherently.

We will use toy data in both experiments. This allows us to perfectly control the data-generating process, to test multiple architecture during the implementation phase, and in particular to use the computational capabilities of a standard PC. The experiments are implemented in Python using the package `NeuralProcesses` created as a complementary output of this thesis. It is based on the open-source library `PyTorch`: one of the most used frameworks – besides `TensorFlow`, `MXNet` and `Caffe2` – to implement deep learning models. The source code of the package and the code for both experiments in presented in the Appendix.

6.1 Experiment 1

Description

As a first experiment, we test Neural Processes on a single classical 1D regression task. A Gaussian Process with Radial Basis Function kernel (Rasmussen and Williams 2006) is used as baseline comparison. We consider a single dataset that consists of input-output pairs generated according to the following scheme

$$y_i = \sin(2\pi x_i) + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0,\ 0.2) \tag{6.1}$$

We assume that we get to see 62 points which we split into a training and a test set. In particular, the training set $\{x_i, y_i\}_{i=1}^{11}$ is formed by 11 training input points generated as 11 equally distant points between 0 and 1, while the training outputs are sampled according to eq. (6.1). We generate the 51 test set pairs $\{(\tilde{x}_i, \tilde{y}_i)\}_{i=1}^{51}$ in the same fashion: the test inputs are 51 equally distant points between 0 and 1, while the test outputs are generated according to eq. (6.1). The training set, test set, and the noiseless data-generating process are depicted in fig. 6.1. As we noted in the introduction to the chapter, this is not a

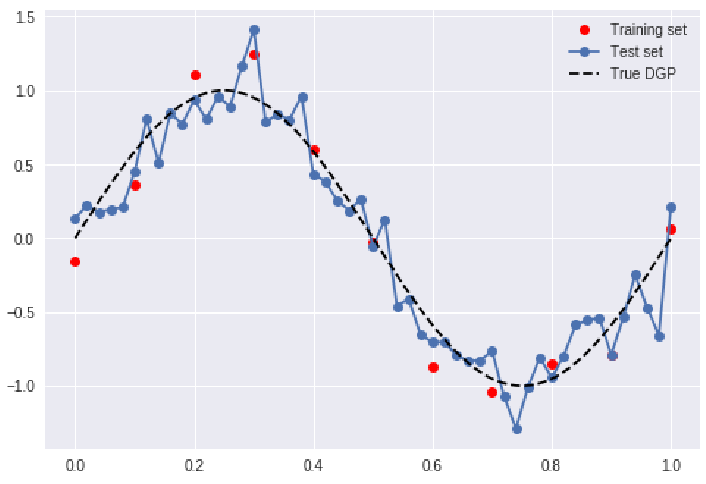

Figure 6.1: *Training set (red), test set (blue), and the true data generating process (dashed black line) for the single task.*

canonical scenario for Neural Processes, which were designed to deal with multi-tasking problems. However, we mimic that scenario in two ways. First, during training we split the training set $\{(x_i, y_i)\}_{i=1}^{11}$ into a context and target set. In particular, at each iteration we randomly choose how many and which specific input-output pairs to allocate to the context set. Second, during evaluation on the test set, we split the test set into context and target set as well. After performing few gradient steps on the test data we evaluate the Neural Process fit.

Model

As we addressed in the introduction to the chapter, this simplified setting allowed us to quickly test multiple architectures using the computational capabilities of a standard PC. Each architecture was tested multiple times using different initializations for the values of the parameters in the networks (encoder and decoder) to avoid any dependence of the results on the specific values at which the weights of the networks were initialized. We found that using deeper and more complicated architectures did not have significant impact on the quality of the prediction. On the contrary, it entailed higher computational costs and a higher risk of overfitting.

We implemented the Neural Process with the following structure:

- Dimension of the encoded representation r: 3
- Encoder h: 2-unit input layer (dimension of x plus the dimension of y), 2 hidden layers both featuring 10 hidden units with $Tanh$ activation function, and a 3-unit output layer with no activation function applied
- Dimension of the latent variable z: 3
- Mapping from r to μ_z: matrix product $\mu_z = M_{3\times 3}r$
- Mapping from r to $\log \sigma_z$: matrix product $\log \sigma_z = \text{softplus}\left(M_{3\times 3}r\right)$
- Decoder g: 4-unit input layer (dimension of z plus the dimension of x), 2 hidden layers both featuring 10 hidden units with $Tanh$ activation function, and a 2-unit output layer with no activation function applied for μ_y, while softplus activation for the log variance of y
- Optimization: the Adam optimizer with learning rate set to 0.001

The generative mechanism of the prediction is the following. The context points $(x_i, y_i)_{i\in\mathcal{C}}$ are mapped through a neural network h to obtain a latent representation r_i. The vectors $\{r_i\}_{i\in\mathcal{C}}$ are aggregated (in practice: averaged) to obtain a single value r with the same dimensionality of a single r_i. The aggregated representation r is used to parametrise the distribution of z, i.e. $q(z|x_\mathcal{C}, y_\mathcal{C}) = \mathcal{N}(\mu_z(r), \sigma_z^2(r))$. To obtain a prediction at a target x_t for $t \in \mathcal{T}$: we sample z, we concatenate it with x_t, and we map (z, x_t) through a neural network g to obtain a sample from the predictive distribution of y, i.e. $p(y_t|z, x_t) = \mathcal{N}(\mu_y(z, x_t), \sigma_y^2(z, x_t))$.

Training

Inference for the Neural Process is carried out in the variational inference framework. The specific training routine is explained step-by-step below. In fig. 6.2 a representation of how the function learns across the various iterations is provided. Specifically, we introduce two approximate distributions: $q(z|\text{context})$ to approximate the conditional prior $p(z|\text{context})$, and $q(z|\text{target})$ to approximate the respective $p(z|\text{target})$ where we have denoted context $:= \{(x_i, y_i)\}_{i\in\mathcal{C}}$ and target $:= \{(x_i, y_i)\}_{i\in\mathcal{T}}$. The approximate posterior $q(z|\cdot)$ is chosen to have the specific form illustrated in fig. 5.3, that is we use the same neural network h for both context and target set to obtain the aggregated r, which in turn is mapped to μ_z and σ_z^2. These parameterize the approximate posterior $q(z|\cdot) = \mathcal{N}(\mu_z, \sigma_z^2)$.

For each training iteration, 10^5 in our specific case,

- Select the number, `n_context`, of context points at random from a discrete uniform distribution with support $\{4, \dots, 11\}$
- Given `n_context`, select at random the actual context points from the context set $\{(x_i, y_i)\}_{i=1}^{11}$, i.e. what input-output pairs in the training set to show the algorithm at one specific training iteration
- Define the target set equal to the entire training set, that is $\mathcal{C} \subseteq \mathcal{T}$
- Compute z_{context}, that is pass only the context set through h and map it to the mean and variance of the posterior distribution of z given the context only
- Compute z_{target}, that is pass the target set through h and map it to the mean and variance of the posterior distribution of z given the target set
- Compute, analytically since they are two Gaussian distributions, the KL divergence, $\text{KL}\big[q(z|\text{target})\|q(z|\text{context})\big]$, between the posterior distribution given the target set and the posterior distribution given the context set only
- Sample 100 times z from the posterior distribution given the target set, i.e. $q(z|\text{target})$; for each of these vectors, concatenate it with the vector of target inputs, and generate the mean and variance of the predictive distribution $p(y_t|x_t, z), t \in \mathcal{T}$; given the parameter of the distribution, compute the log-likelihood of the target set
- To produce a Monte Carlo estimate of the log-likelihood, average the results obtained at the previous step
- Sum the negative log-likelihood and the KL divergence to obtain the ELBO
- Compute the gradient of the ELBO with respect to all parameters
- Set the new values for the parameters and restart the loop

Results

We will now compare the results obtained by fitting a Neural Process with those obtained by fitting a Gaussian Process that has been exposed to the same training and test set, namely 11 input-output pairs in the training set and 51

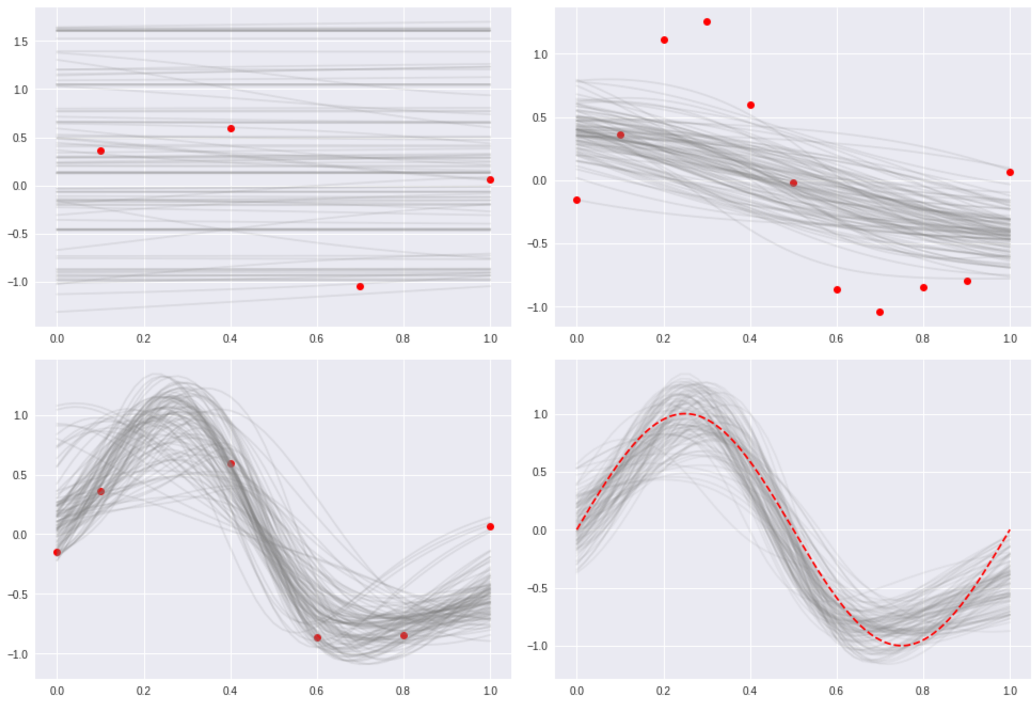

Figure 6.2: *Learned prior distribution (grey lines, 100 samples per iteration) at various iterations of the training process: 1 iteration (top left), 3000 iterations (top right), 8000 iterations (bottom left), 10000 iterations (bottom right). At each iteration a number between 4 and 11 is randomly sampled, and a context set of that size is randomly chosen from the training dataset (red dots). The last iteration shows the comparison between the learned prior distribution and the true underlying data-generating process.*

input-output pairs in the test set. Before feeding the test data to the Neural Process, the learner prior distribution over the training set is depicted in fig. 6.3. After 1000 gradient steps computed using samples from the test set and following the same procedure used for the training phase, the resulting Neural Process regression function prediction is presented in fig. 6.4 together with the posterior distribution of the Gaussian Process. As we can assess from the picture, both the Gaussian Process and the Neural Process are able to capture the true underlying data-generating process. The Gaussian Process is able to contain the true DGP within the boundaries of the confidence region. The Neural Process, on the other hand, is able to capture almost point-wise the true DGP, but – as we can assess from the picture – it is very confident about its prediction and

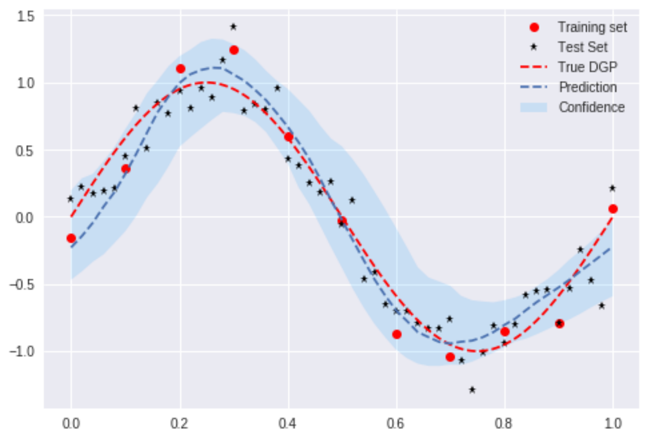

Figure 6.3: *Training set (red), test set (blue) not seen during training, the true data generating process (dashed red line) for the single task, the predicted curve (blue dashed line), and the confidence region which correspond to the $5^{th}, 95^{th}$ percentiles computed across 100 samples of predicted curve corresponding to 100 samples of z.*

even the small deviations from the true DGP are not included in the confidence bands. This effect is not due to the specific number of training iterations: even lowering their number, the result in terms of uncertainty estimated remains unchanged, while the overall prediction deteriorates. We have to consider, though, that this experiment is implemented on a single task. Even trying to reproduce the training regime of Neural Processes, it is unavoidable that the algorithms sees the same points often, thus reducing the variability it is exposed to.

On the other hand, the Gaussian Process is able to capture the uncertainty better. This is due to the strong prior information which is not subdued by the data, given the small number of training data.

Another remark regarding the Neural Process is important. Despite the high flexibility provided by the structure of the decoder and the structure of the encoder (2 deep neural nets), the regularizing action of the KL divergence term prevents the Neural Process to overfit. For example, in correspondence of the input $x = 0.3$ there are some values that would encourage the Neural Process to deviate from the true DGP. In another version of the experiment (not presented here) with a latent variable of dimension 8 and a decoder with 6 layers, 10 hidden units per layer, and $Tanh$ activation, the result changes dramatically: the Neural Process overfits the data over-confidently. This suggests that, as

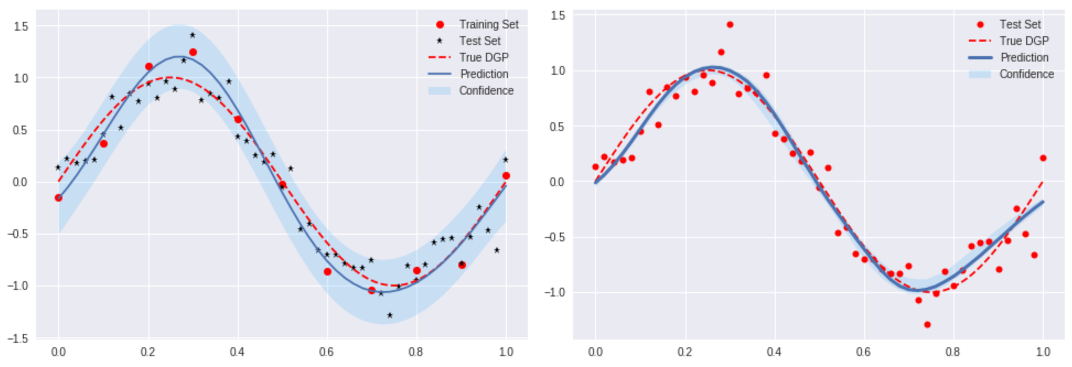

Figure 6.4:* *Left. *GP prediction (blue line), training set (red dots), test set (black stars), the true data generating process (dashed red line), and the confidence region which correspond to the* $5^{th}, 95^{th}$ *percentiles.* ***Right.*** *Training set (red), test set (blue) not seen during training, the true data-generating process (dashed red line) for the single task, the predicted curve (blue dashed line), and the confidence region which correspond to the* $5^{th}, 95^{th}$ *percentiles computed across 100 samples of predicted curve corresponding to 100 samples of* z.

for any other statistical model, parsimony is an important design principle to consider when designing the structure of Neural Processes.

6.2 Experiment 2

Description

As a second experiment, we test Neural Processes in a meta-learning/multi-task setting. The classical 1D regression task is expanded to multiple functions. In particular, 30 tasks are considered, i.e. 30 different functions generated according to the following equation

$$y_i = \sin(a\pi x_i) + \varepsilon_i, \qquad \varepsilon_i \sim \mathcal{N}(0,\ 0.2) \tag{6.2}$$

where $a \sim \text{Unif}(2,4)$. We assume that we get to observe 25 points for each function. In particular, the training set $\{(x_{it}, y_{it})\}_{i=1}^{25}, t \in \{1,\ldots,30\}$ is formed by 25 training inputs points generated as 25 equally distant points between 0 and 1, while the training outputs are sampled according to eq. (6.2). The test set, for each task, is composed by the 150 pairs $\{(\tilde{x}_i, \tilde{y}_i)\}_{i=1}^{150}$ generated in the

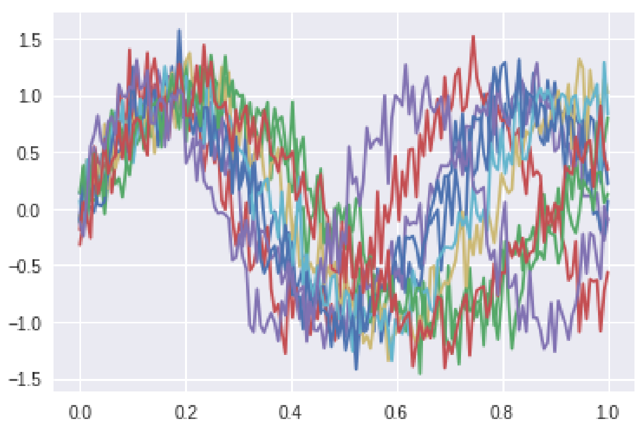

Figure 6.5: *Training set (red), test set (blue), and the true data-generating process (dashed black line) for the single task.*

same way. However, note that in this experiment we train the model on the training dataset (25 points for each task) and then we measure its predictive performances on a newly extracted task, not on the held-out part of the training set. Therefore, the test is never observed by the algorithm, it is only used to create the plot of the various tasks (fig. 6.5). This is to emulate a real multi-task situation. A sample of the tasks can be seen in fig. 6.5.

Model

As for the previous experiment each architecture was tested multiple times using different initializations for the values of the parameters in the networks (encoder and decoder) to avoid any dependence of the results on the specific values at which the weights of the networks were initialized. Again, we found that using deeper and more complicated architectures did not have significant impact on the quality of the prediction. On the contrary, it entailed higher computational costs, a higher risk of overfitting, and an underestimated uncertainty even in this multi-task setting.

For the Neural Process we adopted the following structure:

- Dimension of the encoded representation r: 3
- Encoder h: 2-unit input layer (dimension of x plus the dimension of y), 2 hidden layers both featuring 10 hidden units with $Tanh$ activation function, and a 3-unit output layer with no activation function applied

- Dimension of the latent variable z: 3
- Mapping from r to μ_z: matrix product $\mu_z = M_{3\times3}r$
- Mapping from r to $\log \sigma_z$: matrix product $\log \sigma_z = \text{softplus}\,(M_{3\times3}r)$
- Decoder g: 4-unit input layer (dimension of z plus the dimension of x), 2 hidden layers both featuring 10 hidden units with $Tanh$ activation function, and a 2-unit output layer with no activation function applied for μ_y, while softplus activation for the log variance of y
- Optimization: the Adam optimizer with learning rate set to 0.001

The generative mechanism of the prediction is equal to the one of the previous experiment.

Training

The training routine, similar to the one employed in the first experiment, is explained below. However, given the multi-task nature of this experiment, there are some minor modifications. For each training iteration, 10^5 in our specific case,

- Select randomly a task
- For that task, select the number, `n_context`, of context points at random from a discrete uniform distribution with support $\{1, \ldots, 25\}$, that is at minimum 1 observation, and at maximum the entire dataset for the specific task
- Given `n_context`, select at random the actual context points, i.e. what input-output pairs to show the algorithm at this specific training iteration
- Define the target set equal to the entire training set, that is $\mathcal{C} \subseteq \mathcal{T}$
- Compute z_{context}, that is pass only the context set through the neural net h and map its outputs to the mean and variance of the posterior distribution of z given the context only
- Compute z_{target}, that is pass the target set through the neural net h and map its outputs to the mean and variance of the posterior distribution of z given the target
- Compute, analytically since they are two Gaussian distributions, the KL divergence, $\text{KL}\big[q(z|\text{target})\|q(z|\text{context})\big]$, between the posterior distribu-

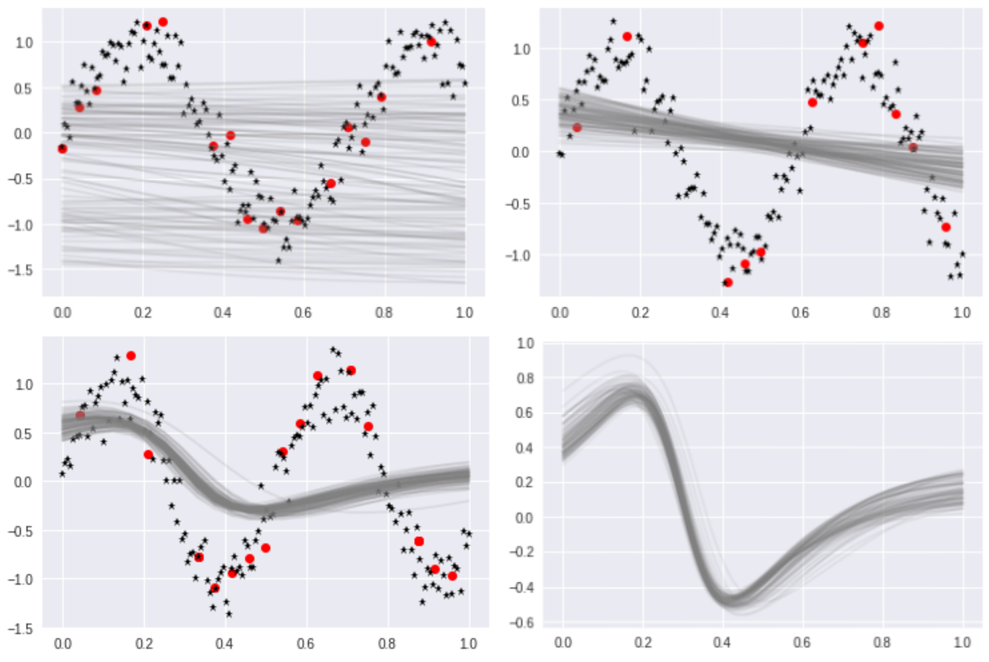

Figure 6.6: *Learned prior distribution (grey lines, 100 samples per iteration) at various iterations of the training process: 1 iteration (top left), 3000 iterations (top right), 8000 iterations (bottom left), 10000 iterations (bottom right). At each iteration a number between 4 and 11 is randomly sampled, and a context set of that size is randomly chosen from the training dataset (red dots). The last iteration shows the comparison between the learned prior distribution and the true underlying data-generating process.*

tion given the target set and the posterior distribution given the context set only

- Sample 100 times z from the posterior distribution given the target set, i.e. $q(z|\text{target})$; for each of these vectors, concatenate it with the vector of target inputs, and generate the mean and variance of the predictive distribution $p(y_t|x_t, z), t \in \mathcal{T}$; given the parameter of the distribution, compute the log-likelihood of the target set
- To produce a Monte Carlo estimate of the log-likelihood, average the results obtained at the previous step
- Sum the negative log-likelihood and the KL divergence to obtain the ELBO
- Compute the gradient of the ELBO with respect to all parameters

- Set the new values for the parameters and restart the loop

In fig. 6.6 a representation of how the function learns across the various iterations is provided.

Results

After 10^5 training iterations the Neural Process learned a prior distribution using the information coming from the various observed tasks. The learned prior distribution is depicted in fig. 6.7. In this phase of the training we do not expect the prediction given the learned prior distribution to resemble any of the 30 functions in the dataset of tasks. On the other hand, we do expect that the Neural Process has learned certain characteristics that makes it “look like” something related to the various tasks. For this specific 1D cases, it is possible to assess, qualitatively, in fig. 6.3 that it learned a sort of sinusoidal shape. Given this learned prior distribution, what we want to test is the fact that we

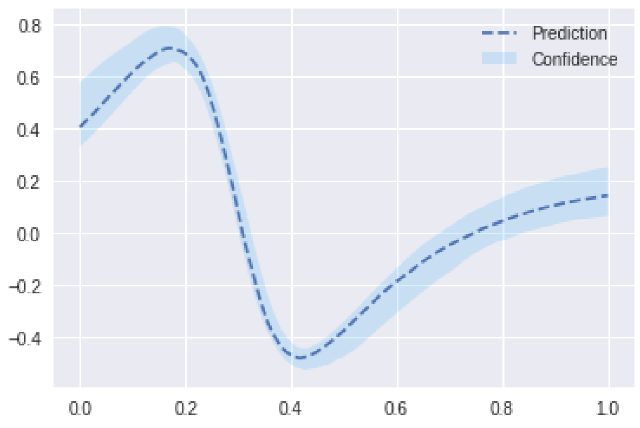

Figure 6.7: *Training set (red), test set (blue) not seen during training, the true data generating process (dashed red line) for the single task, the predicted curve (blue dashed line), and the confidence region which correspond to the $5^{th}, 95^{th}$ percentiles computed across 100 samples of predicted curve corresponding to 100 samples of z.*

are able to accurately predict a new unseen function (task) only observing few data (few-shots) from that task, that is we would like to perform few-shots learning. The new task is generated selecting a specific value for a not used to create the meta-training data, and generate a new function according to eq. (6.2).

Since we want to put ourselves in the setting in which we only have few data to learn from, we sample only 10 input-output pairs. As usual, at each training step we divide the training set into a context and a target set. At each training iteration the number of context points is randomly sampled uniformly from $\{1, \ldots, 10\}$. After the number of context points is decided, a subset of that dimension is randomly sampled from the training data. We perform only 1000 training iterations using the new data, 1/10-th of the number of iteration used in the previous meta-training process.

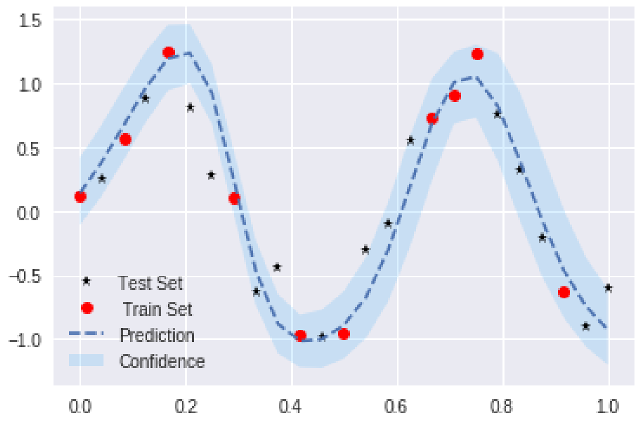

Figure 6.8:* Left.** *GP prediction (blue line), training set (red dots), test set (black stars), the true data generating process (dashed red line), and the confidence region which correspond to the* $5^{th}, 95^{th}$ *percentiles.* ***Right, *Training set (red), test set (blue) not seen during training, the true data generating process (dashed red line) for the single task, the predicted curve (blue dashed line), and the confidence region which correspond to the* $5^{th}, 95^{th}$ *percentiles computed across 100 samples of predicted curve corresponding to 100 samples of* z*.*

After 1000 gradient steps, the resulting Neural Process prediction is presented in fig. 6.8.

As we can assess from the picture, the Neural Process, quite surprisingly, is able to promptly adapt to the new task. Comparing this latest plot with fig. 6.7, it is striking how quickly it is able to adapt to a completely different shape given only 10 input-output pairs. It is able to reconstruct the denoised data-generating process without overfitting the observed data. Furthermore, the uncertainty estimates are very different from the ones resulting from the previous single-task experiment. In the meta-learning setting, the Neural Process is able to capture uncertainty in GP-like fashion. This is due to the exposure to the different datasets during meta-training. As we stated in chapter 5, the Neural Process learns the variability of its prediction from the variability of the data. Note how it correctly reduces its variance at coordinates $[0.2, 0.4]$ even if only one data point is observed in that interval. This is consistent with the prior distribution over function that it learned during meta-training: a lot of tasks with that specific shape in that particular interval have been seen during meta-training. Exploiting the information from these other tasks, the Neural Process is able to correctly estimate its uncertainty. Similarly, notice how in the interval $[0.7, 0.8]$ its uncertainty estimates are bigger even if a third of the training data are concentrated in that interval. Once again, if we go back to the plot of the tasks (fig. 6.5) we can assess how much variability is present in that interval. Exploiting information from other tasks, the NP is able to correctly measure uncertainty.

Chapter 7
Conclusions

Statistics and Machine Learning are still regarded as fairly antithetical fields of research even though they share many commonalities: one pushed forward by theoreticians, while the other mainly by practitioners. Bayesian modelling is based on the vast theory of Bayesian statistics, in which the aim is to describe the processes that has generated the data. This often results in interpretable models, but also limits the choice of the statistician to models that are still simple enough to be interpreted.

Deep Learning, on the other hand, is mostly driven by pragmatic developments – regardless the resulting interpretability of the model – that aim at creating “useful” models. However, Deep Learning still lacks a solid mathematical formalism.

With this thesis, we would like to reinforce the concept that Deep Learning can be interpreted and, thus, extended in a principled way using the mathematical formalism of statistics. Understanding the underlying principles leading to good models allows us to improve upon them. We argued that Deep learning can make use of Bayesian models.

We want to highlight that Deep Learning can make use of Bayesian models: we can combine deep learning with interpretable Bayesian models and build hybrid models that draw from the best of both worlds. We also want to point out that Bayesian models can make use of deep learning: the field of Bayesian modelling can benefit immensely from the simple data representations obtained from deep learning models, especially when dealing with complex and/or high dimensional data.

Bibliography

Bengio, Y. and Bengio, S. (1990). *Learning a Synaptic Learning Rule.* Tech. rep. (751). Montreal, Canada: Département d'Informatique et de Recherche Opérationelle, Université de Montréal.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics).* Berlin, Heidelberg: Springer-Verlag.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2017). "Variational Inference: A Review for Statisticians". *Journal of the American Statistical Association*, **112**(518), 859–877.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). "Weight Uncertainty in Neural Networks". *Proceedings of the 32nd International Conference on International Conference on Machine Learning.* **37**. ICML'15. Journal of Machine Learning Research, 1613–1622.

Braun, J. and Griebel, M. (2009). "On a Constructive Proof of Kolmogorov's Superposition Theorem". *Constructive Approximation*, **30**(3), 653.

Breiman, L., Friedman, J., Stone, C., and Olshen, R. (1984). *Classification and Regression Trees.* The Wadsworth and Brooks-Cole statistics-probability series. Monterey, CA: Taylor & Francis.

Breiman, L. (2001). "Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author)". *Statistical Science*, **16**(3), 199–231.

Bryant, D. W. (2008). "Analysis of Kolmogorov's superposition theorem and its implementation in applications with low and high dimensional data". PhD thesis. University of Central Florida.

Cauchy, A. (1847). “Methode genénérale pour la résolution des systèmes d’ equations simultanées”. *Comptes Rendus Hebd. Seances Acad. Sci.* **10**(383), 399–402.

Dai, Z., Álvarez, M., and Lawrence, N. (2017). “Efficient Modeling of Latent Information in Supervised Learning using Gaussian Processes”. *Advances in Neural Information Processing Systems.* **30**. Curran Associates, Inc., 5131–5139.

Damianou, A. and Lawrence, N. (2013). “Deep Gaussian Processes”. *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics.* Ed. by C. M. Carvalho and P. Ravikumar. **31**. Proceedings of Machine Learning Research. Scottsdale, Arizona, USA, 207–215.

Deng, L. and Yu, D. (2014). “Deep Learning: Methods and Applications”. *Found. Trends Signal Process.* **7**, 197–387.

Denison, D. G. T. (2002). *Bayesian Methods for Nonlinear Classification and Regression.* Chichester, England: Wiley.

Diggle, P. J. and Gratton, R. J. (1984). “Monte Carlo Methods of Inference for Implicit Statistical Models”. *Journal of the Royal Statistical Society. Series B (Methodological)*, **46**(2), 193–227.

Duchi, J., Hazan, E., and Singer, Y. (2011). “Adaptive subgradient methods for online learning and stochastic optimization”. *Journal of Machine Learning Research*, **12**, 2121–2159.

Eslami, S. M. A., Jimenez Rezende, D., Besse, F., Viola, F., Morcos, A. S., Garnelo, M., Ruderman, A., Rusu, A. A., Danihelka, I., Gregor, K., Reichert, D. P., Buesing, L., Weber, T., Vinyals, O., Rosenbaum, D., Rabinowitz, N., King, H., Hillier, C., Botvinick, M., Wierstra, D., Kavukcuoglu, K., and Hassabis, D. (2018). “Neural scene representation and rendering”. *Science*, **360**(6394), 1204–1210.

Finn, C., Abbeel, P., and Levine, S. (2017). “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks”. *Proceedings of the 34th International Conference on Machine Learning.* Ed. by D. Precup and Y. W. Teh. **70**. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia, 1126–1135.

Finn, C., Xu, K., and Levine, S. (2018). “Probabilistic Model-Agnostic Meta-Learning”. *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 9516–9527.

Gal, Y. (2016). “Uncertainty in Deep Learning”. PhD thesis. University of Cambridge.

Gal, Y. and Ghahramani, Z. (2016). “Dropout As a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”. *Proceedings of the 33rd International Conference on International Conference on Machine Learning*. **48**. ICML’16. New York, NY, USA: Journal of Machine Learning Research, 1050–1059.

Garnelo, M., Rosenbaum, D., Maddison, C., Ramalho, T., Saxton, D., Shanahan, M., Teh, Y. W., Rezende, D., and Eslami, S. M. A. (2018a). “Conditional Neural Processes”. *Proceedings of the 35th International Conference on Machine Learning*. Ed. by J. Dy and A. Krause. **80**. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden, 1704–1713.

Garnelo, M., Schwarz, J., Rosenbaum, D., Viola, F., Rezende, D. J., Eslami, S. M. A., and Teh, Y. W. (2018b). “Neural Processes”. *ICML Workshop on Theoretical Foundations and Applications of Deep Generative Models*.

Gentle, J. (2002). *Elements of Computational Statistics*. Ed. 1. Statistics and Computing. Springer Publishing Company, Incorporated.

Ghosh, J. K., Delampady, M., and Samanta, T. (2006). *An Introduction to Bayesian Analysis: Theory and Methods*. Ed. 1. Springer Series in Statistics. Springer New York Inc.

Gikhman, I. I. and Skorokhod, A. V. (1969). *Introduction to the Theory of Random Processes*. Philadelphia: W.B. Saunders Co.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Gordon, J., Bronskill, J., Bauer, M., Nowozin, S., and Turner, R. E. (2019). "Meta-Learning Probabilistic Inference For Prediction". *International Conference on Learning Representations.*

Grant, E., Finn, C., Levine, S., Darrell, T., and Griffiths, T. L. (2018). "Recasting Gradient-Based Meta-Learning as Hierarchical Bayes." *International Conference on Learning Representations.*

Hastie, T. and Tibshirani, R. (1986). "Generalized Additive Models". *Statistical Science*, **1**(3), 297–310.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction.* Ed. 2. Springer Series in Statistics. New York, NY, USA: Springer New York Inc.

Hecht-Nielsen, R. (1987). "Kolmogorov's mapping neural network existence theorem". *Proceedings of the International Conference on Neural Networks III*, 11–14.

Hinton, G. (2012). "Neural networks for machine learning". *Coursera video lecture*, 2121–2159.

Hinton, G. and Salakhutdinov, R. R. (2006). "Reducing the dimensionality of data with neural networks". *Science*, **313**(5786), 504–507.

Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. (2013). "Stochastic variational inference". *Journal of Machine Learning Research*, **14**, 1303–1347.

Itô, K. (1984). *An Introduction to Probability Theory.* Cambridge University Press.

Jacobs, R. (2008). "Increased rates of convergence through learning rate adaptation". *Neural Networks*, **1**, 295–307.

Jordan, M. I. and Bishop, C. M. (1996). "Neural Networks". *Handbook of Computer Science.*

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999). "An Introduction to Variational Methods for Graphical Models". *Mach. Learn.* **37**(2), 183–233.

Kim, H., Mnih, A., Schwarz, J., Garnelo, M., Eslami, S. M. A., Rosenbaum, D., Vinyals, O., and Teh, Y. W. (2019). “Attentive Neural Processes.” *International Conference on Learning Representations.*

Kingma, D. P. and Ba, J. (2015). “Adam: A Method for Stochastic Optimization.” *Proceedings of the International Conference on Learning Representations.* ICLR 2015.

Kingma, D. P. and Welling, M. (2013). *Auto-Encoding Variational Bayes.* cite arxiv:1312.6114.

Kolmogorov, A. N. (1957). “On the representation of continuous functions of many variables by superpositions of continuous functions of one variable and addition”. *Doklay Akademii Nauk USSR*, **14**(5), 953 –956.

Kulis, B. and Jordan, M. I. (2012). “Revisiting k-means: New Algorithms via Bayesian Nonparametrics”. *ICML.* icml.cc / Omnipress.

Kullback, S. and Leibler, R. A. (1951). “On Information and Sufficiency”. *Ann. Math. Statist.* **22**(1), 79–86.

Lacoste, A., Oreshkin, B. N., Chung, W., Boquet, T., Rostamzadeh, N., and Krueger, D. (2019). “Uncertainty in Multitask Transfer Learning.” *Computing Research Repository (CoRR).* abs/1806.07528.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2017). “Building Machines That Learn and Think Like People.” *The Behavioral and brain sciences*, **40**.

Lamperti, J. (1977). *Stochastic processes : a survey of the mathematical theory.* Springer-Verlag New York.

LeCun, Y., Bengio, Y., and Hinton, G. E. (2015). “Deep learning”. *Nature*, **521**(7553), 436–444.

Lee, H. (2004). *Bayesian Nonparametrics Via Neural Networks.* ASA-SIAM Series on Statistics and Applied Probability. Society for Industrial and Applied Mathematics.

Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., and Sohl-Dickstein, J. (2018). “Deep Neural Networks as Gaussian Processes.” *Pro-*

ceedings of the International Conference on Learning Representations. ICLR 2018.

Leni, P. E., Fougerolle, Y., and Truchetet, F. (2011). “Kolmogorov Superposition Theorem and its application to multivariate function decompositions and image representation”. *IEEE*. Ed. by I. C. Society.

Li, F.-F., Fergus, R., and Perona, P. (2006). “One-Shot Learning of Object Categories.” *IEEE Trans. Pattern Anal. Mach. Intell.* **28**(4), 594–611.

Loader, C. (2006). *Local Regression and Likelihood.* Statistics and Computing. Springer New York.

Ma, C., Li, Y., and Hernández-Lobato, J. M. (2018). “Variational Implicit Processes”. *ICML Workshop on Bayesian Deep Learning.*

MacEachern, S. N. (1999). “Dependent Nonparametric Processes”. *ASA Proceedings of the Section on Bayesian Statistical Science*. American Statistical Association, 50–55.

MacKay, D. J. C. (1992). “Bayesian Interpolation”. *Neural Computation*, **4**(3), 415–447.

Mitchell, T. M. (1997). *Machine Learning.* Ed. 1. New York, NY, USA: McGraw-Hill, Inc.

Mnih, A. and Gregor, K. (2014). “Neural Variational Inference and Learning in Belief Networks”. *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML’14. Beijing, China.

Muller, P. and Quintana, F. A. (2004). “Nonparametric Bayesian data analysis”. *Statistical Science*, **19**, 95–110.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks.* Berlin, Heidelberg: Springer-Verlag.

Øksendal, B. (2003). *Stochastic Differential Equations: An Introduction with Applications*. Hochschultext / Universitext. Berlin, Heidelberg: Springer-Verlag.

Paisley, J. W., Blei, D. M., and Jordan, M. I. (2012). “Variational Bayesian Inference with Stochastic Search”. *ICML*. icml.cc / Omnipress.

Petrone, S., Rizzelli, S., Rousseau, J., and Scricciolo, C. (2014). “Empirical Bayes methods in classical and Bayesian inference”. *METRON*, **72**(2), 201–215.

Polson, N. G. and Sokolov, V. (2017). “Deep Learning: A Bayesian Perspective”. *Bayesian Analysis*, **12**(4), 1275–1304.

Qian, N. (1999). “On the momentum term in gradient descent learning algorithms.” *Neural Networks*, **12**(1), 145–151.

Ranganath, R., Gerrish, S., and Blei, D. (2014). “Black Box Variational Inference”. *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*. Ed. by S. Kaski and J. Corander. **33**. Proceedings of Machine Learning Research. Reykjavik, Iceland, 814–822.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). “Stochastic Backpropagation and Approximate Inference in Deep Generative Models”. *Proceedings of the 31st International Conference on Machine Learning*. Ed. by E. P. Xing and T. Jebara. **32**. Proceedings of Machine Learning Research (2). Bejing, China, 1278–1286.

Ritchie, D., Horsfall, P., and Goodman, N. D. (2016). “Deep Amortized Inference for Probabilistic Programs”. *Computing Research Repository (CoRR)*. abs/1610.05735.

Robbins, H. (1956). “An Empirical Bayes Approach to Statistics”. *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*. Berkeley, Calif.: University of California Press, 157–163.

Robert, C. P. and Casella, G. (2005). *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Berlin, Heidelberg: Springer-Verlag.

Rosenblatt, F. (1958). “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. *Psychological Review*, 65–386.

Ruder, S. (2016). “An overview of gradient descent optimization algorithms”. *Computing Research Repository (CoRR)*. abs/1609.04747.

Salas, A., Zohren, S., and Roberts, S. (2018). “Practical Bayesian Learning of Neural Networks via Adaptive Subgradient Methods.” *Computing Research Repository (CoRR)*. abs/1811.03679.

Schmidhuber, J. (1987). “Evolutionary Principles in Self-Referential Learning”. Diploma Thesis. Technische Universitat Munchen, Germany.

— (2015). “Deep Learning in Neural Networks: An overview”. *Neural Networks*, **61**, 85–117.

Schulman, J., Heess, N., Weber, T., and Abbeel, P. (2015). “Gradient Estimation Using Stochastic Computation Graphs.” *NIPS*, 3528–3536.

Silverman, B. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis.

Sprecher, D. A. (1965). “On the structure of continuous functions of several variables”. **115**(3). Ed. by A. M. Soc, 340–355.

— (1972). “A survey of solved and unsolved problems on superpositions of functions”. *Journal of Approximation Theory*, **6**(2), 123–134.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. *J. Mach. Learn. Res.* **15**(1), 1929–1958.

Sutskever, I., Martens, J., Dahl, G. E., and Hinton, G. E. (2013). “On the importance of initialization and momentum in deep learning.” *ICML (3)*. **28**. JMLR Workshop and Conference Proceedings. Journal of Machine Learning Research, 1139–1147.

Vidakovic, B. (2009). *Statistical Modeling by Wavelets*. Wiley Series in Probability and Statistics. Wiley.

Welling, M. and Teh, Y. W. (2011). “Bayesian Learning via Stochastic Gradient Langevin Dynamics”. *ICML*. Ed. by L. Getoor and T. Scheffer. Omnipress, 681–688.

Yoon, J., Kim, T., Dia, O., Kim, S., Bengio, Y., and Ahn, S. (2018). “Bayesian Model-Agnostic Meta-Learning”. *NeurIPS*, 7343–7353.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Póczos, B., Salakhutdinov, R. R., and Smola, A. J. (2017). “Deep Sets”. *NIPS*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, 3394–3404.

Zhang, C., Bütepage, J., Kjellström, H., and Mandt, S. (2018). “Advances in Variational Inference”. *IEEE transactions on pattern analysis and machine intelligence.*

Appendix A

Code

A.1 Neural Process implementation

```
import torch
import numpy as np

###-------- ENCODER h() and a()
class Encoder(torch.nn.Module):
    def __init__(self, input_dim, encoder_specs,
                init_func=torch.nn.init.normal_):
        """
        input_dim: dimension of x_i + dimension of y_i,
        for i in Context
        """
        super().__init__()
        self.input_dim = input_dim
        self.encoder_specs = encoder_specs
        self.init_func = init_func
        for i in range(len(self.encoder_specs)):
            if i == 0:
                self.add_module('h_layer' + str(i),
                                torch.nn.Linear(self.input_dim,
                                self.encoder_specs[i][0]))
                if self.encoder_specs[i][1]:
                    self.add_module('h_layer' + str(i) + '_act',
                                    self.encoder_specs[i][1])
            else:
                self.add_module('h_layer' + str(i),
                        torch.nn.Linear(self.encoder_specs[i-1][0],
                        self.encoder_specs[i][0]))
                if self.encoder_specs[i][1]:
                    self.add_module('h_layer' + str(i) + '_act',
                                    self.encoder_specs[i][1])
        if init_func:
            for layer_name ,_ in self._modules.items():
                if layer_name.endswith('act') == False:
                    init_func(getattr(getattr(self, layer_name),
                                         'weight'))

    def forward(self, x, y):
        r_i = torch.cat([x, y], dim=1)
        for layer_name, layer_func in self._modules.items():
            if layer_name.startswith('h'):
                r_i = layer_func(r_i)
        r = r_i.mean(dim=0)
        return r



###-------- From r to z_mean, z_logvar
class Zparams(torch.nn.Module):
    def __init__(self, r_dim, z_dim):
        super().__init__()
        self.r_dim = r_dim
        self.z_dim = z_dim
        self.r_to_mean = torch.nn.Linear(self.r_dim, self.z_dim)
        self.r_to_logvar = torch.nn.Linear(self.r_dim, self.z_dim)
        self.softplus = torch.nn.Softplus()

    def forward(self, r):
        z_mean = self.r_to_mean(r).unsqueeze(-1)
        z_logvar = self.softplus(self.r_to_logvar(r)).unsqueeze(-1)
        z_std = torch.exp(0.5 * z_logvar)
        return z_mean, z_std



###-------- DECODER g()
class Decoder(torch.nn.Module):
    def __init__(self, input_dim, decoder_specs,
                init_func=torch.nn.init.normal_):
        """
        input_dim: number of features + dimesion of z
        """
        super().__init__()
        self.input_dim = input_dim
        self.decoder_specs = decoder_specs
        for i in range(len(self.decoder_specs)):
            if i == 0:
                self.add_module('g_layer' + str(i),
                                torch.nn.Linear(self.input_dim,
                                self.decoder_specs[i][0]))
                if self.decoder_specs[i][1]:
                    self.add_module('g_layer' + str(i) + '_act',
                                        self.decoder_specs[i][1])
            else:
                self.add_module('g_layer' + str(i),
                        torch.nn.Linear(self.decoder_specs[i-1][0],
                        self.decoder_specs[i][0]))
                if self.decoder_specs[i][1]:
                    self.add_module('g_layer' + str(i) + '_act',
                    self.decoder_specs[i][1])
        if init_func:
            for layer_name,_ in self._modules.items():
                if layer_name.endswith('act') == False:
                    init_func(getattr(getattr(self, layer_name),
                                      'weight'))
        self.softplus = torch.nn.Softplus()

    def forward(self, x, z):
        z_reshape =  z.t().unsqueeze(1).expand(-1, x.shape[0], -1)
        x_reshape = x.unsqueeze(0).expand(z_reshape.shape[0],
                                         x.shape[0], x.shape[1])
        x_concat_z = torch.cat([x_reshape, z_reshape], dim=2)
        y_mean = x_concat_z
        for layer_name, layer_func in self._modules.items():
            if layer_name.startswith('g'):
                y_mean = layer_func(y_mean)

        y_logvar = x_concat_z
        for layer_name, layer_func in self._modules.items():
            if layer_name.startswith('g'):
                y_logvar = layer_func(y_logvar)
        y_logvar = self.softplus(y_logvar)
        y_std = torch.exp(0.5 * y_logvar)
        return y_mean, y_std



###-------- UTILS

# Generate samples from z
def sample_z(z_mean, z_std, how_many, device=None):
    """
    Returns a sample from z of size (z_dim, how_many)
    """
    z_dim = z_std.shape[0]
    if device:
        eps = torch.randn([z_dim, how_many]).to(device)
    else:
        eps = torch.randn([z_dim, how_many])
    z_samples = z_mean + z_std * eps
    return z_samples
```

```
# Log-likelihood
def MC_loglikelihood(inputs, outputs, decoder, z_mean, z_std,
                     how_many, device=None):
    """
    Returns a Monte Carlo estimate of the log-likelihood
    z_mean: mean of the distribution from which to sample z
    z_std: std of the distribution from which to sample z
    how_many: number of monte carlo samples
    decoder: the decoder to be used to produce estimates of mean
    """
    # sample z
    if device:
        z_samples = sample_z(z_mean, z_std, how_many,
                             device=device)
    else:
        z_samples = sample_z(z_mean, z_std, how_many)
    # produce the 10 estimated of the mean and std
    y_mean, y_std = decoder(inputs, z_samples)
    # define likelihood for each value of z
    likelihood = torch.distributions.Normal(y_mean, y_std)
        # for each value of z:
            # evaluate log-likelihood for each data point
            # sum these per-data point log-likelihoods
        # compute the mean
    log_likelihood = likelihood.log_prob(outputs).sum(dim=1).mean()
    return log_likelihood



# KL divergence
def KL_div(mean_1, std_1, mean_2, std_2):
    """Analytical KLD between 2 Gaussians."""
    KL = (torch.log(std_2) - torch.log(std_1) + \
        (std_1**2/ (2*std_2**2)) + \
        ((mean_1 - mean_2)**2 / (2*std_2**2)) - 1).sum()*0.5
    return KL



def predict(inputs, decoder, z_mean, z_std, how_many,
            numpy=True, device=None):
    """
    Generates prediction from the NP
    inputs: inputs to the NP
    decoder: the specific decoder employed
    z_mean: the mean of the latent variable distribution
    z_std: the mean of the latent variable distribution
    how_many: the number of functions to predict
    numpy: convert torch tensor to numpy array
    """
    if device:
        z = sample_z(z_mean, z_std, how_many, device=device)
        y_pred, _ = decoder(inputs, z)
        if numpy:
            return y_pred.cpu().detach().numpy()
        else:
            return y_pred
    else:
        z = sample_z(z_mean, z_std, how_many)
        y_pred, _ = decoder(inputs, z)
        if numpy:
            return y_pred.detach().numpy()
        else:
            return y_pred
```

A.2 Code for Experiment 1

```
!pip install gpytorch

import torch
import numpy as np
import gpytorch
from matplotlib import pyplot as plt
import neural_process as nep

%matplotlib inline



###----- Generate Data
torch.manual_seed(19051994)

# Training data is 11 points in [0,1]
# inclusive regularly spaced
test_x = torch.linspace(0, 1, 51)
test_y = torch.sin(test_x * (2 * np.pi)) +
            torch.randn(test_x.size()) * 0.2

truth_x = torch.linspace(0, 1, 150)
truth_y = torch.sin(truth_x * (2 * np.pi))

train_x = torch.linspace(0, 1, 11)
train_y = torch.sin(train_x * (2 * np.pi)) +
            torch.randn(train_x.size()) * 0.2

plt.style.use('seaborn')
plt.plot(train_x.numpy(), train_y.numpy(), 'ro')
plt.plot(test_x.numpy(), test_y.numpy(), '-o')
plt.plot(truth_x.numpy(), truth_y.numpy(), 'k--')
plt.legend(['Training set', 'Test set', 'True DGP'])
plt.show()



###----- GP Definition
# We will use the simplest form of GP model, exact
#inference
class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(train_x,
                                           train_y,
                                           likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
                    gpytorch.kernels.RBFKernel())

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(
                                            mean_x, covar_x)

# initialize likelihood and model
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ExactGPModel(train_x, train_y, likelihood)



###---- GP Training
model.train()
likelihood.train()

# Use the adam optimizer
optimizer = torch.optim.Adam(params=list(model.parameters()),
                                lr=0.1)

# "Loss" for GPs - the marginal log likelihood
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood,
                                                model)

training_iter = 50
for i in range(training_iter):

    # Zero gradients from previous iteration
    optimizer.zero_grad()

    # Output from model
    output = model(train_x)

    # Calc loss and backprop gradients
    loss = -mll(output, train_y)
    loss.backward()
    optimizer.step()

# Get into evaluation (predictive posterior) mode
model.eval()
likelihood.eval()

# Make predictions by feeding model through likelihood
with torch.no_grad(), gpytorch.settings.fast_pred_var():
    observed_pred = likelihood(model(test_x))

with torch.no_grad():

    # Get upper and lower confidence bounds
    lower, upper = observed_pred.confidence_region()

    # Plot training data as black stars
    plt.plot(train_x.numpy(), train_y.numpy(), 'ro')
    plt.plot(test_x.numpy(), test_y.numpy(), 'k*')
    plt.plot(truth_x.numpy(), truth_y.numpy(), 'r--')

    # Plot predictive means as blue line
    plt.plot(test_x.numpy(), observed_pred.mean.numpy())

    # Shade between the lower and upper confidence bounds
    plt.fill_between(test_x.numpy(), lower.numpy(),
                upper.numpy(), alpha=0.15, facecolor='#089FFF')
    plt.legend(['Training Set', 'Test Set', 'True DGP',
                    'Prediction', 'Confidence'])



###----- Handle data
test_x = test_x.unsqueeze(-1)
test_y = test_y.unsqueeze(-1)
train_x = train_x.unsqueeze(-1)
train_y = train_y.unsqueeze(-1)



###----- NP Definition
x_dim = 1
y_dim = 1
r_dim = 3
z_dim = 3
```

```
encoder_specs = [(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 (10, torch.nn.Tanh()),
                 (r_dim, None)]

decoder_specs = [(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 (10, torch.nn.Tanh()),
                 (y_dim, None)]

encoder_input_dim = x_dim + y_dim
decoder_input_dim = x_dim + z_dim

h = nep.Encoder(encoder_input_dim, encoder_specs,
                init_func=torch.nn.init.kaiming_normal_)
r_to_z = nep.Zparams(r_dim, z_dim)
g = nep.Decoder(decoder_input_dim, decoder_specs,
                init_func=torch.nn.init.kaiming_normal_)
optimizer = torch.optim.Adam(params=list(g.parameters()) +
                             list(h.parameters()) +
                             list(r_to_z.parameters()), lr=1e-3)



###----- NP Training
epochs = 10000
kl = []
elbo = []

for epoch in range(epochs):
    optimizer.zero_grad()

    # select number of context points randomly
    n_context = np.random.randint(4, 12)
    context_size.append(n_context)

    # select ‘n_context‘ points and create the
    # context set and target set
    context_indeces = np.random.choice(train_x.shape[0],
                                       n_context, replace=False)
    x_c = train_x[context_indeces]
    y_c = train_y[context_indeces]
    x_t = train_x
    y_t = train_y

    # variational parameters (mean, std) of
    # approximate prior
    z_mean_c, z_std_c = r_to_z(h(x_c, y_c))
    # variational parameters (mean, std) of
    # approximate posterior
    z_mean_t, z_std_t = r_to_z(h(x_t, y_t))

    log_likelihood = nep.MC_loglikelihood(x_t, y_t,
                                g, z_mean_t, z_std_t, 10)
    KL = nep.KL_div(z_mean_t, z_std_t, z_mean_c, z_std_c)
    ELBO = - log_likelihood + KL
    kl.append(KL)
    elbo.append(ELBO)
    ELBO.backward()
    optimizer.step()

    if epoch % 1000 == 0:
        print('Epoch:', epoch)
        y_pred = nep.predict(test_x, g, z_mean_t, z_std_t, 100)

        # Plot of sample functions
        plt.plot(x_c.numpy(), y_c.numpy(), 'ro')
        for i in y_pred:
            plt.plot(test_x.numpy(), i, 'grey', alpha=0.15)
        plt.show()
print('Training successful')



# DIAGNOSTICS
plt.subplot(2,1,1)
plt.plot(elbo)
plt.title('Diagnostics')
plt.ylabel('ELBO')

plt.subplot(2, 1, 2)
plt.plot(kl)
plt.ylabel('KL divergence')
plt.xlabel('epochs')
plt.show()



# PLOTS
y_pred = nep.predict(test_x, g, z_mean_t, z_std_t, 100)
quantile_05, median, quantile_95 = np.percentile(y_pred,
                                        [5, 50, 95], axis=0)

# Plot of sample functions
for i in y_pred:
    plt.plot(test_x.numpy(), i, 'grey', alpha=0.1)
plt.plot(truth_x.numpy(), truth_y.numpy(), 'r--')
plt.show()

# Plot of distribution over functions
plt.plot(train_x.numpy(), train_y.numpy(), 'ro')
plt.plot(test_x.numpy().flatten(), test_y.numpy().flatten(), 'k*')
plt.plot(truth_x.numpy(), truth_y.numpy(), 'r--')
plt.plot(test_x.numpy().flatten(), median.flatten(), '--')
plt.fill_between(test_x.numpy().flatten(), quantile_05.flatten(),
                    quantile_95.flatten(), alpha=0.15,
                    facecolor='#089FFF')
plt.legend(['Training set', 'Test Set', 'True DGP',
                        'Prediction', 'Confidence'])
plt.show()



###----- NP Testing
epochs = 1000

for epoch in range(epochs):
    optimizer.zero_grad()

    # select number of context points randomly
    n_context = np.random.randint(1, 12)

    # select `n_context` points and create the context
    # set and target set
    context_indeces = np.random.choice(test_x.shape[0],
                                        n_context, replace=False)
    x_c = test_x[context_indeces]
    y_c = test_y[context_indeces]
    x_t = test_x
    y_t = test_y

    # variational parameters (mean, std) of approximate prior
    z_mean_c, z_std_c = r_to_z(h(x_c, y_c))

    # variational parameters (mean, std) of approximate posterior
    z_mean_t, z_std_t = r_to_z(h(x_t, y_t))

    log_likelihood = nep.MC_loglikelihood(x_t, y_t, g,
                                        z_mean_t, z_std_t, 10)
    KL = nep.KL_div(z_mean_t, z_std_t, z_mean_c, z_std_c)
    ELBO = - log_likelihood + KL
    kl.append(KL)
    elbo.append(ELBO)
    ELBO.backward()
    optimizer.step()

    if epoch % 1000 == 0:
        print('Epoch:', epoch)
        y_pred = nep.predict(test_x, g, z_mean_t, z_std_t, 100)

        # Plot of sample functions
        plt.plot(x_c.numpy(), y_c.numpy(), 'ro')
        for i in y_pred:
            plt.plot(test_x.numpy(), i, 'grey', alpha=0.1)
        plt.plot(test_x.numpy().flatten(),
                    test_y.numpy().flatten(), 'k*')
        plt.show()
print('Training successful')



# PLOTS
y_pred = nep.predict(test_x, g, z_mean_t, z_std_t, 100)
quantile_05, median, quantile_95 = np.percentile(y_pred,
                                    [5, 50, 95], axis=0)
# Plot of sample functions
for i in y_pred:
    plt.plot(test_x.numpy(), i, 'grey', alpha=0.1)
plt.plot(truth_x.numpy(), truth_y.numpy(), 'r--')
plt.show()

# Plot of distribution over functions
plt.plot(test_x.numpy().flatten(), test_y.numpy().flatten(),
                                'ro', markersize=6)
plt.plot(truth_x.numpy(), truth_y.numpy(), 'r--')
plt.plot(test_x.numpy().flatten(), median.flatten(),
                        '-', linewidth=3)
plt.fill_between(test_x.numpy().flatten(), quantile_05.flatten(),
            quantile_95.flatten(), alpha=0.15, facecolor='#089FFF')
plt.legend(['Test Set', 'True DGP', 'Prediction', 'Confidence'])
plt.show()
```

A.3 Code for Experiment 2

```
import neural_process as nep
import numpy as np
import torch
import matplotlib.pyplot as plt
%matplotlib inline



###----- Generate data
torch.manual_seed(19051994)
datasets = {}
n_tasks = 30
```

```
period = []

truth_x = torch.linspace(0, 1, 150)
x = torch.linspace(0, 1, 25)

for i in range(n_tasks):
    a = np.random.uniform(2, 4)
    y = torch.sin(a * np.pi * x) + torch.randn(x.size()) * 0.2
    truth_y = torch.sin(a * np.pi * truth_x) +
                torch.randn(truth_x.size()) * 0.2
    datasets['task_{}'.format(i)] = y.unsqueeze(-1),
                truth_y.unsqueeze(-1)
    plt.plot(truth_x.numpy(), truth_y.numpy())

# plt.title('Tasks')
# plt.ylabel('y = a sin(bx)')
# plt.xlabel('x')
print('Sizes\n', ' x:', x.shape, '\n  y:', y.shape, '\nfor each task')
plt.show()



###----- NP Definition
x_dim = 1
y_dim = 1
r_dim = 3#8
z_dim = 3#8

encoder_specs = [(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 (10, torch.nn.Tanh()),
                 (r_dim, None)]

decoder_specs = [(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()),
                 #(10, torch.nn.Tanh()), (10, torch.nn.Tanh()),
                 (10, torch.nn.Tanh()),
                 (y_dim, None)]

encoder_input_dim = x_dim + y_dim
decoder_input_dim = x_dim + z_dim

h = nep.Encoder(encoder_input_dim, encoder_specs,
                init_func=torch.nn.init.kaiming_normal_)
r_to_z = nep.Zparams(r_dim, z_dim)
g = nep.Decoder(decoder_input_dim, decoder_specs,
                init_func=torch.nn.init.kaiming_normal_)
optimizer = torch.optim.Adam(params=list(g.parameters()) +
                list(h.parameters()) + list(r_to_z.parameters()),
                lr=1e-3)



###----- Handle data
x_t = x.unsqueeze(-1)
truth_x = truth_x.unsqueeze(-1)



###----- NP Training
epochs = 10000
kl = []
elbo = []

for epoch in range(epochs):
    optimizer.zero_grad()

    # select a task randomly
    task_id = 'task_' + str(np.random.randint(n_tasks))
    y_t, truth_y = datasets[task_id]

    # select number of context points randomly
    n_context = np.random.randint(1, x_t.shape[0])

    # select `n_context` points and create the context
    #set and target set
    context_indeces = np.random.randint(x_t.shape[0],
                                        size=n_context)
    x_c = x_t[context_indeces]
    y_c = y_t[context_indeces]

    # variational parameters (mean, std) of approximate prior
    z_mean_c, z_std_c = r_to_z(h(x_c, y_c))

    # variational parameters (mean, std) of approximate posterior
    z_mean_t, z_std_t = r_to_z(h(x_t, y_t))

    log_likelihood = nep.MC_loglikelihood(x_t, y_t, g,
                                        z_mean_t, z_std_t, 10)
    KL = nep.KL_div(z_mean_t, z_std_t, z_mean_c, z_std_c)
    ELBO = - log_likelihood + KL
    kl.append(KL)
    elbo.append(ELBO)
    ELBO.backward()
    optimizer.step()

    if epoch % 1000 == 0:
        print('Epoch:', epoch)
        y_pred = nep.predict(x_t, g, z_mean_t, z_std_t, 100)
        # Plot of sample functions
        plt.plot(x_c.numpy(), y_c.numpy(), 'ro')
        plt.plot(truth_x.numpy(), truth_y.numpy(), 'k*')
        for i in y_pred:
            plt.plot(x.numpy(), i, 'grey', alpha=0.15)
        plt.show()
print('Training successful')



###----- NP Learned Prior
y_pred = nep.predict(truth_x, g, z_mean_t, z_std_t, 100)
quantile_05, median, quantile_95 = np.percentile(y_pred,
                [5, 50, 95], axis=0)

# Plot of sample functions
for i in y_pred:
    plt.plot(truth_x.numpy(), i, 'grey', alpha=0.1)
plt.show()

# Plot of distribution over functions
plt.plot(truth_x.numpy().flatten(), median.flatten(), '--')
plt.fill_between(truth_x.numpy().flatten(), quantile_05.flatten(),
            quantile_95.flatten(), alpha=0.15, facecolor='#089FFF')
plt.legend(['Prediction', 'Confidence'])
plt.show()



###----- Generate new task
index_train = np.random.choice(y.shape[0], 10, replace=False)
y_new = torch.sin(3.5 * np.pi * x) + torch.randn(x.size()) * 0.2
y_new = y_new.unsqueeze(-1)
x_new = x.unsqueeze(-1)

y_new_train = y_new[index_train]
x_new_train = x_new[index_train]
print(index_train.shape)



###----- Train on new task (few-shots)
epochs = 1000

for epoch in range(epochs):
    optimizer.zero_grad()

    # select number of context points randomly
    n_context = np.random.randint(1, y_new_train.shape[0])
    # select ‘n_context‘ points and create the context set
    # and target set
    context_indeces = np.random.choice(y_new_train.shape[0],
                            n_context, replace=False)
    x_c = x_new_train[context_indeces]
    y_c = y_new_train[context_indeces]

    # variational parameters (mean, std) of approximate prior
    z_mean_c, z_std_c = r_to_z(h(x_c, y_c))

    # variational parameters (mean, std) of approximate posterior
    z_mean_t, z_std_t = r_to_z(h(x_new_train, y_new_train))

    # Monte Carlo estimate of log-likelihood
    # (expectation wrt approximate posterior)
    log_likelihood = nep.MC_loglikelihood(x_t, y_t, g,
                                    z_mean_t, z_std_t, 10)

    # compute KL divergence analytically
    KL = nep.KL_div(z_mean_t, z_std_t, z_mean_c, z_std_c)

    # compute negative ELBO
    ELBO = - log_likelihood + KL

    z = nep.sample_z(z_mean_c, z_std_c, how_many=1)
    y_pred, _ = g(x_new_train, z)
    kl.append(KL)
    elbo.append(ELBO)

    # compute gradient of ELBO and take a gradient step
    ELBO.backward()
    optimizer.step()

    if epoch % 1000 == 0:
        print('Epoch:', epoch)
        y_pred = nep.predict(x_new, g, z_mean_t, z_std_t, 100)

        # Plot of sample functions
        plt.plot(x_new.numpy(), y_new.numpy(), 'k*')
        plt.plot(x_new_train.numpy(), y_new_train.numpy(), 'ro')
        for i in y_pred:
            plt.plot(x_new.numpy(), i, alpha=0.4)
        plt.show()
print('Training successful')



###----- Plot learned posterior
y_pred = nep.predict(x_new, g, z_mean_t, z_std_t, 100)
```

```
quantile_05 , median , quantile_95 = np.percentile(y_pred ,
                      [5 , 50 , 95] , axis=0)

plt.plot(x_new.numpy() , y_new.numpy() , 'k*')
plt.plot(x_new_train.numpy() , y_new_train.numpy() , 'ro')
plt.plot(x_new.numpy().flatten() , median.flatten() , '--')
plt.fill_between(x_new.numpy().flatten() , quantile_05.flatten()-0.2 ,
            quantile_95.flatten()+0.2 , alpha=0.15 ,
            facecolor='#089FFF')
plt.legend(['Test Set', ' Train Set', 'Prediction', 'Confidence'])
plt.show()
```