Advancements in Variational Inference: Methods, Optimisation Techniques, and Applications

Tom Blain

Supervised by Dr Dennis Prangle
Level M
40 Credit Points

August 14, 2023

Project complies with guidelines set out in the Undergraduate Project Handbook

Acknowledgement of Sources

For all ideas taken from other sources (books, articles, internet), the source of the ideas is mentioned in the main text and fully referenced at the end of the report.

All material which is quoted essentially word-for-word from other sources is given in quotation marks and referenced.

Pictures and diagrams copied from the internet or other sources are labelled with a reference to the web page or book, article etc.

Signed Tom Blain

Date 24/04/2023

Abstract

Variational Inference (VI) has emerged as a powerful tool for approximating intractable posterior distributions in Bayesian inference. This paper presents a comprehensive review and analysis of the developments, techniques, and applications of VI. We begin by exploring the Kullback-Leibler (KL) divergence and its properties, and derivation of the Evidence Lower Bound (ELBO), which serves as the foundation for VI. We study classical implementations of VI such as Coordinate Ascent VI with the mean field assumption, as well as more modern methods focused on scalability such as Stochastic VI (SVI) and easily implementable methods such as Automatic differentiation VI (ADVI). We focus on reviewing methods and developments of efficient implementation and scalability. Finally, we conclude by identifying potential areas for future research, such as alternative divergences, SGD variance reduction, and the integration of Markov Chain Monte Carlo (MCMC) with VI. This paper serves as a valuable resource for students, researchers, and practitioners interested in understanding and applying VI techniques in Bayesian modeling. We also include examples in python throughout, which are intended to serve as a guide for various implementations of VI.

Contents

1 Introduction

A fundamental challenge in modern statistics is to estimate probability densities that are difficult to compute. This challenge is especially important in Bayesian statistics, which frames all inference about unknown quantities as a calculation about the posterior. In order to deal with models for which the posterior is computationally infeasible, modern Bayesian statistics relies on algorithms for approximating these distributions [1].

We begin by defining the posterior as follows with parameters θ and data x,

$$p(\theta|x) = \frac{p(\theta)p(x|\theta)}{p(x)} \tag{1}$$

where $p(\theta)$ is the likelihood, $p(x|\theta)$ is the prior distribution and $p(x)$ is the normalising constant. A prior distribution allows us to express our prior beliefs as a probability distribution which can capture uncertainty or information about a parameter or latent variable.

The normalising constant, also known as the evidence or marginal likelihood, is defined as follows for continuous θ

$$p(x) = \int p(\theta)p(x|\theta)d\theta \tag{2}$$

Equation (1) comes directly from Bayes' theorem. Bayes' theorem provides a way to update our beliefs about a hypothesis based on new evidence, combining prior knowledge with observed data to determine the most probable outcome.

The difficulty in finding our true posterior lies in evaluating our normalising constant, which can often not be easily computed explicitly. For most models, this integral is high dimensional, thus computing the normalization term is intractable. For the exact posterior in complex models, we will often have to resort to approximation hence in Bayesian methods we will often instead find the posterior up to proportionality. We might use proportionality because it allows us to simplify the process of updating our beliefs. Sometimes we only need to compare the posterior probabilities of different hypotheses, rather than computing their exact values, for our inference.

$$p(\theta|x) \propto p(\theta)p(x|\theta) \tag{3}$$

The basic idea of VI is to approximate the posterior with a simpler distribution. We can choose a family of probability density functions and then find the density as close to the actual density as possible - often minimising the KL divergence which is a measure of dissimilarity between two distributions [2]. This approach therefore is an optimisation problem.

Historically, Markov Chain Monte Carlo algorithms (MCMC) have been the most widely used approaches by mathematicians for approximating the true posterior (construct a markov chain, sample from the chain, approximate posterior from an estimate collected from samples) [3]. MCMC has been widely studied, extended, and applied. However, MCMC convergence can be slow and this especially runs into problems when we look into the world of big data and complex models [4].

MCMC approaches rely on sampling. It is usually not difficult to construct a chain with properties of convergence to the true posterior [5]. On the other hand, optimisation based methods are much faster but often suffer from oversimplified models which do not capture the true posterior accurately. When deciding on an approach to take, the drawbacks of both methods should be considered.

This paper will focus on Variational Inference, an optimisation based method which has significantly grown in popularity over recent years due in part to the need for efficient and scalable inference.

1.1 Paper Outline

An overview of the paper:

- Chapter 2 will dive into the KL divergence - how and why is this divergence used for our purpose in VI, and are there other alternative divergences we should think about.
- Chapter 3 follows naturally by deriving the ELBO from the KL divergence, and proving the equivalence in our optimisation.
- Chapter 4 will show the classical mean field VI algorithm with coordinate ascent, and show our first easily followable example of implementing VI for estimating the parameters of a guassian mixture model.
- Chapter 5 looks into stochastic gradient descent and adaptive learning rate methods, as well as investigating the natural gradient and scalability of the natural gradient via an example. We study stochastic optimisation conditions, and many variance reduction methods which show potential to improve SGD algorithms.
- Chapter 6 defines the reparameterization trick, a method enabling efficient backpropagation of gradients by transforming random variables into deterministic functions.
- Chapter 7 will show Automatic Differentiation VI, and closely looks into methods computers use to calculate derivatives and how automatic differentiation has been a highly valuable tool in scientific computing. We conclude this section with two examples implementing Automatic Differentiation Variational Inference (ADVI) using PyMC3.
- Chapter 8 will conclude the paper and summarise the key points. We also make a note of future areas of research for VI, and mention some interesting uses currently being developed.
- Appendix A provides an introduction to Neural Networks, which are used in many examples throughout the paper. Refer to this appendix for a background in Neural network terminology and architecture.

2 KL Divergence

2.1 Introduction

In variational inference, we are looking to approximate the true posterior distribution with a variational distribution. In order to find the best variational distribution to use, we need to find a way of measuring how similar this approximation is to the true posterior.

With our goal in mind, what properties might we want from a metric or divergence that will help us to find our best variational distribution [6]. We need to optimise this function over a range of possible parameter choices for our variational distribution.

- The exact posterior should be the best candidate.
- The quality of the approximation should be proportional to the value of our function.
- The function should be optimisable.

This problem appears to parallel information theory, where we are concerned with information loss; we want to maintain as much information as possible. From information theory comes a powerful divergence known as the Kullback–Leibler divergence.

2.2 Definition

The Kullback–Leibler divergence (often abbreviated KL Divergence) is a divergence which gives us a measures of the dissimilarity of two probability distributions [7].

A statistical divergence $D : X \times X \to \mathbb{R}^+$ should satisfy

- non-negativity: $D(p, q) \geq 0$
- zero if and only if equal: $D(p, q) = 0$ iff $p = q$

where p and q are probability densities and X refers to the sample space of a probability distribution.

The KL divergence between two continuous probability densities p and q is defined as:

$$D_{KL}(p(x)||q(x)) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{4}$$

The parameters of the densities, θ, are not explicitly included in the formula for the KL divergence. However, they are assumed to be fixed for a given comparison between p and q. The densities themselves are functions of x, not θ, so the integration variable in the formula must also be x.

Note this is a divergence and not a metric since it does not satisfy symmetry and triangle inequality conditions.

Symmetry: $D(p, q) = D(q, p)$,
Triangle inequality: $D(p, q) \leq D(p, g) + D(g, q)$, where g is a continuous probability density.

2.3 Proof of Properties

To prove the KL divergence is always non negative [8], we will use Jensen's inequality [9], which states that for any convex function f, we have that

$$f\left(\int x dx\right) \leq \int f(x) dx \tag{5}$$

If f is concave, then the inequality reverses.

Proof: Let $A = \{x : p(x) > 0\}$ be the support of $p(x)$ Using the concavity of the log function and Jensen's inequality, we have that

$$\begin{aligned}
-D_{KL}(p(x)||q(x)) &= -\int_{x \in A} p(x) \log \frac{p(x)}{q(x)} dx && (6)\\
&= \int_{x \in A} p(x) \log \frac{q(x)}{p(x)} dx && (7)\\
&\leq \log \int_{x \in A} p(x) \frac{q(x)}{p(x)} dx && (8)\\
&\leq \log \int_{x \in X} q(x) dx && (9)\\
&= \log 1 && (10)\\
&= 0 && (11)
\end{aligned}$$

We have equality in our first inequality if and only if $p(x) = cq(x)$ for some constant c, and equality in the second inequality if and only if $\int_{x \in A} q(x)dx = \int_{x \in X} q(x)dx$ which implies c = 1. Hence $D_{KL}(p(x)||q(x)) = 0$ if and only if $p(x) = q(x)$ for all x.

2.4 Further Properties

2.4.1 Invariance to Reparameterizations

If we transform our random variable from x to some $y = f(x)$ we know that $p(x)dx = p(y)dy$ and $q(x)dx = q(y)dy$, Hence the KL divergence remains the same for both random variables [6].

In variational inference, invariance to reparameterization allows us to express the same variational distribution in multiple ways, giving us greater flexibility in selecting the parametric family for $q(x)$. Furthermore, and importantly, it will allow us to make use of the reparameterization trick which we discuss in section 6 [10].

2.4.2 Chain Rule for KL Divergence

The KL divergence satisfies a natural chain rule. Proof from source: [11]:

$$D_{KL}(p(x,y)||q(x,y)) = \int p(x,y) \log \frac{p(x,y)}{q(x,y)} dx\ dy \tag{12}$$

$$= \int p(x,y) \left[\log \frac{p(x)}{q(x)} + \log \frac{p(y|x)}{q(y|x)} \right] dx\ dy \tag{13}$$

$$= D_{KL}(p(x)||q(x)) + E_{p(x)} \left[D_{KL}(p(y|x)||q(y|x)) \right] \tag{14}$$

2.4.3 Convexity

The log-sum inequality states [12] that for any non negative real numbers $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$:

$$\sum_{i=1}^{n} a_i \log \frac{a_i}{b_i} \geq \left(\sum_{i=1}^{n} a_i \right) \log \frac{\sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} b_i} \tag{15}$$

Thus we can rewrite the KL of the mixture distribution as

$$D_{KL} \left[\lambda p_1 + (1-\lambda) p_2 || \lambda q_1 + (1-\lambda) q_2 \right] \tag{16}$$

$$= \sum_{x \in X} \left[[\lambda p_1(x) + (1 - \lambda p_2(x)] \cdot \log \frac{\lambda p_i(x) + (1 - \lambda p_2(x)}{\lambda q_1(x) + (1 - \lambda q_2(x)} \right] \tag{17}$$

And using the log sum inequality, 15

$$\leq \sum_{x \in X} \left[\lambda p_1(x) \cdot \log \frac{\lambda p_1(x)}{\lambda q_1(x)} + (1 - \lambda) p_2(x) \cdot \log \frac{(1 - \lambda) p_2(x)}{(1 - \lambda) q_2(x)} \right] \tag{18}$$

$$= \lambda \sum_{x \in X} p_1(x) \cdot \log \frac{p_1(x)}{q_1(x)} + (1 - \lambda) \sum_{x \in X} p_2(x) \cdot \log \frac{p_2(x)}{q_2(x)} \tag{19}$$

$$= \lambda D_{KL}[p_1 || q_1] + (1 - \lambda) D_{KL}[p_2 || q_2] \tag{20}$$

Hence we have proven that

$$D_{KL} \left[\lambda p_1 + (1 - \lambda) p_2 || \lambda q_1 + (1 - \lambda) q_2 \right] \leq \lambda D_{KL}[p_1 || q_1] + (1 - \lambda) D_{KL}[p_2 || q_2] \tag{21}$$

And hence the KL divergence is convex in (p,q). Proof from source: [13]

2.5 Forward and Reverse KL

In general, we have asymmetry in the definition of the KL divergence,

$$D_{KL}(p||q) \neq D_{KL}(q||p) \text{ for all } p, q$$

Consider two Gaussian distributions, $p(x)$ and $q(x)$, with different means and variances and compute the KL divergences $D_{KL}(p||q)$ and $D_{KL}(q||p)$.

$$D_{KL}(p||q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

$$D_{KL}(q||p) = \int_{-\infty}^{\infty} q(x) \log \frac{q(x)}{p(x)} dx$$

Solving this integral, for arbitrary means and variances we have

$$D_{KL}(p||q) = \frac{1}{2} \left(\log \frac{\sigma_q^2}{\sigma_p^2} - 1 + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{\sigma_q^2} \right)$$

$$D_{KL}(q||p) = \frac{1}{2}\left(\log\frac{\sigma_p^2}{\sigma_q^2} - 1 + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{\sigma_p^2}\right)$$

It is therefore easy to see that these are not always equal for different $\mu_p, \mu_q, \sigma_p, \sigma_q$

In the context of machine learning, the "forward KL" refers to the KL divergence from the model distribution to the true distribution, while the "reverse KL" refers to the KL divergence from the true distribution to the model distribution [14] [15].

Forward KL divergence can be represented as:

$$D_{KL}(p||q)$$

Reverse KL divergence can be represented as:

$$D_{KL}(q||p)$$

Where p is our true distribution and q is our variational distribution.

In general usage, the forward KL divergence represents the amount of information loss when approximating the true distribution with the model distribution [16]. It is often used as a loss function in generative models to measure the difference between the generated data and the real data [17]. In this case, minimizing the forward KL divergence is equivalent to maximizing the likelihood of the data [18].

On the other hand, the reverse KL divergence measures the amount of information gained by using the variational distribution instead of the true distribution. It is often used in variational inference where the goal is to find the closest approximation of the true posterior distribution using a simpler distribution. In this case, minimizing the reverse KL divergence is equivalent to finding the optimal approximation [15].

Finding a q that is close to p by minimizing $D_{KL}(p||q)$ (forwards KL) gives different behavior than minimizing $D_{KL}(q||p)$ (reverse KL). To prevent $D_{KL}(p||q)$ from becoming very large, we must have $q > 0$ whenever $p > 0$, which means the behaviour becomes "zero avoiding". In contrast, to prevent $D_{KL}(q||p)$

from becoming very large, we must have $q = 0$ whenever $p = 0$, which creates "zero forcing" behavior [14].

2.5.1 Gaussian Example

Let

$$p(x) = \sum_{k=1}^{K} \pi_k \text{Normal}(x; \mu_k, \sigma_k^2) \tag{22}$$

$$q_\phi(x) = \text{Normal}(x; \mu_q, \sigma_q^2) \tag{23}$$

Where $K = 2$, $\pi_k = 0.5$, $\sigma_k^2 = 1 (k = 1, 2)$, $\mu_1 = 0$, μ_2 is increasing from 0 to 10 and $\phi = (\mu_q, \sigma_q^2)$

The behavior of minimising the forwards and reverse KL divergences with respect to q_ϕ is shown in figure 1 [14]

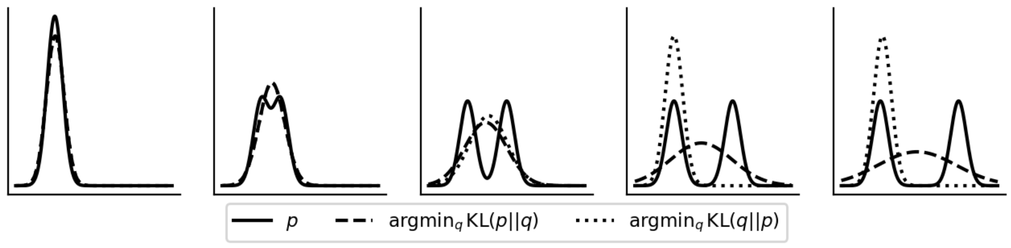


Figure 1: Plot demonstrating zero seeking and zero avoiding behavioral differences between forward and reverse KL. From left to right, $\mu_2 = [0, 2.5, 5, 7.5, 10]$. For each figure, we minimise the KL divergence for our variational distribution (q) and true distribution (p). Figure and example taken from [14].

2.6 Alternative Divergences

The KL-divergence is a strong choice in variational inference

- Information theory foundation: KL divergence is grounded in information theory, measuring the average number of extra bits needed to encode events from one distribution using the optimal code for another distribution [19].

- Computational tractability: KL divergence has desirable properties which were highlighted in this section, such as being non-negative, invariant to reparameterization, convex and following the chain rule. Each property has a specific advantage, but to summarise these make optimisation easier and more efficient in practice.

However, it would be unwise to assume its always going to be the best choice,

Utilising other divergences could be useful due to the following reasons:

- Sensitivity to choice of approximating family: KL divergence is sensitive to the choice of the approximating distribution family $q(\theta)$. If the family does not include the true posterior, the KL divergence may lead to biased estimates. Alternative divergences could be more robust to model misspecification [20].
- Different divergence properties: Alternative divergences may have different properties, which might be more suitable for specific applications. For example, the Reverse KL divergence $(KL(p||q))$ puts more emphasis on fitting the tails of the true posterior, while the Forwards KL divergence $(KL(q||p))$ focuses on fitting the modes [15] [21].
- Robustness to outliers: Some divergences can provide more robust estimates in the presence of outliers, as they balance the trade-off between fitting the modes and tails of the true posterior [22].

Several alternative divergence measures can be used in Variational Inference to overcome the limitations of the KL divergence. Some of these alternatives include [23] [24]:

- Rényi divergence: Also known as α-divergence is a family of divergence measures indexed by a scalar parameter α. Both KL divergence and Reverse KL divergence are special cases of Rényi divergence. By varying α, we can balance the trade-off between fitting the modes and tails of the true posterior, which might be beneficial in some applications [25].

  $$D_\alpha(p||q) = \frac{1}{\alpha - 1} \log \sum_{x \in X} p(x)^\alpha q(x)^{1-\alpha} \tag{24}$$

  This is equal to the KL divergence when α tends to 1.

- β-divergence: This family encompasses various divergence measures, including KL divergence, Reverse KL divergence, and the Hellinger distance. Like Rényi divergence, The β divergence allows for a flexible trade-off between fitting the modes and tails of the true posterior [26] [22].

$$D_\beta(p||q) = \frac{1}{\beta(\beta-1)} \sum_{x \in X} \left(p(x)^\beta - \beta p(x) q(x)^{\beta-1} + (1-\beta) q(x)^\beta \right) \quad (25)$$

- Total Variation (TV) distance: The TV distance is a measure of dissimilarity between two probability distributions based on the absolute difference of their probability densities. It is more robust to outliers and can better handle distributions with disjoint support compared to KL divergence. However, it is computationally more challenging to optimise.

- Wasserstein distance: The Wasserstein distance measures the dissimilarity between two probability distributions based on the minimum amount of "work" required to transform one distribution into another. It has gained popularity in generative modeling, specifically in the context of Generative Adversarial Networks (GANs) [23]. Wasserstein distance can be more robust to mode collapse (see [27]) and provide better convergence properties compared to KL divergence.

- Jenson-Shannon divergence: The JS divergence is a symmetric measure of dissimilarity between two probability distributions, based on the average of two KL divergences with respect to a mixture distribution. It is more robust to model misspecification and often easier to optimise due to its symmetric nature.

$$D_{JS}(p||q) = \frac{1}{2} D_{KL}(p||m) + \frac{1}{2} D_{KL}(q||m) \quad (26)$$

  where $m = \frac{1}{2}(p+q)$ is the mixture distribution, and $D_{KL}(p||q)$ is the Kullback-Leibler divergence:

Ultimately, the KL divergence is very widely used and is well balanced in terms of computational simplicity and accuracy, offering a highly scalable solution. Implementation of alternative divergences is an actively studied area as researchers look into implementation [28].

3 Evidence Lower Bound

3.1 Introduction

Recalling our outline of Variational Inference, we are concerned with finding an optimal approximating distribution to the true posterior distribution. Statistical divergences, such as the KL divergence, provide ways to measure closeness of two distributions and therefore provide functions we could optimise for the VI method.

In this section we derive the Evidence Lower BOund (ELBO), a computationally tractable objective which when maximised is equivalent to minimising the KL. Deriving the ELBO shows why the KL divergence is a very practical divergence to use in VI. We then continue and show how the ELBO closely relates to the evidence and the KL divergence, KL = evidence - ELBO. We conclude by showing a basic recipe for VI.

3.2 Deriving the ELBO

In variational inference, we specify a family of densities $\mathbb{Q}$ over the parameters. Each $q(\theta)$ is a variational approximation to the exact posterior. Our goal is to find the best variational approximation, for which we use the KL divergence. We can formulate this as

$$q^*(\theta) = argmin_{q(\theta) \in \mathbb{Q}} D_{KL}(q(\theta)||p(\theta|x)) \tag{27}$$

However, this objective is not computable because it requires computing $\log p(x)$ from our definition of the KL divergence. Recall that $p(x)$ is the evidence from our original posterior equation, which was the often intractable element we set out trying to approximate.

We attempt to find a bound for $\log p(x)$

$$\log p(x) = \log \int_\theta p(x,\theta) d\theta \tag{28}$$

$$= \log \int_\theta p(x,\theta) \frac{q(\theta)}{q(\theta)} d\theta \tag{29}$$

$$= \log \left(\mathbb{E}_q \left[\frac{p(x,\theta)}{q(\theta)} \right] \right) \tag{30}$$

By using Jensen's inequality [9], we have

$$\log p(x) \geq \mathbb{E}q \left[\log p(x,\theta)\right] - \mathbb{E}q \left[\log q(\theta)\right] \tag{31}$$

This is the ELBO (Evidence Lower Bound). The goal of variational inference is to find an optimal $q(\theta)$ that maximises the ELBO, bringing it closer to the true log marginal likelihood, $\log p(x)$. [29] [30].

We therefore just focus on maximising the term

$$\mathcal{L}(\psi) \triangleq \mathbb{E}_{q(\theta|\psi)} \left[\log p(x|\theta) + \log p(\theta) - \log q(\theta|\psi)\right] \tag{32}$$

Here, $q(\theta|\psi)$ is the approximate posterior distribution with variational parameters ψ used to define the approximate posterior distribution.

This is equivalent to minimising the KL divergence up to an additive constant. Maximising the ELBO gives as tight a bound as possible on the marginal probability of x.

3.3 The Gap Between the Evidence and the ELBO

We find that the gap between the evidence and the ELBO is precisely the KL divergence between $p(\theta|x)$ and $q(\theta)$

This can be derived as follows [29]:

$$\begin{aligned}
D_{KL}(q(\theta|\psi)||p(\theta|x)) &= \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log\frac{q(\theta|\psi)}{p(\theta|x)}\right] && (33)\\
&= \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log q(\theta|\psi)\right] - \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log\frac{p(x,\theta)}{p(x)}\right] && (34)\\
&= \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log q(\theta|\psi)\right] - \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log p(x,\theta)\right] + \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log p(x)\right] && (35)\\
&= \log p(x) - \mathbb{E}_{\theta\sim q(\theta|\psi)}\left[\log\frac{p(x,\theta)}{q(\theta|\psi)}\right] && (36)\\
&= \text{evidence} - \text{ELBO} && (37)
\end{aligned}$$

3.4 Basic VI Recipe

We are now able to define a fundamental outline of a VI algorithm, using the ELBO objective function.

1. Define the model: Specify the joint probability distribution $p(x, \theta)$, where x represents the observed data and θ denotes the model parameters. This model should incorporate both the likelihood $p(x|\theta)$ and the prior distribution $p(\theta)$ over the parameters.

2. Choose the variational family: Select an appropriate variational family for the variational distribution, $q(\theta|\psi)$. The choice of the variational family depends on the desired trade-off between the complexity of the approximation and computational efficiency. Common choices include Gaussian distributions, mean-field approximations, or more expressive distributions such as Gaussian mixtures.

3. Compute the ELBO: Derive the Evidence Lower Bound as a function of the variational parameters ψ. The ELBO serves as a lower bound on the evidence and is defined as:

$$\mathcal{L}(\psi) \triangleq \mathbb{E}_{q(\theta|\psi)}\left[\log p(x|\theta) + \log p(\theta) - \log q(\theta|\psi)\right]$$

4. Optimize the ELBO: Find the optimal values of the variational parameters ψ that maximise the ELBO. This step can be performed using optimisation methods, such as stochastic gradient ascent (SVI) or coordinate ascent (CAVI).

5. Approximate posterior inference: Use the optimized approximate posterior distribution $q(\theta|\psi)$ for desired posterior inference tasks.

This recipe proposed is the backbone of all modern variational inference algorithms. Complexities and difficulties then come from various choices, simplifications, and scalability trade offs.

Classic approaches such as using the mean field assumption [31], which adds an assumption that our parameters are independant, simplifies computation but may lead to less accuracy. Modern approaches such as ADVI [32] focus more on ease of implementation, usually picking a simply guassian family for the variational family and utilising tools such as automatic differentation [33] for efficient and simple gradient calculation. A focus on scalability is usually an important consideration for variational inference algorithms, since this is where we will see advantages over MCMC methods [1].

4 Mean Field Variational Inference

4.1 Introduction

We begin by showing Mean Field Variational Inference, since this is one of the most computationally simple methods for performing VI. By making additional assumptions on the parameters of the model, we are able to factorise and simplify our maximisation with practical algorithms such as Coordinate ascent.

This approach offers a statistically simplified and easier to implemented algorithm and consequently, has been used as a standard method since the inception of the method in 1999 [34]. However, imposing this additional assumption for computational simplicity may result in less accurate approximations [35] [36].

4.2 Definition

There is a balancing act in choosing $q(\theta; \psi)$ expressive enough to approximate the posterior well, and simple enough to lead to a tractable approximation. A common choice is a fully factorised distribution, also called mean field distribution. A mean field approximation assumes that all parameters are independent, which simplifies derivations.

$$q(\theta; \psi) = \prod_{i=1}^{m} q(\theta_i; \psi_i) \tag{38}$$

Where $q(\theta_i; \psi_i)$ is the variational approximation for a single parameter [30].

We could also partition the parameters $\theta_1, \ldots, \theta_m$ into R groups $\theta_{G_1}, \ldots, \theta_{G_R}$ and use the approximation

$$q(\theta) = q(\theta_{G_1}, \ldots, \theta_{G_R}) = \prod_{r=1}^{R} q(\theta_{G_r}) \tag{39}$$

Typically, this approximation does not contain the true posterior because our assumption is often not representative of the true distribution where the parameters might be dependant or correlated. Nonetheless, this approximation can still provide valuable insights and be useful in practice.

A popular method for optimizing the ELBO with this mean field assumption is coordinate ascent, by iteratively optimizing the variational approximation of each parameter $q(\theta_i; \psi_i)$ while holding the others fixed [31].

Recall the probability chain rule for continuous random variables $X_1, ..., X_n$:

$$p(X_1, ..., X_n) = p(X_n | X_1, ..., X_{n-1}) \cdot p(X_1, ..., X_{n-1}) \tag{40}$$

Or in its general form,

$$p\left(\bigcap_{k=1}^{n} X_k\right) = \prod_{k=1}^{n} p\left(X_k | \bigcap_{j=1}^{k-1} X_j\right) \tag{41}$$

We apply the chain rule to the joint probability density function $p(\theta_{1:m}, x_{1:n})$ for m parameters and n data points.

$$p(\theta_{1:m}, x_{1:n}) = p(x_{1:n}) \prod_{j=1}^{m} p(\theta_j | \theta_{1:(j-1)}, x_{1:n}) \tag{42}$$

We can decompose the entropy term in the ELBO using the mean field approximation as:

$$\mathbb{E}_q[\log q(\theta_{1:m})] = \sum_{j}^{m} \mathbb{E}_{q_j}[\log q(\theta_j; \psi_j)] \tag{43}$$

Where q_j refers to the variational distribution for the j^{th} model parameter,

Inserting the two previous reductions in the ELBO equation 32, we can now express the ELBO for the mean field approximation as:

$$ELBO(q) = \mathbb{E}_q[\log p(x_{1:n}, \theta_{1:m})] - \mathbb{E}_{q_j}[\log q(\theta_{1:m}; \psi_{1:m})] \tag{44}$$

$$= \mathbb{E}_q\left[\log\left(p(x_{1:n}) \prod_{j=1}^{m} p(\theta_j | \theta_{1:(j-1)}, x_{1:n})\right)\right] - \sum_{j}^{m} \mathbb{E}_{q_j}[\log q(\theta_j; \psi_j)] \tag{45}$$

$$= \log p(x_{1:n}) + \sum_{j}^{m} \mathbb{E}_q[\log p(\theta_j | \theta_{1:(j-1)}, x_{1:n})] - \sum_{j}^{m} \mathbb{E}_{q_j}[\log q(\theta_j; \psi_j)] \quad (46)$$

We can now apply the coordinate ascent algorithm to compute the ELBO under the mean field assumption.

4.3 Coordinate Ascent

Coordinate Ascent Variational Inference (CAVI) is a widely-used optimisation algorithm in the context of VI under the mean field assumption. The mean field assumption assumes that the approximating distribution factorises over the parameters, which simplifies the optimisation problem and allows for efficient computations. CAVI makes use of this factorised structure to perform iterative, coordinate-wise optimisation of the variational parameters to minimise the divergence between the approximating distribution and the true posterior distribution.

The CAVI algorithm proceeds by iteratively updating each coordinate (i.e., variational parameter) of the approximating distribution while keeping the other coordinates fixed. This coordinate-wise optimisation strategy enables the algorithm to focus on one aspect of the problem at a time, thereby ensuring convergence to a local minimum of the negative ELBO [37]. As a deterministic optimisation approach, CAVI provides stable convergence behavior, which is particularly advantageous for models that exhibit conjugacy between the likelihood and prior distributions. Stable convergence implies that the algorithm behaves in a predictable and controlled manner during the optimisation process, making progress toward the optimal solution at each iteration. However, despite its desirable properties, CAVI may face challenges in scaling to large datasets or handling non-conjugate models [38].

The general update formula for the i^{th} parameter (θ_i) in Coordinate Descent can be written as:

$$\theta_i^{(t+1)} = \arg\min_{\theta_i} L(\theta_1^{(t)}, \theta_2^{(t)}, \ldots, \theta_{i-1}^{(t)}, \theta_i, \theta_{i+1}^{(t)}, \ldots, \theta_n^{(t)}) \quad (47)$$

Here, L is the loss function, n is the total number of parameters, and the superscripts t indicate the iteration number for each parameter. The algorithm iterates over all parameters, updating them one by one, until convergence is achieved or a stopping criterion is met.

4.3.1 Coordinate Ascent VI (CAVI) Summary

In the context of Coordinate Ascent Variational Inference (CAVI), the goal is to minimise the divergence between the approximating distribution $q(\theta)$ and the true posterior distribution $p(\theta|x)$ [1]. Under the mean field assumption, the approximating distribution factorises over the parameters, which can be written as:

$$q(\theta) = \prod_{i=1}^{n} q_i(\theta_i) \tag{48}$$

Here, n is the number of parameters, and $q_i(\theta_i)$ represents the approximating distribution for the i^{th} parameter.

The CAVI algorithm uses the general update formula for Coordinate Descent to iteratively optimise the approximating distribution for each parameter. Specifically, the update formula for the i^{th} latent variable in CAVI can be written as [37]:

$$q_i^{(t+1)}(\theta_i) = \arg\max_{q_i(\theta_i)} D_{KL}\left(q(\theta)||p(\theta|D)\right) \tag{49}$$

Here, KL denotes the Kullback-Leibler divergence, and t is the iteration number. The superscripts indicate the iteration number for each approximating distribution. Note that when minimizing KL divergence in CAVI, we implicitly optimise the Evidence Lower BOund (ELBO), which is equivalent to maximizing the lower bound on the log marginal likelihood.

In CAVI, conjugacy between the likelihood and prior distributions is not strictly required but allows for each each update for $q_i(\theta_i)$ to be done in closed-form by calculating the expectation with respect to all other parameters $q_j(\theta_j)$ for $j \neq i$, which exploits the mean field assumption. This closed-form solution speeds up the algorithm and contributes to its stable convergence behavior [39]. The algorithm iterates over all parameters, updating their approximating distributions one by one, until convergence is achieved or a stopping criterion is met.

4.4 Simple Coordinate Ascent Variational Inference (CAVI) Example

In this example, we apply the mean field assumption and coordinate ascent algorithms to VI to simply and efficiently estimate the parameters of a Gaussian mixture model (GMM)

1. Synthetic data is generated by drawing samples from two Gaussian distributions with different means and variances.
2. CAVI is applied to the synthetic data to estimate the parameters (means and variances) of the underlying Gaussian components. This involves iteratively updating the responsibilities (ϕ) and the variational parameters (μ, τ) until convergence.

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data
np.random.seed(42)
data = np.concatenate([np.random.normal(-5, 2, 100),
    np.random.normal(5, 2, 100)])

# Parameters
num_components = 2
# Iterations
num_iterations = 100

# Initialise variational parameters
mu_init = np.random.normal(0, 1, num_components)
tau_init = np.ones(num_components)
phi_init = np.ones((len(data), num_components)) / num_components
```

We generate a synthetic dataset of 200 data points by drawing samples from two Gaussian distributions: 100 data points with with mean -5 and standard deviation 2, and another 100 data points with mean 5 and standard deviation 2. We set num_components to 2, which represents the number of Gaussian components in the GMM. We also set num_iterations to 100, which determines the number of iterations for the CAVI algorithm. This value can be adjusted to control the convergence of the algorithm.

We next initialise the variational parameters (μ, τ) and the responsibilities (ϕ).

- mu_init represents the initial means of the Gaussian components in the variational distribution family. We randomly initialise these means by drawing samples from a Gaussian distribution with mean 0 and standard deviation 1.
- tau_init represents the initial precisions (inverse variances) of the Gaussian components in the variational distribution family. We initialize them as an array of ones with the same length as the number of Gaussian components.
- phi_init represents the initial responsibilities for each data point, which indicate the probability of each data point belonging to each Gaussian component. We initialize the responsibilities as a matrix with dimensions (number of data points, number of Gaussian components), filled with equal probabilities for each Gaussian component (1 / number of Gaussian components).

```
# CAVI algorithm
def cavi(data, mu, tau, phi, num_iterations):
    for _ in range(num_iterations):
        # Update responsibilities (phi)
        for i, x in enumerate(data):
            phi[i, :] = np.exp(np.log(tau) + x * mu
                - 0.5 * (mu ** 2 + 1 / tau))
            phi[i, :] /= np.sum(phi[i, :])

        # Update variational parameters (mu, tau)
        for k in range(num_components):
            mu[k] = np.sum(phi[:, k] * data) / np.sum(phi[:, k])
            tau[k] = np.sum(phi[:, k]) / (mu[k] ** 2 + 1)

return mu, tau, phi

mu, tau, phi = cavi(data, mu_init, tau_init, phi_init, num_iterations)
```

This part of the code defines the Coordinate Ascent Variational Inference algorithm, which is used to estimate the parameters of the Gaussian Mixture

Model. The CAVI algorithm iteratively updates the responsibilities (ϕ) and the variational parameters (μ, τ) until convergence.

We then execute the function with out initial values and data. This will return our updated parameters

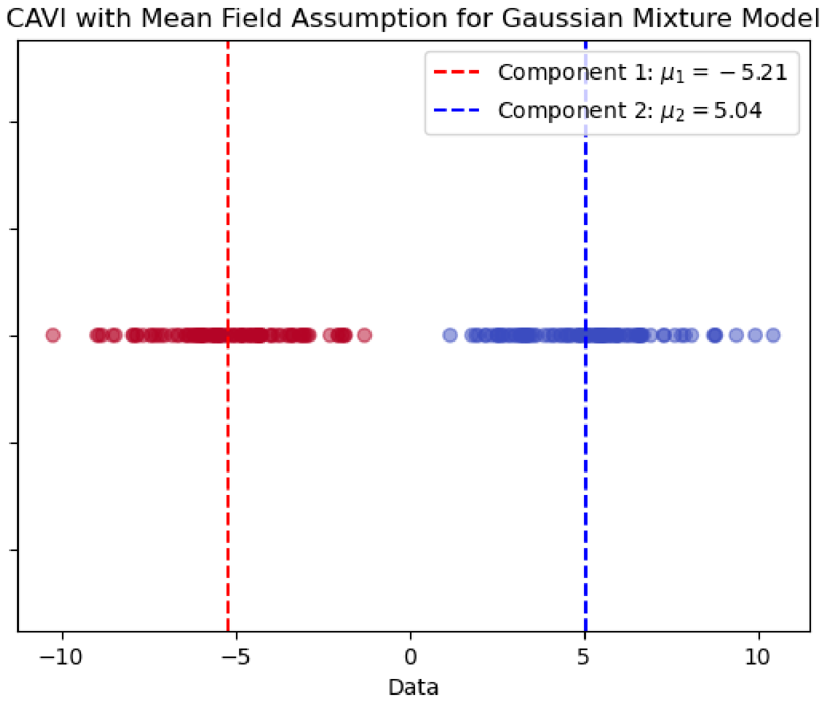


Figure 2: Synthetic data points and the estimated means of each Gaussian component of the Gaussian Mixture Model using the CAVI algorithm with the mean field assumption. The data points are colored based on their responsibilities to the first Gaussian component (blue-to-red gradient), and the estimated means for each Gaussian component are represented as vertical dashed lines (red for Component 1 and blue for Component 2).

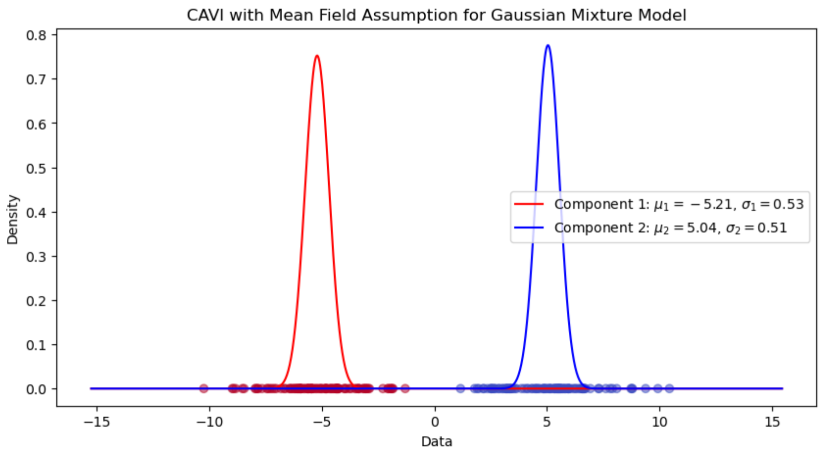


Figure 3: Synthetic data points and the estimated Gaussian distributions for each component of the Gaussian Mixture Model using the CAVI algorithm with the mean field assumption. The data points are colored based on their responsibilities to the first Gaussian component (blue-to-red gradient), and the Gaussian components are represented as continuous lines (red for Component 1 and blue for Component 2). The estimated means and standard deviations for each Gaussian component are provided in the legend.

5 Stochastic Gradient Descent

5.1 Introduction

Gradient descent is one of the most popular algorithms to perform optimisation and by far the most common way to minimise loss functions in neural networks [40]. Gradient descent allows us to minimise an objective function $J(\theta)$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters.

Stochastic gradient descent can be considered a stochastic approximation of gradient descent optimisation, since we aim to approximate the true gradient of the objective function using a noisy estimate of the gradient. This has the benefit of being computationally cheaper than gradient descent which finds the true gradient by using the entire dataset. Stochastic gradient descent provides unbiased estimates of the true gradient, meaning the expectation of our gradient estimate will converge to the true gradient over multiple batches and runs. Work on Stochastic gradient descent is extensive, and the ideas can be traced back to robbins-monro in the 1950s [41].

These methods can be fully applied to VI, where our goal is to optimise the negative ELBO as our function $J(\theta)$. Minimising $J(\theta)$ allows us to find the best variatational distribution to approximate the true distribution over our family of variational densities.

5.2 Gradient Descent

In the context of VI, the goal is to approximate a complex posterior distribution. Gradient descent can be used to optimise the parameters of the simpler distribution to better match the posterior distribution, as measured by a loss function such as the negative ELBO (equation 32). The gradient of the negative ELBO with respect to the parameters of the simpler distribution is computed, and the parameters are updated in the direction of the negative gradient. This process is repeated iteratively until convergence, at which point the optimised simpler distribution provides an approximate solution to the posterior distribution.

Batch gradient descent (BGD) computes the gradient of the loss function with respect to θ for the entire training dataset, and makes the update.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t), \tag{50}$$

where the learning rate η determines the size of the steps we take each iteration to reach a local minimum. Too high of a learning rate may mean we step past the local minimum resulting in missing a potentially optimal solution. Too low of a learning rate can also be bad as it may result in slow convergence and increase the likelihood of converging to a suboptimal minimum.

BGD has the limitation where we must load our entire dataset into memory which is not always possible. We also are unable to feed new data into the model.

```
for i in range(epochs):
    params_grad = evaluate_gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Here we evaluate the gradient with respect to the parameters given all the data, and update our parameters in the direction of the negative gradient with a step size defined by the learning rate. This process is repeated with a predefined number of epochs.

5.3 Stochastic Gradient Descent

Stochastic Gradient Descent instead performs a parameter update for each training datapoint, x_i, y_i. In practice, this is usually performed with mini batches of a selected size to find a balance with computational efficiency and convergence properties. Mini batches can also take advantage of parallel processing by processing multiple data points simultaneously before making the parameter update.

$$\theta = \theta - \eta \nabla_\theta J(\theta; x_i; y_i) \tag{51}$$

Since we update for each x_i, y_i, this algorithm is often much faster to converge since BGD will compute the gradient over similar and redundant data before updating the parameters. The frequent updates of SGD with a higher variance gradient estimate can cause the loss function to fluctuate heavily. To reduce the fluctuations and risk of stepping past a minimum, we can slowly decrease the learning rate as training goes on.

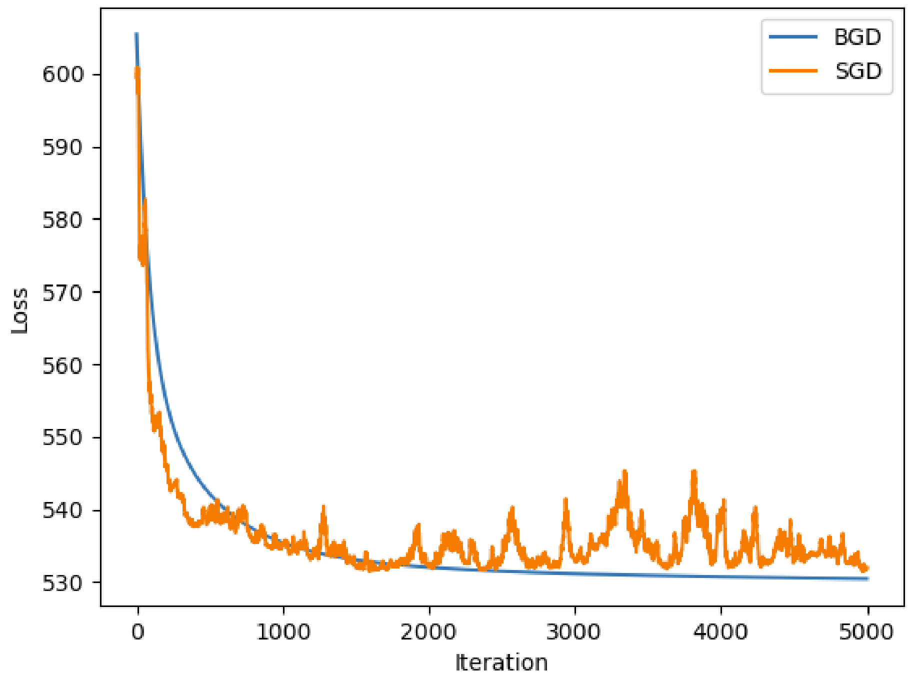


Figure 4: Fluctuation of SGD for Boston Housing dataset [42] prediction model

```
for i in range(epochs):
    np.random.shuffle(data)
    for example in data:
        params_grad = evaluate_gradient(loss_function, example, params)
        params = params - learning_rate * params_grad
```

The key difference here is that we randomly sample from the data and update our parameters for each datapoint.

Figure 4 shows the fluctuation of SGD over 5000 iterations, we however note that in comparison to BGD, we only evaluate 32 data points at each iteration so the convergence is much faster. This will only improve as the dataset scales up. Decreasing the learning rate may help reduce fluctuation however may mean the model will converge too slowly with big data. Algorithms such as the adam optimiser or adagrad use techniques of adaptive learning rate to automatically adjust the learning rate during training based on the observed behavior of the loss function.

5.4 Adaptive Learning Rate Methods

Adaptive learning rate methods are an extension to gradient descent which adapts the learning rate for each parameter based on the historical gradient information. There is a disadvantage of SGD that it scales the gradient uniformly in all directions. This may lead to poor performance as well as limited training speed when the training data are sparse. To address this problem, recent work has proposed a variety of adaptive methods that scale the gradient by square roots of some form of the average of the squared values of past gradients. Popular methods include adam optimiser which is widely used in machine learning applications, or AdaGrad [43].

AdaGrad adapts the learning rate for each parameter based on the accumulated squared gradients. The key idea is to increase the learning rate for infrequently updated parameters and decrease it for frequently updated ones [44].

The AdaGrad update rule can be written as:

$$g_t = \nabla L(\theta_t)$$

$$G_t = G_{t-1} + g_t \odot g_t$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

The algorithm maintains a running sum of squared gradients for each model parameter and uses this information to adapt the learning rate for each parameter individually. "$\odot$" refers to the element wise product, which makes it possible to update each parameter with its own learning rate.

Adam is an extension of the ideas from AdaGrad and RMSProp (Root Mean Square Propagation). It uses both the first moment (mean) and the second moment (uncentered variance) of the gradients to adapt the learning rate for each parameter. The key idea is to combine the benefits of both momentum-based methods and adaptive learning rates [45].

The Adam update rule can be written as:

$$g_t = \nabla L(\theta_t)$$

$$\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t \odot g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \odot \hat{m}_t
\end{aligned}$$

In this algorithm, β_1 and β_2 are hyperparameters which are used to control the exponential decay rates of the first and second moment estimates respectively. β_1 controls the decay rate for the first moment estimate, which influences the momentum term in the update rule - Momentum helps by accumulating gradients over multiple iterations and using the running average of past gradients to update the model parameters, relating to the idea of momentum in physics. β_2 controls the decay rate for the second moment estimate, relating to the uncentered variance of the gradient. In their 2015 conference paper introducing the Adam optimiser, Kingma, Ba, recommends settings for the tested machine learning problems are $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$. These recommended settings are closely followed by implementations in various popular deep learning libraries [46].

Both AdaGrad and Adam optimisers adjust the learning rate for each parameter during the optimisation process. AdaGrad does this by accumulating the squared gradients [40], while Adam extends this idea by considering both the first and second moments of the gradients [47]. These adaptive learning rate methods can lead to faster convergence and improved performance in many optimisation problems.

5.4.1 Example

In this example, we construct a neural network and train our parameters using SGD, Adam, and AdaGrad. We use the California Housing dataset, where the target is to predict the average house value [48].

Our neural network consists of an input layer, one hidden layer, and an output layer. The input layer takes in the 8 features as input. The input is passed through the first fully connected layer, which computes the weighted sum of the inputs and adds a bias term. The ReLU activation function is then applied and we pass this result through the second fully connected layer to compute the final output, which is a single continuous value representing the predicted house price.

We therefore have many weights and biases in the fully connected layers we want to train to minimise our loss function. We use the Mean Squared Error (MSE) loss. Given n samples, the true target values y_i and the predicted values $\hat{y}_i$, the MSE loss can be written as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

We create identical neural networks for each method of optimisation and plot the results over 10 epochs, with a batch size of 32.

We see from the results that the adam optimiser converges the fastest during training. This could be due to its adaptive learning rate and momentum qualities. The adam optimiser also has the least variance during training, which could be due to the adaptive learning rates for each parameter helping the algorithm to make more fine tuned adjustments.

5.4.2 Conclusion

When implementing gradient descent methods, we must be careful when setting our learning rate. A constant learning rate may be too small, causing the optimisation algorithm to take small steps and converge very slowly towards the minimum or too large, causing the optimizer to overshoot the minimum and oscillate around it. Adaptive learning rate methods for SGD offer significant improvements in the optimisation process, leading to faster and more stable convergence in various machine learning tasks. These methods adapt the learning rate for each parameter individually based on their historical gradients allowing for more fine-grained and efficient updates during the training process, leading to faster and more stable convergence.

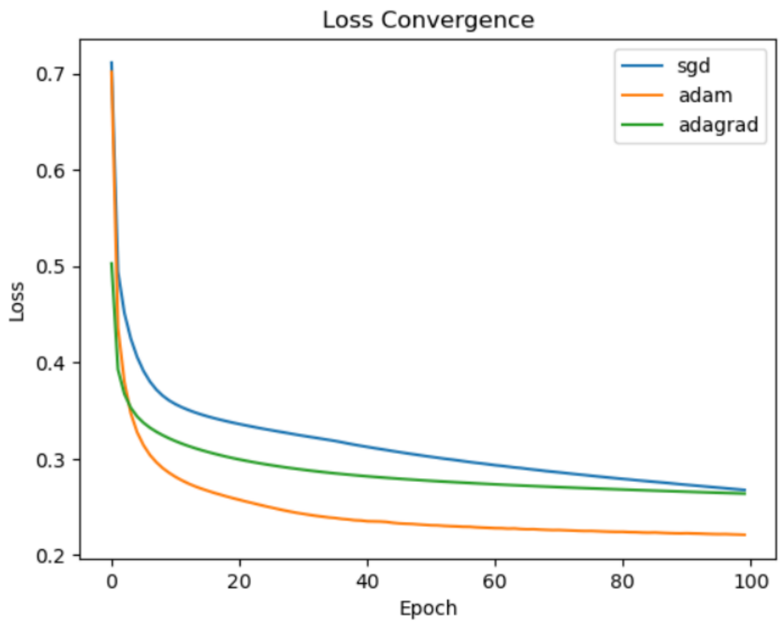


Figure 5: The convergence of the training loss over epochs for each optimizer (SGD, Adam, and Adagrad)

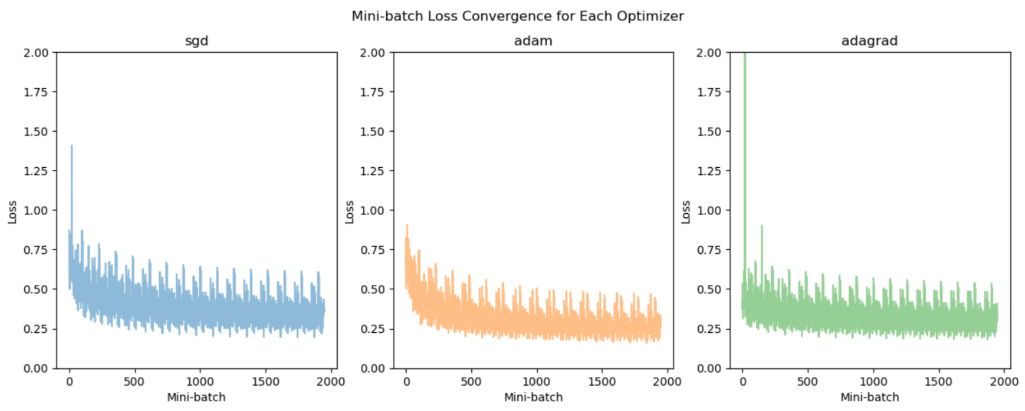


Figure 6: The mini-batch loss for each optimizer (SGD, Adam, and Adagrad) over a selected range of mini-batches.

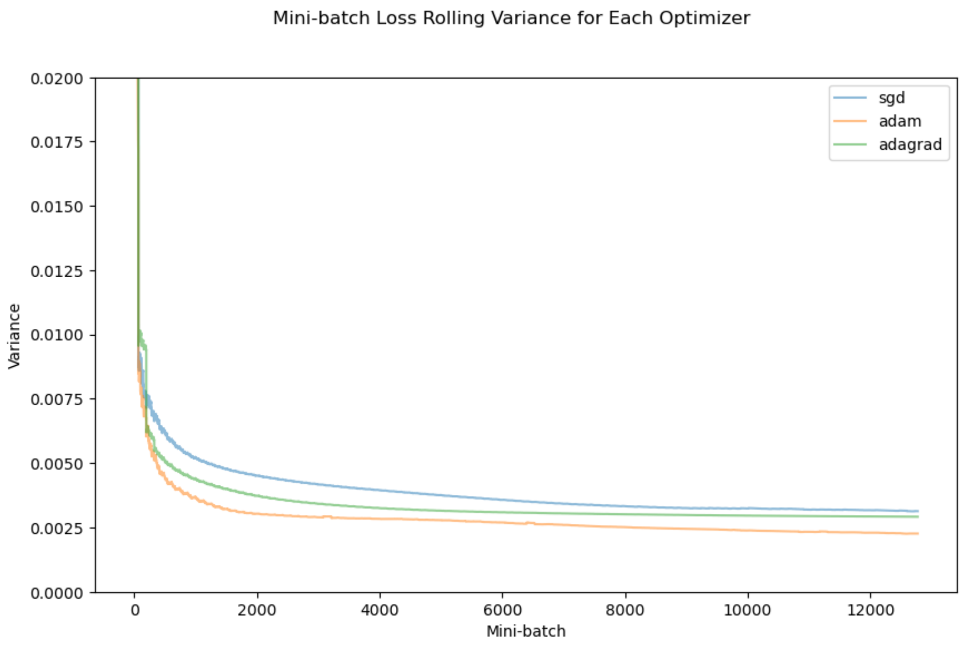


Figure 7: The rolling variance of the mini-batch loss for each optimiser (SGD, Adam, and Adagrad) during training.

5.5 Variance Reduction

5.5.1 Stochastic Optimisation

Stochastic optimisation is a family of optimisation techniques that trade off computational efficiency and statistical precision to solve large-scale optimisation problems more effectively.

The main idea behind stochastic optimisation in SVI is to estimate the gradient of the ELBO using a random subset of the data, rather than calculating the full gradient using the entire dataset. This subset of data called a mini batch, can be thought of as a noisy but computationally cheaper approximation of the true gradient [49]. By updating the variational parameters using these noisy gradient estimates, SVI can achieve significant speed-ups and improvement for scalability [50].

In section 6.2, we studied natural gradients. The natural gradient accounts for the geometric structure of probability parameters and has good theoretical properties. Stochastic variational inference can use the natural gradient in a stochastic optimisation algorithm. Stochastic optimisation algorithms follow noisy but cheap-to-compute gradients to reach the optimum of an objective function. (In the case of the ELBO, stochastic optimisation will reach a local optimum). Robbins and Monro (1951) proved results implying that optimisation algorithms can successfully use noisy, unbiased gradients, as long as the step size sequence satisfies certain conditions [41].

The Robbins and Monro conditions state that the learning rates should satisfy:

$$\sum_{t=1}^{\infty} \eta_t = \infty \tag{52}$$

$$\sum_{t=1}^{\infty} \eta_t^2 < \infty \tag{53}$$

This guarantees that every point in the parameter space can be reached, while the gradient noise decreases quickly enough to ensure convergence.

The optimal choice of the size of the mini batch emerges from a trade-off between the computational overhead associated with processing a mini-batch, such as performing inference over global parameters (favoring larger mini-batches which have lower gradient noise, allowing larger learning rates), and the cost of iterating over local parameters in the mini-batch (favoring small mini-batches). Additionally, this trade off is also affected by memory structures in modern hardware such as GPUs. The data in the minibatch must be drawn uniformly at random.

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. Algorithms which use adaptive learning rates were described in section 6.1.

5.5.2 Stochastic Variance Reduced Gradient (SVRG)

One practical issue for SGD is that in order to ensure convergence the learning rate η_t has to decay to zero [41], where η_t is an adaptive learning rate at time t. This leads to slower convergence. The need for a small learning rate is especially due to the noise incorporated by using a mini batch of samples to approximate the full gradient, and stable convergence can be reached by effectively reducing this noise [51]. SGD has been viewed in the machine learning community as an ideal optimisation algorithm, due to its low per-iteration cost. However, [52] argues that SGD suffers from the adverse effect of noisy gradient estimates. This prevents it from converging to the solution when fixed stepsizes are used and leads to a slow, sublinear rate of convergence when a diminishing stepsize sequence η_t is employed. We are therefore interested in finding a way to reduce the noise, allowing a higher learning rate to improve convergence speed.

Different variants of SGD have been developed to reduce the variance, such as the stochastic average gradient (SAG) method [53], which randomly updates the gradient of one component function while keeping the others unchanged [54]. This has lead to an active area of research into improvements on variance reduction techniques.

Johnson and Zhang [55] presented a stochastic variance reduced gradient (SVRG) method, which selects a stochastic gradient with low variance as the unbiased estimate of the full gradient. One advantage of the SVRG method is that it removes the memory requirement of all gradients. This method has been widely used in large-scale machine learning applications [56].

Algorithm 1 Stochastic Variance Reduced Gradient (SVRG) Algorithm

```
1: **Input:** Parameters $\theta_0$, learning rate $\eta$, loss function $J(\theta)$, number of
   epochs $T$, number of iterations per epoch $m$
2: for $t = 0, 1, \ldots, T-1$ do
3:     Compute the full gradient $\nabla J(\theta_t)$
4:     Set $\theta_{t,m} \leftarrow \theta_t$
5:     for $k = 0, 1, \ldots, m-1$ do
6:         Choose a random index $i_k \in \{1, \ldots, n\}$ uniformly at random
7:         Compute the stochastic gradient $g_{i_k}(\theta_{t,m})$ and $g_{i_k}(\theta_t)$
8:         Update the parameters:
               $\theta_{t,m+1} \leftarrow \theta_{t,m} - \eta \left( g_{i_k}(\theta_{t,m}) - g_{i_k}(\theta_t) + \nabla J(\theta_t) \right)$
9:     end for
10:    Update the parameters for the next epoch:
               $\theta_{t+1} \leftarrow \theta_{t,m}$
11: end for
12: Output: Parameters $\theta_T$
```

In SVRG, the parameter updates are based on a combination of stochastic gradients and full gradients. For each epoch, the full gradient is computed once and the inner loop updates the parameters using a combination of stochastic gradients and the full gradient. The temporary parameters serve as an intermediate set of parameters used within each epoch to perform the updates, allowing the algorithm to maintain a reference point of the current parameters θ_t while making updates based on the stochastic gradients and full gradients. The SVRG method does not require the storage of gradients, and thus is more easily applicable to complex problems such as neural network learning [55].

5.5.3 Non-uniform Sampling

Non-uniform sampling can be used to select mini-batches with a lower gradient variance. Although effective, these methods are not always practical, as the computational complexity of the sampling mechanism relates to the dimensionality of model parameter. Approaches such as stratified sampling or cluster based sampling using methods such as k-means have been shown to reduce variance and can also be used for learning on imbalanced data [15].

[57] provides great insight into how we might be able to combine approaches for reducing stochastic variance in minibatches to greatly accelerate training.

5.6 Natural Gradients

Natural gradients, take the information geometry of the model into account. They are obtained by pre-multiplying the gradient with the inverse Fisher information matrix. The natural gradient of a function accounts for the information geometry of its parameter space, using a Riemannian metric to adjust the direction of the traditional gradient [15].

The problem with Euclidean distance is especially clear in our setting, where we are trying to optimise an objective with respect to a parameterized probability distribution $q(\beta|\lambda)$. When optimizing over a probability distribution, the Euclidean distance between two parameter vectors λ and λ' is often a poor measure of the dissimilarity of the distributions $q(\beta|\lambda)$ and $q(\beta|\lambda')$. For example, suppose $q(\beta)$ is a univariate normal and λ is the mean μ and scale σ. The distributions $N(0, 10000)$ and $N(10, 10000)$ are almost indistinguishable, and the Euclidean distance between their parameter vectors is 10. In contrast, the distributions $N(0, 0.01)$ and $N(0.1, 0.01)$ barely overlap, but this is not reflected in the Euclidean distance between their parameter vectors, which is only 0.1. The natural gradient corrects for this issue by redefining the basic definition of the gradient [58].

The natural gradient becomes quite involved from linear algebra. For a more detailed mathematical derivation, see [59]

5.7 Natural Gradient Example

In this example we train a neural network to compare the performance of vanilla stochastic gradient descent with gradient descent using the natural gradient. We evaluate the training process with respect to training loss and test accuracy over the iterations.

The MNIST dataset is a common example used in ML. MNIST is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST database contains 60,000 training images and 10,000 testing images [60], as shown in figure 8.

We implement a simple two-layer fully connected network which we train with the cross Entropy loss function, which is commonly used for classification tasks. For reference to neural network basics, see [61], and for the cross entropy loss function see [62].

```python
import torch
import torchvision
import torch.nn as nn
from torchvision import transforms
from torch.utils.data import DataLoader
from tqdm import tqdm
import matplotlib.pyplot as plt

# Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))])

train_data = torchvision.datasets.MNIST(root='./data', train=True,
    download=True, transform=transform)
```

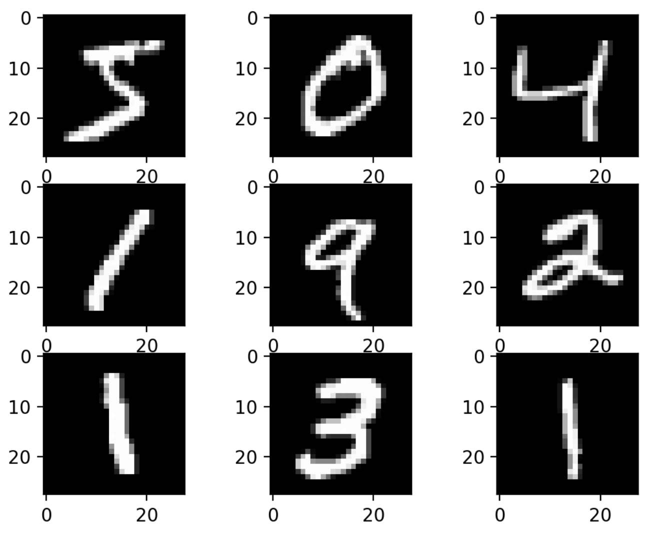


Figure 8: 9 handwritten digits from the MNIST dataset. Each digit is represented by 28x28 pixels.

```
test_data = torchvision.datasets.MNIST(root='./data', train=False,
    download=True, transform=transform)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=128, shuffle=False)

# Define neural network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

This part of the code imports the necessary libraries and sets up the MNIST dataset and a simple neural network. We load the dataset and normalise the images, before setting up batch dataloaders to efficiently handle the in mini batches. The neural network contains two fully connected layers. The input images are flattened and passed through fc1, followed by a ReLU activation function. The output of this activation is then passed through the fc2 layer to produce the final class probabilities.

```
def train(model, optimiser, dataloader, fisher_matrix=None):
model.train()
total_loss = 0
progress_bar = tqdm(dataloader, desc="Training", ncols=100)
for data, target in progress_bar:
    optimiser.zero_grad()
    output = model(data)
    loss = nn.functional.cross_entropy(output, target)

    if fisher_matrix is not None:
        # Compute gradients using diagonal Fisher matrix
        grad_fisher = torch.autograd.grad(loss, model.parameters(),
            create_graph=True)
        grad_flat = torch.cat([grad.reshape(-1) for grad in grad_fisher])
        fisher_vec_prod = fisher_matrix * grad_flat
        fisher_grads = [fisher_vec_prod[i:i+p.numel()].reshape(p.shape) for
            i, p in enumerate(model.parameters())]
        for p, fisher_grad in zip(model.parameters(), fisher_grads):
            if p.grad is None:
                p.grad = torch.zeros_like(p)
            with torch.no_grad():
                p.grad.copy_(fisher_grad)
    else:
        loss.backward()

    optimiser.step()
    total_loss += loss.item()

    # Update progress bar
    progress_bar.set_postfix(loss=loss.item())
return total_loss / len(dataloader)

# Testing function
def test(model, dataloader):
    model.eval()
    correct = 0
    with torch.no_grad():
        for data, target in dataloader:
            output = model(data)
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    return correct / len(dataloader.dataset)
```

This section of the code defines the train and test functions. We keep track of training with a progress bar. For the natural gradient calculations, we need to use the fisher information matrix to find natural gradients by the definition of natural gradients.

```
# Training and evaluation
learning_rate = 0.01
epochs = 10

# Vanilla SGD
model_vanilla = Net()
optimiser_vanilla = torch.optim.SGD(model_vanilla.parameters(),
    lr=learning_rate)

# Natural Gradient with Fisher matrix
model_natural = Net()
optimiser_natural = torch.optim.SGD(model_natural.parameters(),
    lr=learning_rate)

losses_vanilla = []
losses_natural = []
accuracies_vanilla = []
accuracies_natural = []

# Create a Fisher mini-batch
```

```
fisher_batch_size = 512
fisher_loader = DataLoader(train_data, batch_size=fisher_batch_size,
     shuffle=True)
fisher_data, fisher_target = next(iter(fisher_loader))

for epoch in range(epochs):
    print(f"Epoch {epoch + 1}:")

    # Compute diagonal Fisher matrix for the current epoch
    fisher_matrix = []
    for i in range(fisher_batch_size):
        data, target = fisher_data[i:i+1], fisher_target[i:i+1]
        output = model_natural(data)
        loss = nn.functional.cross_entropy(output, target)
        grad_fisher = torch.autograd.grad(loss,
            model_natural.parameters(), create_graph=True)
        grad_flat = torch.cat([grad.reshape(-1) for grad in grad_fisher])
        fisher_matrix.append(grad_flat ** 2)
    fisher_matrix = sum(fisher_matrix) / fisher_batch_size

    train_loss_vanilla = train(model_vanilla,
        optimizer_vanilla, train_loader, None)
    train_loss_natural = train(model_natural,
        optimizer_natural, train_loader, fisher_matrix)

    # Test models
    test_acc_vanilla = test(model_vanilla, test_loader)
    test_acc_natural = test(model_natural, test_loader)

    accuracies_vanilla.append(test_acc_vanilla)
    accuracies_natural.append(test_acc_natural)

    print(f"Epoch {epoch + 1}:")
    print(f"Vanilla SGD - Loss: {train_loss_vanilla:.4f},
        Test Accuracy: {test_acc_vanilla:.4f}")
    print(f"Natural Gradient - Loss: {train_loss_natural:.4f},
        Test Accuracy: {test_acc_natural:.4f}")
    print()
```

Finally, we initialise our models and compute the Fisher information matrix for the natural gradient method model. To save on memory requirements, we approximate the Fisher information matrix by using only the diagonal elements of the matrix. Using the full Fisher matrix can provide a more accurate approximation, but does not scale well since the dimensions of the matrix is proportional to the square of the number of parameters in our model. Since this is a two fully connected layer neural network, the memory requirements to store and manipulate the full matrix are too great for any feasable implementation, so using the diagonal Fisher matrix simplifies the computational requirements. A downside to using the diagonal Fisher matrix is that is makes additional assumptions that all of the parameters are independant, and this assumption may not be correct and will make our natural gradients less accurate.

This assumption is reflected in our overall performance.

```
Epoch 1:
Vanilla SGD - Loss: 0.7943, Test Accuracy: 0.8918
Natural Gradient - Loss: 2.3297, Test Accuracy: 0.1212

Epoch 2:
Vanilla SGD - Loss: 0.3706, Test Accuracy: 0.9101
Natural Gradient - Loss: 2.2917, Test Accuracy: 0.1426

Epoch 3:
Vanilla SGD - Loss: 0.3171, Test Accuracy: 0.9174
Natural Gradient - Loss: 2.2554, Test Accuracy: 0.1723

...

Epoch 9:
Vanilla SGD - Loss: 0.2010, Test Accuracy: 0.9456
Natural Gradient - Loss: 2.0682, Test Accuracy: 0.4197

Epoch 10:
Vanilla SGD - Loss: 0.1897, Test Accuracy: 0.9473
Natural Gradient - Loss: 2.0407, Test Accuracy: 0.4585

---
```

```
Vanilla SGD - Final Test Accuracy: 0.9473
Natural Gradient - Final Test Accuracy: 0.4585
```

As we see from our outputs, using the natural gradient has not performed well. For a complex models with many parameters, we were unable to efficiently compute the Fisher information matrix and instead had to approximate, which led to poor convergence. The natural gradient appears to perform well in simpler examples, but lack efficient scalability.

For Variational Inference where we may be highly concerned with scalability, adding in extra computational complexity and large memory requirements to compute and score the information matrix may not be a wise approach, and approximations using the diagonal seem to result in poor gradient steps in this example.

5.8 Stochastic VI

Modern data analysis requires computation with big data, i.e. huge amounts of data with a growing number of features, often difficult to manage. This highlights the usefulness of scalability in algorithms.

The scalability issue when using classical approaches such as Coordinate ascent variational inference (CAVI) arises from the computational complexity associated with evaluating the full gradient of the ELBO. Optimizing the ELBO requires calculating the gradient with respect to each variational parameter for every data point [49], which can be computationally expensive and for large scale datasets. CAVI updates one variational parameter at a time which can lead to slow convergence, particularly when the number of parameters is large [1]. This iterative, coordinate-wise optimisation approach can be inefficient.

Stochastic Variational Inference (SVI) is often a better approach to address issues with scalability. SVI leverages stochastic optimisation techniques to reduce the computational complexity associated with optimizing the ELBO. By enabling more efficient optimisation and offering better convergence properties, SVI provides a more suitable solution for tackling large-scale Bayesian inference problems [50].

By using the ideas put forward in this chapter we are able to have an efficiently scalable algorithm, following the basic recipe of VI with a strong emphasis on modern stochastic optimisation techniques [49].

6 Reparameterization Trick

6.1 Introduction

The reparameterization trick is a powerful technique used in Variational Inference to enable efficient and low-variance gradient estimation of the Evidence Lower Bound. The optimisation involves taking gradients of the ELBO with respect to the variational parameters [63].

Obtaining accurate gradient estimates can be challenging in certain situations:

- The approximating distribution is highly expressive: When the approximating distribution is flexible or has a large number of parameters, it can capture complex structures in the true posterior distribution. However, this may lead to many local optima.
- The model is highly expressive: If the model itself is complex or has a large number of parameters (e.g. deep neural networks), it can capture intricate patterns in the data. However, this may also make the optimisation problem more challenging due to the increased complexity of the joint distribution of the model parameters and the data, making accurately estimating the gradients of the ELBO difficult.
- Noisy samples: When estimating gradients of the ELBO, we often rely on Monte Carlo (MC) sampling to approximate the expectations involved. However, the sampling process can introduce variability or noise in the gradient estimates, especially when the number of samples is small or the true posterior distribution is complex i.e. non Gaussian, leading to high variance gradient estimates.

The reparameterization trick addresses these challenges, paving the way for more effective optimisation using gradient-based methods [10], such as stochastic gradient descent and its variants.

The central idea of the reparameterization trick is to transform the random variables in the model by expressing them as deterministic functions of some other random variables with simpler, fixed distributions. This transformation allows us to compute gradients of the ELBO with respect to the variational parameters more efficiently and with lower variance. The reparameterization trick is especially useful as it enables backpropagation through the sampling process which is a method used for training deep neural networks [64].

A key advantage of the reparameterization trick is its ability to reduce the variance of gradient estimates, leading to faster and more stable convergence during optimisation. The trick has been key in the development and success of variational autoencoders (VAEs), a popular type of generative model.

6.2 Definition

Consider a probabilistic model with observed data X and latent variables z. We want to optimise the Evidence Lower Bound (ELBO) with respect to the variational parameters θ of the approximating distribution $q(z|\theta)$. To do this, we need to compute the gradient however computing this gradient directly can be challenging due to the expectation with respect to the variational distribution $q(z|\theta)$. This expectation involves an integral over the latent variables z, and in many cases, it cannot be computed analytically.

The reparameterization trick can be defined as follows:

1. Introduce an auxiliary random variable ϵ with a fixed distribution $p(\epsilon)$, typically a standard normal distribution.
2. Express the latent variable z as a deterministic function $f(\epsilon, \theta)$ of ϵ and θ: $z = f(\epsilon, \theta)$.
3. Rewrite the expectation in the ELBO using the reparameterized latent variable:

   $$\mathbb{E}_{q(z|\theta)}[\log p(X|z)] - D_{KL}(q(z|\theta)||p(z)) = \mathbb{E}_{p(\epsilon)}[\log p(X|f(\epsilon,\theta))] - D_{KL}(q(z|\theta)||p(z))$$

   The KL term of the ELBO remains unchanged.

This transforms the sampling process of the latent variable into a deterministic function of an auxiliary random variable and the variational parameters, enabling easy gradient computation [15].

This reparameterization is useful for our case since it can be used to rewrite an expectation with respect to $q_\phi(z|\theta)$ such that the Monte Carlo estimate of the expectation is differentiable w.r.t. ϕ [63].

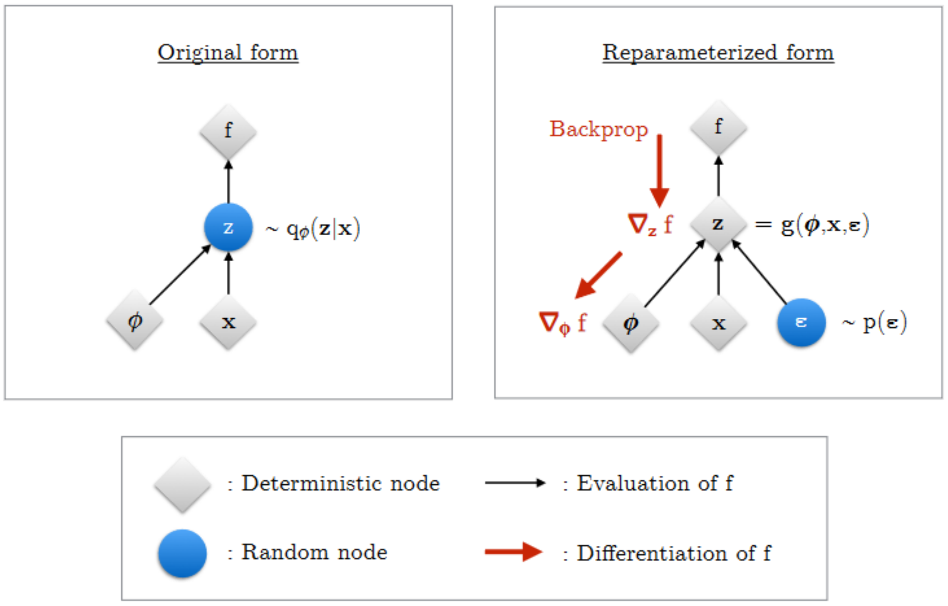


Figure 9: Reparameterization Trick overview. Source: [65]

A proof is as follows.

Given the deterministic mapping $z = g_\phi(\epsilon, x)$ we know that

$$q_\phi(z|x) \prod_i dz_i = p(\epsilon) \prod_i d\epsilon_i.$$

Therefore,

$$\int q_\phi(z|x) f(z) dz = \int p(\epsilon) f(z) d\epsilon = \int p(\epsilon) f(g_\phi(\epsilon, x)) d\epsilon.$$

It follows that a differentiable estimator can be constructed:

$$\int q_\phi(z|x) f(z) dz \simeq \frac{1}{L} \sum_{l=1}^{L} f(g_\phi(x, \epsilon^{(l)}))$$

where $\epsilon^{(l)} \sim p(\epsilon)$. [63]. Since $g_\phi(\epsilon, x)$ is a deterministic function, its gradients with respect to ϕ can be computed directly.

For more detail on the reparameterization trick, see papers on variational auto-encoders (VAEs) [66] [67] [63]

7 Automatic Differentiation VI

7.1 Introduction

A limitation of the VI methods covered previously is the difficulty of implementation. It requires a level of statistical understanding to be able to efficiently and confidently implement a model. Implementation can be challenging for a number of reasons such as selecting appropriate variational families, optimizing variational parameters, and calculating gradients of the objective function. These difficulties often require significant expertise and can be a barrier for practitioners who wish to apply VI to their problems.

Automatic Differentiation VI proposes a black box style approach to VI enabling easier implementation by automating variational family selection and using automatic differentiation methods to efficiently evaluate the partial derivative of a function.

7.2 Computational Differentiation

The primary challenges in computing derivatives for computers arise from:

- Complexity of functions: Functions can have complicated algebraic expressions, making it difficult to determine their derivatives analytically.
- Numerical precision: Accurate calculation of derivatives often demands high numerical precision, which can be computationally demanding or subject to round-off errors. round-off errors can occur due to inexactness in the representation of real numbers.
- Computational efficiency: Some methods may be computationally expensive, especially for high-dimensional functions or when higher-order derivatives are needed.

There are three main methods computers might use for calculating derivatives, Symbolic differentation, numerical differentiation, or automatic differentiation [68].

7.2.1 Symbolic Differentiation

Symbolic differentiation is based off the logic that a mathematical function can be broken down into the composition of well-known simple functions where the chain the rule can applied repeatedly. A symbolic differentiation program will find the derivative of a given formula producing a new formula as output.

A divide and conquer strategy for symbolic differentiation can be defined as follows [69]:

1. If the problem to be solved is an easy problem, solve it at once.
2. If the problem to be solved is a hard problem,
    (a) Break the problem into smaller subproblems.
    (b) Use the problem-solver itself, recursively, to solve the subproblems.
    (c) Combine the solutions of the subproblems to make a solution for the larger problem.

Symbolic differentiation is used in programs such as Matlab or Mathematica [70] or implemented in popular programming languages with libraries [71], and is essential to computer algebra systems. Symbolic differentiation can provide exact expressions but may lead to large expressions, also known as "expression swell". Expression swell happens when the numbers or expressions grow exponentially in size [72].

7.2.2 Numerical Differentiation

Numerical differentiation is a set of techniques used to approximate the derivatives of a function using data points or values of the function. The most common method in numerical differentiation is finite difference approximations, which involves calculating the difference between function values at two nearby points and dividing by the distance between those points.

A simple two point estimation is to compute the slope of a line through the points, for this we can use Newton's difference quotient as $h \to 0$

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}$$

for a small h.

Higher order methods for approximating the derivative exist. Using this equation for numerical differentiation is known as the forwards finite difference method. Alternative methods such as the central (or symmetric) finite difference requires two evaluations of the function per differentiation point.

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h}$$

Numerical differentiation methods are straightforward to implement and computationally efficient. These methods do not require an analytical expression of the function, making them suitable for many cases. The disadvantages of this method relates to errors which are introduced in calculation, such as truncation error and round-off error [73]. Truncation error arises due to the approximation in numerical differentiation methods. For example, in finite differences, the derivative is approximated using a finite difference over a small interval h. Round-off error occurs due to the limited precision of floating-point arithmetic in computers. When calculating differences between function values and dividing by small step sizes, round-off errors can accumulate and affect the accuracy of the derivative approximation [74].

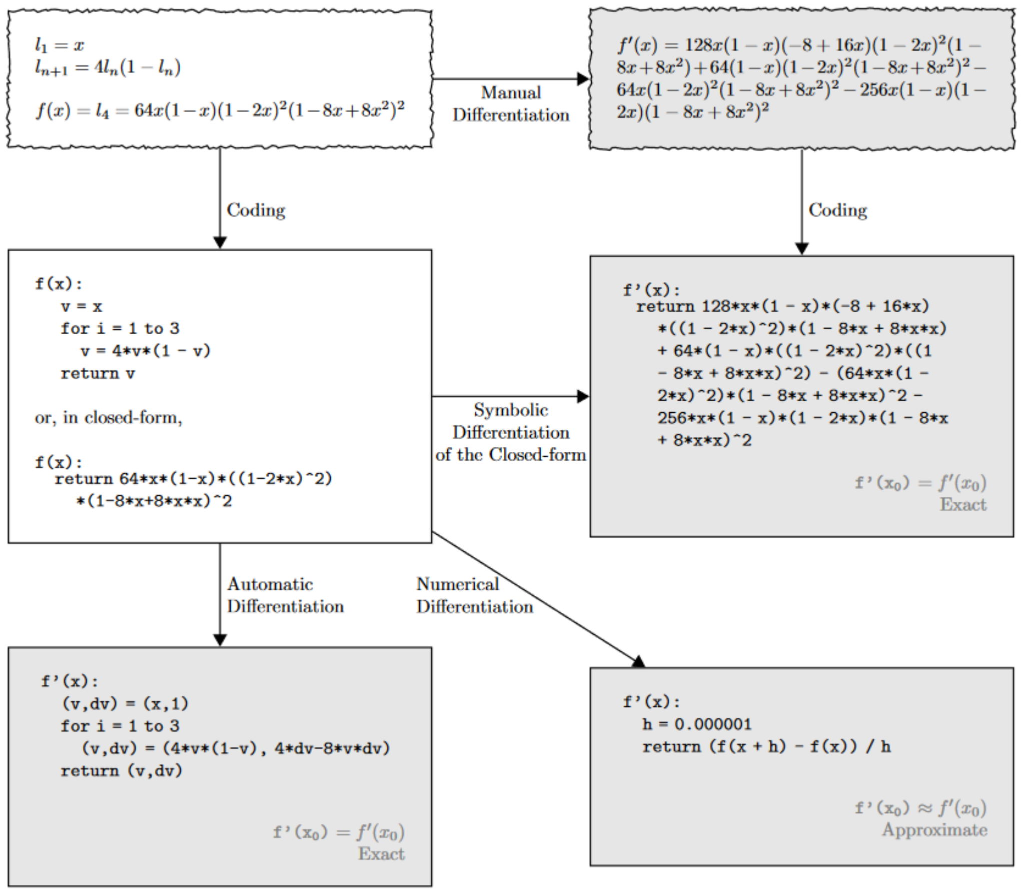
The errors can be reduced with careful choices of the order of the approximation and optimal choice of step size h, however, they will always be present when using numerical differentiation methods [74].

7.3 Automatic Differentiation

Automatic Differentiation (AD) uses symbolic rules for differentiation which are more accurate than finite difference approximations. Unlike symbolic differentiation, automatic differentiation evaluates expressions numerically rather than carrying out large symbolic computations. In other words, automatic differentiation evaluates derivatives at particular numeric values and does not construct symbolic expressions for derivatives [75].

- Forward mode AD evaluates a numerical derivative by performing elementary derivative operations concurrently with the operations of evaluating the function itself.
- Reverse mode AD extends the sequence of operations used in forward mode to allow the calculation of a gradient through a reverse traversal of these operations. Backpropagation, an algorithm commonly employed in training neural networks, is a particular instance of reverse AD.

All numerical computations are ultimately compositions of a finite set of elementary operations for which derivatives are known, and combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition [33].



$$l_1 = x$$
$$l_{n+1} = 4l_n(1 - l_n)$$
$$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

$$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$$

```
f(x):
  v = x
  for i = 1 to 3
    v = 4*v*(1 - v)
  return v
```

or, in closed-form,

```
f(x):
  return 64*x*(1-x)*((1-2*x)^2)
    *(1-8*x+8*x*x)^2
```

```
f'(x):
  return 128*x*(1 - x)*(-8 + 16*x)
    *((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
    + 64*(1 - x)*((1 - 2*x)^2)*((1
    - 8*x + 8*x*x)^2) - (64*x*(1 -
    2*x)^2)*(1 - 8*x + 8*x*x)^2 -
    256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
    + 8*x*x)^2
```

```
f'(x):
  (v,dv) = (x,1)
  for i = 1 to 3
    (v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
  return (v,dv)
```

```
f'(x):
  h = 0.000001
  return (f(x + h) - f(x)) / h
```

Manual Differentiation
Coding
Coding
Symbolic Differentiation of the Closed-form
Automatic Differentiation
Numerical Differentiation
f'(x₀) = f′(x₀) Exact
f'(x₀) = f′(x₀) Exact
f'(x₀) ≈ f′(x₀) Approximate

Figure 10: The range of approaches for differentiating mathematical expressions and computer code. Source: [33]

To demonstrate the methods of automatic differentiation, we adopt notation used by Griewank and Walther (2008) [76], where a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is constructed using intermediate variables v_i such that

- variables $v_{i-n} = x_i, i = 1, \ldots, n$ are the input variables,

- variables $v_i, i = 1, \dots, l$ are the working (intermediate) variables
- variables $y_{m-i} = v_{l-i}, i = m-1, \dots, 0$ are the output variables.

This will help us to understand the trace when applying AD methods

7.3.1 Forwards AD

Example: $y = f(x_1, x_2) = x_1 x_2 - sin(x_2)\exp(x_1)$ evaluated at $(x_1, x_2) = (-1, 2)$ and setting $x_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$

Algorithm 2 Forward Primal Trace

```
  $v_{-1} = x_1 = -1$
  $v_0 = x_2 = 2$
  ---
  $v_1 = v_{-1} \times v_0 = -1 \times 2$
  $v_2 = \sin(v_0) = \sin(2)$
  $v_3 = \exp(v_{-1}) = \exp(-1)$
  $v_4 = v_2 \times v_3 = 0.0349 \times 0.3679$
  $v_5 = v_1 + v_4 = -2 + 0.3679$
  ---
  $y = v_5 = -1.6321$
```

The Forward Primal Trace evaluates the function by breaking it down into a sequence of elementary operations and corresponding intermediate variables v_i. These intermediate variables are computed step by step, and their values are updated accordingly.

The derivative is computed by applying the chain rule to the intermediate variables, in this case, $\dot{v}_i$, following the same sequence of operations as in the Forward Primal Trace. The final value of $\dot{y}$ represents the derivative $\frac{\partial y}{\partial x_1}$ evaluated at the given point.

These algorithms will compute one column of our Jacobian matrix for our function $f : \mathbb{R}^n \to \mathbb{R}^m$ at point $x = a$

$$J_f = \left.\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}\right|_{x=a}$$

Algorithm 3 Forward Derivative Trace

```
$\dot{v_{-1}} = \dot{x_1} = 1$
$\dot{v_0} = \dot{x_2} = 0$

$\dot{v_1} = \dot{v_{-1}} \times v_0 + \dot{v_0} \times v_{-1} = 1 \times 2 + 0 \times -1$
$\dot{v_2} = \dot{v_0} \times \cos(v_0) = 0 \times \cos(2)$
$\dot{v_3} = \dot{v_{-1}} \times \exp(v_{-1}) = 1 \times \exp(-1)$
$\dot{v_4} = \dot{v_2} \times v_3 + \dot{v_3} \times v_2 = 0\cos(2) \times \exp(-1) + 1\exp(-1) \times \sin(2)$
$\dot{v_5} = \dot{v_1} + \dot{v_4} = 2 + 0.0128$

$\dot{y} = \dot{v_5} = 2.0128$
```

Thus, the full Jacobian can be computed in n evaluations.

Forward mode AD is very efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where n is small since we compute the Jacobian in n passes. For cases $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n \gg m$, reverse mode AD is often preferred.

7.3.2 Reverse AD

Reverse AD differs in that it propagates derivatives backward from a given output. This is done by complementing each intermediate variable v_i with an adjoint

$$\bar{v_i} = \frac{\partial y_j}{\partial v_i}$$

which represents the sensitivity of a considered output y_j with respect to changes in v_i.

Returning to our previous example for the function
$y = f(x_1, x_2) = x_1 x_2 - sin(x_2)\exp(x_1)$ evaluated at $(x_1, x_2) = (-1, 2)$. Starting from $\bar{v_5} = \bar{y} = 1$

Our forward primal trace remains the same as above.

In reverse AD, the reverse trace propagates the adjoints backwards, following the reverse order of the intermediate variables. At each step, the adjoint of the current variable is computed by multiplying the adjoint of the next variable with the partial derivative of the next variable with respect to the current variable.

Algorithm 4 Forward Primal Trace

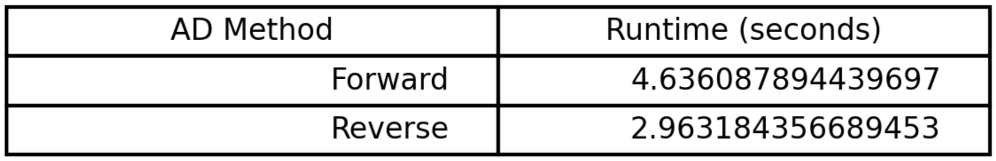
```
    $v_{-1} = x_1 = -1$
    $v_0 = x_2 = 2$
    ---
    $v_1 = v_{-1} \times v_0 = -1 \times 2$
    $v_2 = \sin(v_0) = \sin(2)$
    $v_3 = \exp(v_{-1}) = \exp(-1)$
    $v_4 = v_2 \times v_3 = 0.0349 \times 0.3679$
    $v_5 = v_1 + v_4 = -2 + 0.3679$
    ---
    $y = v_5 = -1.6321$
```

Algorithm 5 Reverse Derivative Trace

```
    $\bar{v_5} = \bar{y} = 1$
    ---
    $\bar{v_4} = \bar{v_5} \times \frac{\partial v_5}{\partial v_4} = 1$
    $\bar{v_1} = \bar{v_5} \times \frac{\partial v_5}{\partial v_1} = 1$
    $\bar{v_3} = \bar{v_4} \times \frac{\partial v_4}{\partial v_3} = 1 \times v_2 = \sin(2)$
    $\bar{v_2} = \bar{v_4} \times \frac{\partial v_4}{\partial v_2} = 1 \times v_3 = \exp(-1)$
    $\bar{v_0} = \bar{v_1} \times \frac{\partial v_1}{\partial v_0} + \bar{v_3} \times \frac{\partial v_3}{\partial v_{-1}} = 1 \times v_{-1} - \sin(2) \times \exp(-1) = -1 - 0.0349$
    $\bar{v_{-1}} = \bar{v_1} \times \frac{\partial v_1}{\partial v_{-1}} + \bar{v_2} \times \frac{\partial v_2}{\partial v_0} = 1 \times v_0 + \exp(-1) \times \cos(2) = 2 + 0.0128$
    ---
    $\bar{x_1} = \bar{v_{-1}} = 2 + 0.0128$
    $\bar{x_2} = \bar{v_0} = -1 - 0.0349$
```

The reverse derivative trace may seem a little more un-intuitive from the forwards AD derivative, and this can be down to the way we are taught to calculate derivatives with the chain rule.

Reverse mode AD can be significantly quicker to evaluate for functions with a large number of inputs, that is for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ where n is large. Reverse mode AD performs better then $m \ll n$. Reverse AD naturally lends itself to neural networks since we typically have many more parameters than inputs. Forwards AD would be required to complete a pass for each input variable. This specific case of reverse AD is called back propagation.



AD Method	Runtime (seconds)
Forward	4.636087894439697
Reverse	2.963184356689453

Figure 11: Comparison in speed between Forward and Reverse AD on a neural network with 4 fully connected layers to predict a target variable based on a 50-dimensional input.

7.4 ADVI Implementation

Different implementations of ADVI have different ways of choosing a variational distribution to use in our algorithm. Pymc documentation of the 'advi' class states "This class implements the meanfield ADVI, where the variational posterior distribution is assumed to be spherical Gaussian without correlation of parameters and fit to the true posterior distribution." [77]. However, pymc also includes other VI implementations such as Amortized

Table 1: Comparison of gradient values
Function: $f(x) = \log(\sin(x^2) + 1) + \sqrt{e^x}$
Evaluation Point: $x = 1.0$

Method	h	Gradient
True Gradient	–	1.4111766815185547
Numerical Gradient (Forward)	0.01	1.4052391052246094
Numerical Gradient (Central)	0.01	1.4110922813415527
Numerical Gradient (Forward)	0.001	1.4107226133346558
Numerical Gradient (Central)	0.001	1.4111994504928589
Numerical Gradient (Forward)	0.0001	1.41143798828125
Numerical Gradient (Central)	0.0001	1.41143798828125
Numerical Gradient (Forward)	1e-05	1.430511474609375
Numerical Gradient (Central)	1e-05	1.4185905456542969
Automatic Differentiation	–	1.4111766815185547

Stein Variational Gradient Descent (ASVGD) [78], Stein Variational Gradient Descent (SVGD), and Full Rank Automatic Differentiation Variational Inference (ADVI).

Stan implementation allows choice between a fully factorised (mean field) Guassian approximation or a Gaussian with a full-rank covariance matrix for the approximation [32]. The Edward python library for probabalistic programming built upon deep learning library 'tensorflow' [79] also supports Black box variational inference [80] and Stochastic variational inference [81].

7.5 ADVI Polynomial Regression Example

We now implement ADVI using the PyMC3 library in python. PyMC is a Python package for Bayesian statistical modeling and probabilistic machine learning which focuses on advanced Markov chain Monte Carlo and variational fitting algorithms [82]. PyMC allows us to utilise probabilistic programming to define and fit our model.

In this problem, we are working with a synthetic dataset generated using a third-degree polynomial function with some added noise. Our goal is to fit a

Bayesian polynomial regression model to the data and make predictions on new data points.

```
    import numpy as np
    import matplotlib.pyplot as plt
    import pymc3 as pm

    # Generate synthetic data for polynomial regression
    np.random.seed(42)
    n = 100
    alpha_true = 2
    beta_true = np.array([3, -2, 1])
        # polynomial coefficients for x^1, x^2, and x^3
    sigma_true = 1

    X = np.random.randn(n)
    Y = alpha_true + np.dot(beta_true, np.array([X, X**2, X**3]))
        + np.random.normal(0, sigma_true, size=n)

    # Visualize the synthetic data
    plt.scatter(X, Y)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.show()
```

We begin by generating our synthetic data which comes from a polynomial function with added noise. We can then visualise our data.

Now we use the built in ADVI functionality from the PYMC3 library to create our model.

```
# Build a PyMC3 model
with pm.Model() as model:
    # Priors for unknown model parameters
    alpha = pm.Normal('alpha', mu=0, sd=10)
    beta = pm.Normal('beta', mu=0, sd=10, shape=3)
    sigma = pm.HalfNormal('sigma', sd=10)

    # Expected value of outcome
    mu = alpha + pm.math.dot(beta, np.array([X, X**2, X**3]))
```

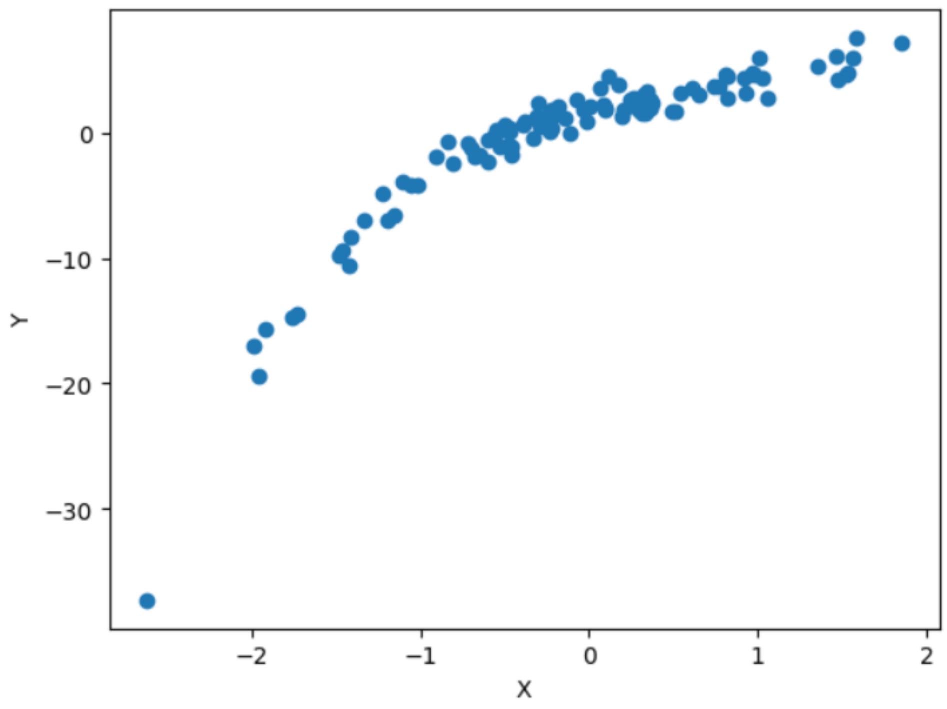


Figure 12: Data generated from our polynomial function

```
    # Likelihood (sampling distribution) of observations
    Y_obs = pm.Normal('Y_obs', mu=mu, sd=sigma, observed=Y)

    # Fit the model using ADVI
    approx = pm.fit(n=100000, method='advi')
```

For our model, we define priors for each of our parameters in our polynomial regression model. We define an expected value and the likelihood of observations, which we specify to follow a normal distribution.

We can then fit the model for 100000 iterations using the 'advi' method.

Running the model we can see that the trace plot in figure 13 shows that the ELBO converges.

Studying our posterior distributions in figure 14 for the model parameters,

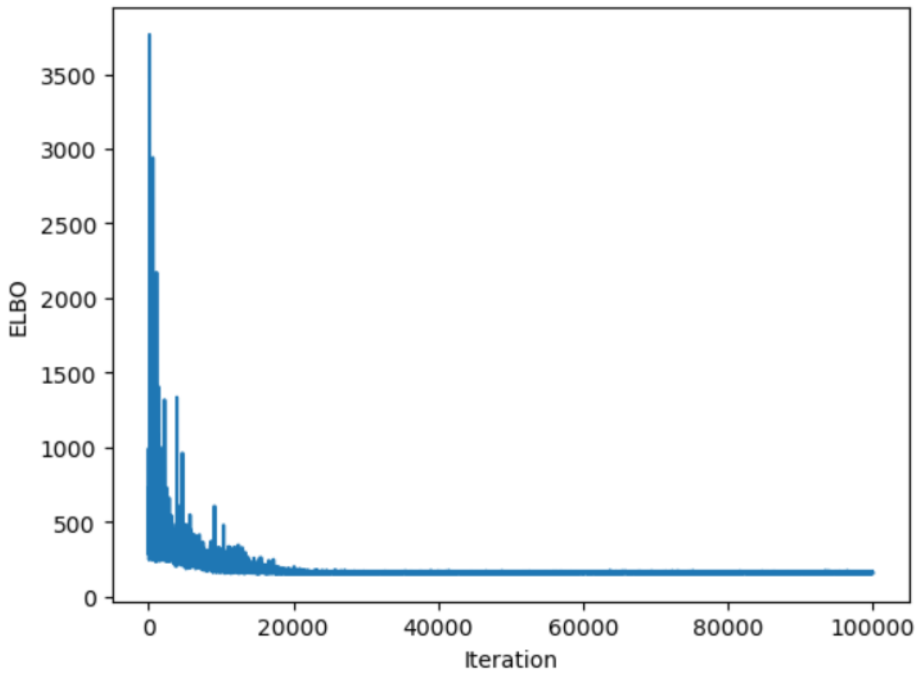


Figure 13: Trace plot showing the ELBO at 100000 iterations of 'advi' method.

we see that they have roughly converged to the true distribution of the parameters.

Finally, we use our trained model to predict new data values and plot the predictive distribution.

```
# Predict new data
X_new = np.linspace(-3, 3, 50)
Y_pred = np.zeros((len(X_new), len(trace)))

for i, val in enumerate(trace):
    Y_pred[:, i] = val['alpha'] + np.dot(val['beta'],
        np.array([X_new, X_new**2, X_new**3]))

# Plot the predictive distribution
```

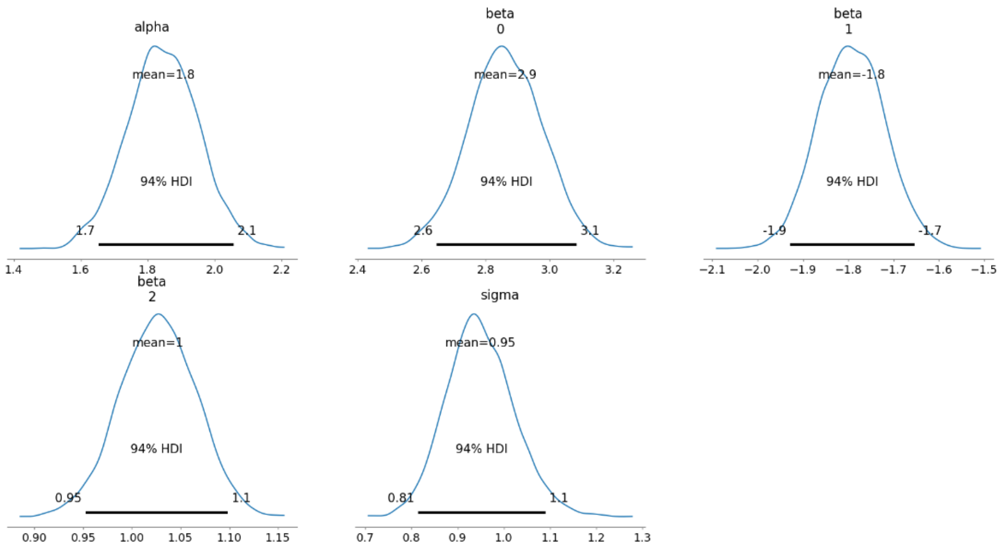


Figure 14: Posterior distributions of model parameters

```
plt.scatter(X, Y, label='Observed data')
plt.plot(X_new, Y_pred.mean(axis=1), label='Predictive mean')
plt.fill_between(X_new, np.percentile(Y_pred, 2.5, axis=1),
    np.percentile(Y_pred, 97.5, axis=1), alpha=0.5, label='95% CI')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

In conclusion, this method was very easy to implement due to it being a more black box approach. The majority of the difficulties of implementing a VI method were removed as we simply defined a model, defined priors and likelihoods, and then ran the code. This was a huge quality of life improvement over modular implementation of a VI algorithm which can often require a high level of expertise.

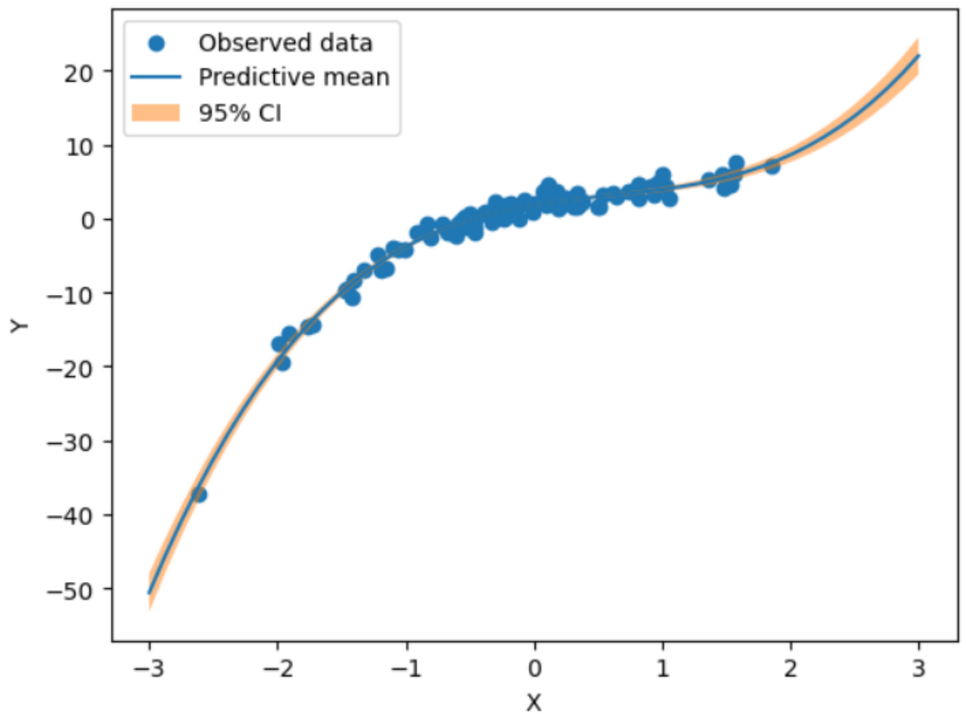


Figure 15: Plot the predictive distribution.

7.6 ADVI Bayesian Neural Network Example

7.6.1 Bayesian Neural Network (BNN)

A Bayesian neural network combines probabilistic models with the architecture of a neural network. We place distributions on the weights of our neural network. These probability distributions of network weights are used to estimate the uncertainty in weights and predictions. The posterior weights are then calculated using Bayes rule. See figure 16.

As a result, when training our network we are updating our distributions of the weights and biases, and these can be intractable, so we use can use methods such as MCMC or VI to train our model.

Bayesian neural networks combine strengths of Bayesian inference with the flexibility and representational power of neural networks, leading to improved model performance and robustness in various tasks. It is useful to use a bayesian framework in a neural network since we have the power of incorporating priors into our model, as well as having a sense of measurable uncertainty in our predictions.

The power of bayesian inference in the model can be useful for many reasons:

- Understanding uncertainty: Bayesian neural networks maintain a probability distribution over model weights which provides more informative predictions
- Regularisation: Since we encode priors into our model, the weights are somewhat constrained and this can help avoid overfitting
- Adaptability: Bayesian models can easily incorporate new data due to Bayes rule. As new data is streamed in, we can keep training our model.

7.6.2 Example

In our example, we construct a bayesian neural network and train the weight using ADVI implemented from PyMC3, as in the previous example. Implementation from PyMC3 allows us to very easily train the weights using VI.

The California Housing dataset is a widely-used dataset that provides information on various aspects of housing in the state of California [48]. The dataset consists of over 20,000 data points with various features such as median house value, median income, total rooms, total bedrooms, population, households, and housing units. Additionally, it includes geographical information like longitude and latitude. The primary purpose of the dataset is to study the relationships between these features and predict housing prices or identify patterns in housing affordability across the state, making it an ideal candidate for machine learning applications. It is similar to the famous Boston housing dataset [42], but the Boston housing dataset contains many inherent racial biases which are inappropriate for machine learning models. The target variable is the median house value for California districts.

Our code is as follows.

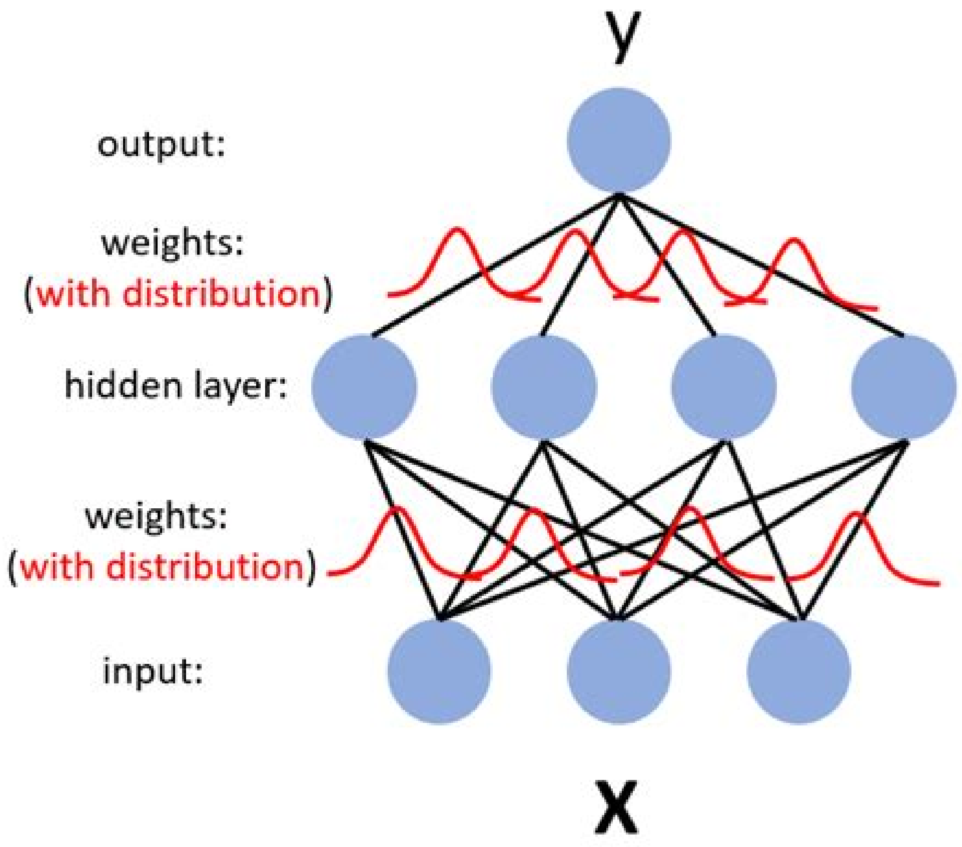


Figure 16: A simple bayesian neural network with one hidden layer.

```
import numpy as np
import theano
import theano.tensor as tt
import pymc3 as pm
import arviz as az
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score,
    confusion_matrix
```

```
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns

# Load California Housing dataset
data = fetch_california_housing()
X = data['data']
y = data['target']

# Standardize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
```

This part of the code is preparing the dataset and setting up the input and target variables for the Bayesian Neural Network. We simply rescale the features and create a test train split

```
input_dim = X.shape[1]
hidden_dim = 15
output_dim = 1

# BNN model
with pm.Model() as bnn_model:
    # Input layer
    weights_1 = pm.Normal("weights_1", mu=0, sigma=1, shape=
        (input_dim, hidden_dim))
    bias_1 = pm.Normal("bias_1", mu=0, sigma=1, shape=
        (hidden_dim,))
    layer_1 = tt.nnet.relu(tt.dot(X_train, weights_1) + bias_1)

    # Hidden layer
    weights_15 = pm.Normal("weights_15", mu=0, sigma=1, shape=
        (hidden_dim, hidden_dim))
    bias_15 = pm.Normal("bias_15", mu=0, sigma=1, shape=
        (hidden_dim,))
    layer_15 = tt.nnet.relu(tt.dot(layer_1, weights_15) +
        bias_15)

    # Output layer
    weights_2 = pm.Normal("weights_2", mu=0, sigma=1, shape=
        (hidden_dim, output_dim))
    bias_2 = pm.Normal("bias_2", mu=0, sigma=1, shape=
        (output_dim,))
    mu = tt.dot(layer_15, weights_2) + bias_2
    sd = pm.HalfNormal("sd", sigma=1)
    obs = pm.Normal("obs", mu=mu, sigma=sd,
        observed=y_train.reshape(-1, 1))
```

Our next section of code defines the BNN model. The BNN has one input layer, one hidden layer, and one output layer. We define priors for all weights and biases as Gaussian distributions with mean 0 and variance 1, which helps regularise the network helping to prevent overfitting. In a model such as a BNN it is difficult to impart any sort of informative prior, since the network is very complex and uninterpretable, however, attempts have been made [83]. We use ReLU activation functions for the input and hidden layers, before outputting a single value since this is a regression problem.

```
    # ADVI training
    with bnn_model:
        advi = pm.ADVI()
        approx = pm.fit(n=50000, method=advi, callbacks=
            [pm.callbacks.CheckParametersConvergence(diff='absolute')],
                obj_optimizer=pm.adam(learning_rate=0.01), progressbar=True)

    def predict(X, samples):
        n_samples = len(samples['weights_1'])
        predictions = np.zeros((n_samples, X.shape[0]))

        for i, sample in enumerate(samples):
            layer_1 = np.maximum(0, np.dot(X, sample['weights_1']) +
                sample['bias_1'][None, :])
            layer_15 = np.maximum(0, np.dot(layer_1,
                sample['weights_15']) + sample['bias_15'][None, :])
            y_pred = np.dot(layer_15, sample['weights_2']) +
                sample['bias_2'][None, :]
            predictions[i] = y_pred.reshape(-1)

        return np.mean(predictions, axis=0)
```

In this section, we train our model using PyMC3's ADVI method. The predict function is then designed to make predictions using the trained model.

```
    # Generate posterior samples
    posterior_samples = approx.sample(draws=5000)

    # Make predictions on the train and test sets
    y_pred_train = predict(X_train, posterior_samples)
    y_pred_test = predict(X_test, posterior_samples)

    # Calculate mean squared error
    train_mse = mean_squared_error(y_train, y_pred_train)
    test_mse = mean_squared_error(y_test, y_pred_test)
    print(f"Train MSE: {train_mse:.2f}")
    print(f"Test MSE: {test_mse:.2f}")
```

This part of the code evaluates the performance of the trained Bayesian Neural Network (BNN) model on the test dataset by sampling from the posterior predictive distribution and calculating the accuracy.

By drawing samples of the model parameters, we can create predictions which account for the uncertainty surrounding the true value of each weight and bias in the BNN. We show the predictions in figure 17.

Since we use a bayesian neural network, we have distributions over our weights meaning we are able to see the uncertainty of our predictions. We show the uncertainty in the predictions of the first 50 data points in figure 18.

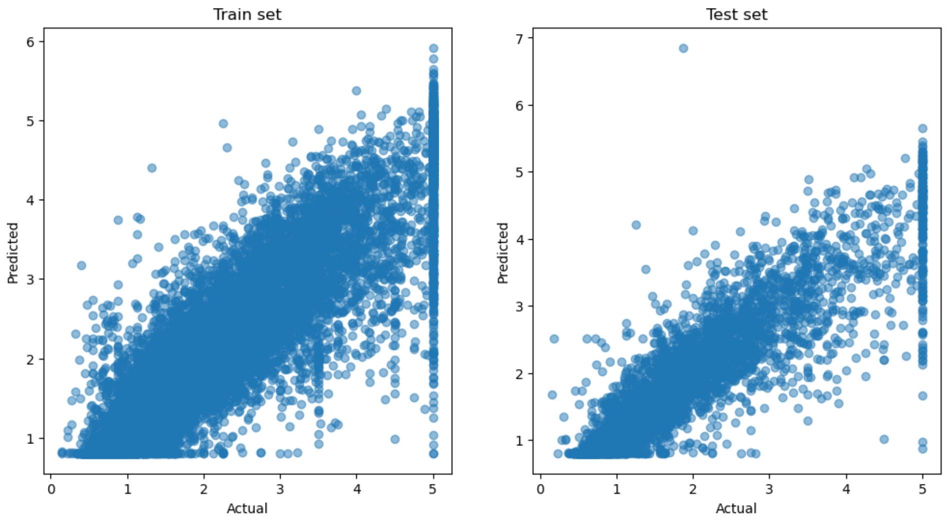


Figure 17: Actual values against predictions plot for train set and test set.

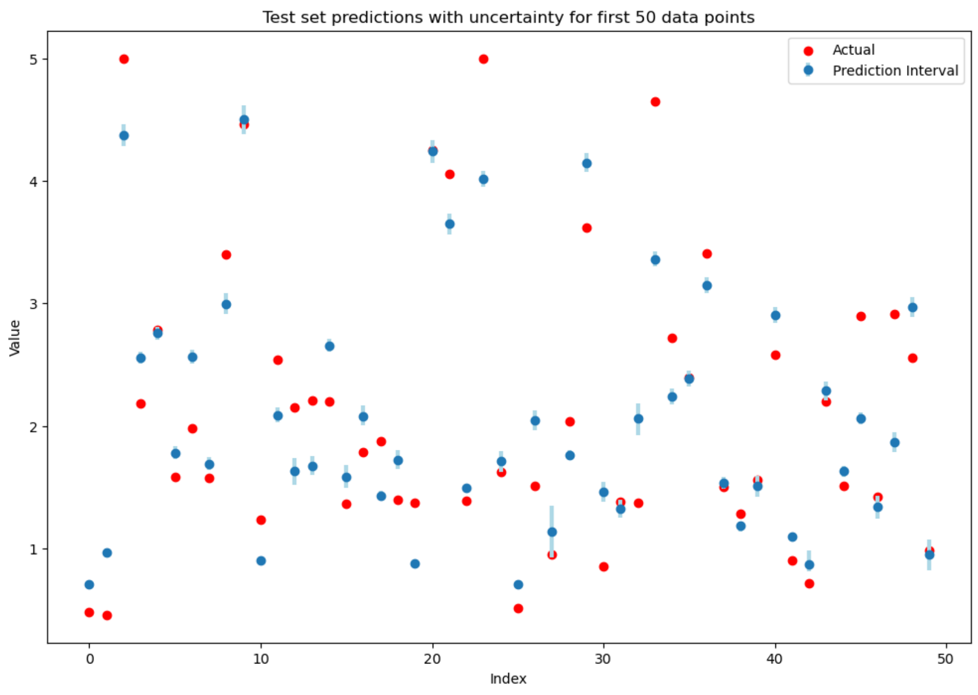


Figure 18: Uncertainty from the model parameters in our first 50 data point predictions.

8 Conclusion and Future Research

There is a need for scalable and efficient algorithms for bayesian inference and probabalistic machine learning. VI provides a powerful, flexible and efficient method for approximating posterior distributions in bayesian frameworks. We have shown the building blocks of VI. and brought to light the potential for areas of research to provide improvements in accuracy, computational complexity, and specific domains. Despite efforts, usage can be limited for non-experts due to the difficult tricks often required for accelerating convergence and deep understanding required for selecting appropriate models. We hope this paper brings insight into the topic, especially where MCMC methods are much more widely taught.

8.1 Identified Future Research

Through modularly looking at the sections which make up the overall VI algorithm, we identify specific areas of interest for future work to be done

8.1.1 Theoretical Aspects of VI

Thoughtout this paper we struggle to address the theoretical aspects of VI. This begs fundamental questions about how we can actually guarantee accuracy in our algorithms. This field could vastly benefit from being able to quantify the error in our variational approximations, perhaps from a deeper connection with information theory [15].

8.1.2 Alternative Divergences

We identify the potential to use alternative divergences rather than the standard KL divergence. In summary, alternative divergences offer a flexible and application specific way to improve accuracy at times for our overall approximation. It seems possible to conclude that there may be no correct divergence which will always result in best approximations, and that alternative divergences can offer benefits. The KL divergence works so well because of the ease of implementation and strong research backing as the standard method for VI since its inception.

8.1.3 SGD variance reduction

Many modern methods look into the area of variance reduction, with aims to accelerate and reduce variance in the SGD gradient estimation. We were unable to go into a large amount of detail in this paper specifically, however, various papers focused on the topic of variance reduction show promising results in terms of computational speed and complexity for implementing these algorithms to improve convergence of SGD. We refer the reader to useful papers which help to compare and combine methods with the potential of offering speed ups to VI [51] [15] [84].

8.1.4 MCMC VI

We conclude that VI has an important role in being potentially more scalable and fast than typical MCMC methods, however, often suffers drawbacks such as a less accuracy in the output approximations, since we are fitting a variational distribution. There is work being done on looking into how we might be able to combine these approaches to leverage the benefits of both methods, offering a boost in accuracy to VI [85] [15] [86]. Note that these methods are not widely adopted, but possibly show potential.

References

[1] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American statistical Association*, vol. 112, no. 518, pp. 859–877, 2017. [Online]. Available: https://arxiv.org/abs/1601.00670

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

[3] J. Rocca, "Bayesian inference problem, mcmc and variational inference," 2019. [Online]. Available: https://towardsdatascience.com/bayesian-inference-problem-mcmc-and-variational-inference-25a8aa9bce29

[4] T. Salimans, D. P. Kingma, and M. Welling, "Markov chain monte carlo and variational inference: Bridging the gap," 2015.

[5] D. van Ravenzwaaij, P. Cassey, and S. Brown, "A simple introduction to markov chain monte–carlo sampling," *Psychonomic Bulletin& Review*, vol. 25, 03 2016.

[6] K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023. [Online]. Available: http://probml.github.io/book2

[7] Y. Zhang, W. Liu, Z. Chen, K. Li, and J. Wang, "On the properties of kullback-leibler divergence between multivariate gaussian distributions," 05 2022.

[8] K. P. Murphy, *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. [Online]. Available: probml.ai

[9] E. W. Weisstein, ""jensen's inequality." from mathworld–a wolfram web resource." [Online]. Available: https://mathworld.wolfram.com/JensensInequality.html

[10] M. Xu, M. Quiroz, R. Kohn, and S. A. Sisson, "Variance reduction properties of the reparameterization trick," 2018. [Online]. Available: https://arxiv.org/abs/1809.10330

[11] Y. XIe, "Lecture 3: Chain rules and inequalities," 2012, eCE587: Information Theory. [Online]. Available: https://www2.isye.gatech.edu/~yxie77/ece587/Lecture3.pdf

[12] S. Dutta and S. Furuichi, "On log-sum inequalities," 2022.

[13] J. Soch, "Proof: Convexity of the kullback-leibler divergence," 2020. [Online]. Available: https://statproofbook.github.io/P/kl-conv.html

[14] T. A. Le, "Reverse vs forward kl," 2017. [Online]. Available: https://www.tuananhle.co.uk/notes/reverse-forward-kl.html

[15] C. Zhang, J. Bütepage, H. Kjellström, and S. Mandt, "Advances in variational inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 8, pp. 2008–2026, 2019.

[16] W. Kurt, "Kl divergence explained," 2017. [Online]. Available: https://www.countbayesie.com/blog/2017/5/9/kullback-leibler-divergence-explained

[17] C. Versloot, "how to use kullback-leibler divergence kl divergence with keras," 2022. [Online]. Available: https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-kullback-leibler-divergence-kl-divergence-with-keras.md

[18] O. Kosheleva and V. Kreinovich, "Why deep learning methods use kl divergence instead of least squares: A possible pedagogical explanation," 2017. [Online]. Available: https://scholarworks.utep.edu/cs_techrep/1192/

[19] *Entropy, Relative Entropy, and Mutual Information.* John Wiley& Sons, Ltd, 2005, ch. 2, pp. 13–55. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/047174882X.ch2

[20] P. Gustafson, "On measuring sensitivity to parametric model misspecification," *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, vol. 63, no. 1, pp. 81–94, 2001. [Online]. Available: http://www.jstor.org/stable/2680635

[21] Wikipedia contributors, "Divergence (statistics) — Wikipedia, the free encyclopedia," 2023, [Online; accessed 7-April-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Divergence_(statistics)&oldid=1136432314

[22] J.-B. Regli and R. Silva, "Alpha-beta divergence for variational inference," 2018. [Online]. Available: https://arxiv.org/abs/1805.01045

[23] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," 2017.

[24] A. L. Gibbs and F. E. Su, "On choosing and bounding probability metrics," 2002.

[25] Y. Li and R. E. Turner, "Rényi divergence variational inference," 2016.

[26] A. Basu, I. R. Harris, N. L. Hjort, and M. C. Jones, "Robust and efficient estimation by minimising a density power divergence," *Biometrika*, vol. 85, pp. 549–559, 1998.

[27] Machinelearning.wtf, "Mode collapse," 2021. [Online]. Available: https://machinelearning.wtf/terms/mode-collapse/

[28] Y. Li and Y. Gal, "Dropout inference in Bayesian neural networks with alpha-divergences," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 2052–2061. [Online]. Available: https://proceedings.mlr.press/v70/li17a.html

[29] M. N. Bernstein, "The evidence lower bound (elbo)," 2020. [Online]. Available: https://mbernste.github.io/posts/elbo/

[30] D. M. Blei, "Variational inference lecture cos597c," 2011. [Online]. Available: https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf

[31] B. Magalhaes, "Variational inference: Elbo, mean-field approximation, cavi and gaussian mixture models," 2019. [Online]. Available: https://brunomaga.github.io/Variational-Inference-GMM

[32] A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic differentiation variational inference," 2016.

[33] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, "Automatic differentiation in machine learning: a survey," *CoRR*, vol. abs/1502.05767, 2015. [Online]. Available: http://arxiv.org/abs/1502.05767

[34] M. Jordan, Z. Ghahramani, T. Jaakkola, and L. Saul, "An introduction to variational methods for graphical models," *Machine Learning*, vol. 37, pp. 183–233, 11 1999.

[35] R. Yao and Y. Yang, "Mean field variational inference via wasserstein gradient flow," *arXiv preprint arXiv:2207.08074*, 2022. [Online]. Available: https://arxiv.org/abs/2207.08074e;

[36] S. Farquhar, L. Smith, and Y. Gal, "Try depth instead of weight correlations: Mean-field is a less restrictive assumption for deeper networks," *CoRR*, vol. abs/2002.03704, 2020. [Online]. Available: https://arxiv.org/abs/2002.03704

[37] S. Plummer, D. Pati, and A. Bhattacharya, "Dynamics of coordinate ascent variational inference: A case study in 2d ising models," *Entropy*, vol. 22, no. 11, p. 1263, nov 2020. [Online]. Available: https://arxiv.org/abs/2007.06715

[38] S. J. Wright, "Coordinate descent algorithms," 2015. [Online]. Available: https://arxiv.org/abs/1502.04759

[39] D. Durante and T. Rigon, "Conditionally conjugate mean-field variational bayes for logistic models," *Statistical Science*, vol. 34, no. 3, aug 2019. [Online]. Available: https://doi.org/10.1214%2F19-sts712

[40] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[41] Wikipedia contributors, "Stochastic approximation — Wikipedia, the free encyclopedia," 2023, [Online; accessed 8-April-2023]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Stochastic_approximation&oldid=1144917516

[42] "Boston housing dataset," 1978. [Online]. Available: https://scikit-learn.org/0.16/modules/generated/sklearn.datasets.load_boston.html

[43] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive gradient methods with dynamic bound of learning rate," *CoRR*, vol. abs/1902.09843, 2019. [Online]. Available: http://arxiv.org/abs/1902.09843

[44] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[45] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980

[46] J. Brownlee, "Gentle introduction to the adam optimization algorithm for deep learning," 2017. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[47] DeepAI, "What is the adam optimization algorithm?" [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/adam-machine-learning

[48] "California housing dataset description," 1990. [Online]. Available: https://developers.google.com/machine-learning/crash-course/california-housing-data-description

[49] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference," 2013. [Online]. Available: https://arxiv.org/abs/1206.7051

[50] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Y. Lechevallier and G. Saporta, Eds. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186.

[51] R. M. Gower, M. Schmidt, F. Bach, and P. Richtárik, "Variance-reduced methods for machine learning," *Proceedings of the IEEE*, vol. 108, no. 11, pp. 1968–1983, 2020.

[52] T. Yu, X.-W. Liu, Y.-H. Dai, and J. Sun, "Stochastic variance reduced gradient methods using a trust-region-like scheme," *J. Sci. Comput.*, vol. 87, no. 1, apr 2021. [Online]. Available: https://doi.org/10.1007/s10915-020-01402-x

[53] N. L. Roux, M. Schmidt, and F. Bach, "A stochastic gradient method with an exponential convergence rate for finite training sets," 2013.

[54] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," 2018.

[55] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 315–323.

[56] G. C. K. P. D. P. Rishi Kaashyap Balaji, Shreshtha Dhankar, “Stochastic variance reduced gradient,” 2022. [Online]. Available: https://optimization.cbe.cornell.edu/index.php?title=Stochastic_variance_reduced_gradient

[57] D. Csiba and P. Richtárik, “Importance sampling for minibatches,” 2016. [Online]. Available: https://arxiv.org/abs/1602.02283

[58] N. Foti, “The natural gradient,” 2013. [Online]. Available: https://lips.cs.princeton.edu/the-natural-gradient/

[59] S.-i. Amari, “Natural gradient works efficiently in learning,” *Neural Computation*, vol. 10, no. 2, pp. 251–276, 1998. [Online]. Available: https://bsi-ni.brain.riken.jp/database/file/176/181.pdf

[60] Y. LeCun, “The mnist database of handwritten digits.” [Online]. Available: http://yann.lecun.com/exdb/mnist/index.html

[61] T. Shin, “A beginner-friendly explanation of how neural networks work,” 2020. [Online]. Available: https://towardsdatascience.com/a-beginner-friendly-explanation-of-how-neural-networks-work-55064db60df4

[62] J. Brownlee, “A gentle introduction to cross-entropy for machine learning,” 2019. [Online]. Available: https://machinelearningmastery.com/cross-entropy-for-machine-learning/

[63] D. P. Kingma and M. Welling, “Auto-encoding variational bayes,” *CoRR*, vol. abs/1312.6114, 2013. [Online]. Available: https://arxiv.org/abs/1312.6114

[64] stackexchange, “How does backprop work through the random sampling layer in a variational autoencoder?” 2022. [Online]. Available: https://ai.stackexchange.com/a/33829

[65] A. Soleimany, “Mit 6.s191 (2020): Introduction to deep learning, deep generative modeling,” 2020. [Online]. Available: http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L4.pdf

[66] D. P. Kingma and M. Welling, “An introduction to variational autoencoders,” *CoRR*, vol. abs/1906.02691, 2019. [Online]. Available: http://arxiv.org/abs/1906.02691

[67] C. Doersch, “Tutorial on variational autoencoders,” 2021.

[68] J. E. Tolsma and P. I. Barton, "On computational differentiation," *Computers& Chemical Engineering*, vol. 22, no. 4, pp. 475–490, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0098135497002640

[69] utexas, "Cs 381k: Symbolic differentiation," 2007. [Online]. Available: https://www.cs.utexas.edu/users/novak/asg-symdif.html

[70] mathworks, "Symbolic operations in matlab." [Online]. Available: https://uk.mathworks.com/help/symbolic/performing-symbolic-computations.html

[71] S. D. Team, "Sympi: a python library for symbolic mathematics." 2021. [Online]. Available: https://www.sympy.org/en/index.html

[72] N. Ketkar and J. Moolayil, *Automatic Differentiation in Deep Learning*, 04 2021, pp. 133–145.

[73] Q. Kong, T. Siauw, and A. M. Bayen, "Chapter 23. ordinary differential equation - boundary value problems," in *Python Programming and Numerical Methods*. Academic Press, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128195499000105

[74] E. Kirr, "Errors in numerical methods," 2007. [Online]. Available: https://faculty.math.illinois.edu/~ekirr/page/teaching/math385/handout2.pdf

[75] A. Griewank, "A mathematical view of automatic differentiation," *Acta Numerica*, vol. 12, p. 321–398, 2003.

[76] A. Griewank and A. Walther, *Evaluating Derivatives*, 2nd ed. Society for Industrial and Applied Mathematics, 2008. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898717761

[77] pymc dev, "pymc.advi," 2017. [Online]. Available: https://www.pymc.io/projects/docs/en/latest/api/generated/pymc.ADVI.html

[78] Q. Liu and D. Wang, "Stein variational gradient descent: A general purpose bayesian inference algorithm," 2019.

[79] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser,

M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[80] R. Ranganath, S. Gerrish, and D. M. Blei, "Black box variational inference," 2013.

[81] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei, "Edward: A library for probabilistic modeling, inference, and criticism," *arXiv preprint arXiv:1610.09787*, 2016.

[82] J. Salvatier, T. Wiecki, and C. Fonnesbeck, "Probabilistic programming in python using pymc," 2015.

[83] V. Fortuin, A. Garriga-Alonso, S. W. Ober, F. Wenzel, G. Rätsch, R. E. Turner, M. van der Wilk, and L. Aitchison, "Bayesian neural network priors revisited," 2022.

[84] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. Curran Associates Inc., 2013, p. 315–323.

[85] F. J. R. Ruiz and M. K. Titsias, "A contrastive divergence for combining variational inference and mcmc," 2019.

[86] A. Thin, N. Kotelevskii, J.-S. Denain, L. Grinsztajn, A. Durmus, M. Panov, and E. Moulines, "Metflow: A new efficient method for bridging the gap between markov chain monte carlo and variational inference," 2020.

[87] T. Blain, "Adaptive learning rate methods for stochastic gradient descent and vanishing gradient problems from backpropagation," 2023, Unpublished, Data Science Toolbox.

[88] deepai, "What is a hidden layer?" [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning

[89] PyTorch, “Pytorch hub for researchers,” 2023. [Online]. Available: https://pytorch.org/hub/research-models

[90] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015.

[91] PyTorch, “resnet152,” 2015. [Online]. Available: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet152.html

[92] IBM, “What are convolutional neural networks?” [Online]. Available: https://www.ibm.com/topics/convolutional-neural-networks

[93] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation Applied to Handwritten Zip Code Recognition,” *Neural Computation*, vol. 1, no. 4, pp. 541–551, 12 1989. [Online]. Available: https://doi.org/10.1162/neco.1989.1.4.541

[94] Hecht-Nielsen, “Theory of the backpropagation neural network,” in *International 1989 Joint Conference on Neural Networks*, 1989, pp. 593–605 vol.1.

A Introduction to Neural Networks

Neural networks are a class of machine learning models that are loosely inspired by the biological neural networks that make up the human brain. These models consist of interconnected layers of artificial neurons or nodes, where each connection has a weight associated with it. The neurons receive input signals, process them, and produce output signals that are passed on to the next layer. The process of learning in a neural network involves adjusting the weights of these connections to minimise the error between the network's predictions and the ground truth. This section is taken from [87] where here it serves as a useful appendix to introduce fundamental ideas of neural networks to the reader in order to better understand examples throughout the paper.

A.0.1 Perceptrons

Neural networks can be in their simplist form, perceptron algorithms. A perceptron consists of an input layer, a processing unit, and an output layer. The input layer receives a feature vector $\boldsymbol{x} \in \mathbb{R}^d$, where d is the number of features. Each feature in the input vector is multiplied by a corresponding weight w_i, and the sum of these weighted inputs is computed:

$$z = \sum_{i=1}^{d} w_i x_i + b, \tag{54}$$

where b is the bias term. The sum z is then passed through an activation function $f(\cdot)$ to produce the output $\hat{y}$:

$$\hat{y} = f(z). \tag{55}$$

In neural networks, the objective function $J(\theta)$ is typically a loss function that quantifies the difference between the predicted outputs and the actual target values. The goal is to minimise this loss function by adjusting the model's the weights w_i and biases b of the connections between the neurons.

A.0.2 Activation Functions

Activation functions are typically mathematical functions which transform the input received by a neuron into an output signal. This allows us to introduce non-linearity into our networks which enables the nerual network to learn more complex relationships between the inputs and outputs. There are many possible popular choices for activation function, and this is usually selected based on computational complexity and the specific problem.

- Sigmoid function: The sigmoid function maps input values to the range (0, 1). It is defined as:
$$f(x) = \frac{1}{1+e^{-x}}. \tag{56}$$

- Tanh function: The tanh function maps input values to the range (-1, 1). It is defined as:
$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{57}$$

- Rectified Linear Unit (ReLU) function: The ReLU function is a simple non-linear function that outputs the input if it is positive, and 0 otherwise. It is defined as:
$$f(x) = \max(0, x). \tag{58}$$

A.0.3 Neural Networks

A neural network is a more complex model composed of multiple layers. Each neuron in a layer receives input from the neurons in the previous layer, applies weights and biases, and passes the result through an activation function to produce an output. This output is then passed on to the neurons in the next layer. Neural networks can have one or more hidden layers between the input and output layers, each with a specified number of nodes, and each node with its own set of weights and biases. Different activation functions can be used on the different layers.

Hidden layers are a key reason why neural networks are able to capture very complex relationships and achieve high performance in many tasks. Deep

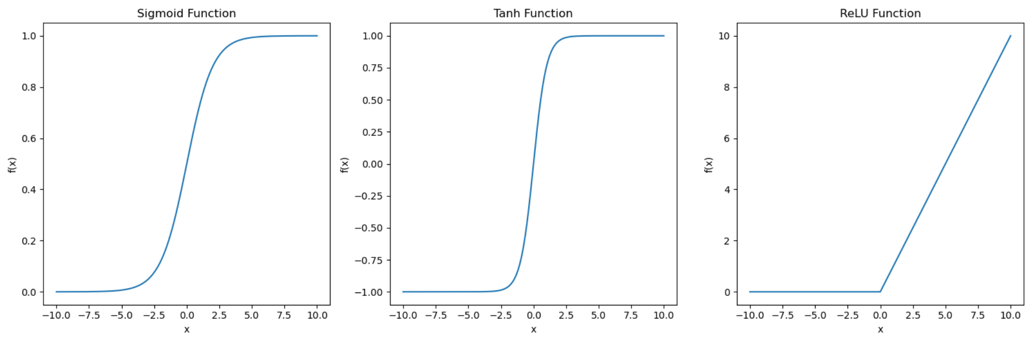


Figure 19: Figure showing common activation functions for Neural Networks, (left to right [Sigmoid, Tanh, ReLU])

learning architectures often use many hidden layers to solve complex problems, such as image recognition. Popular models designed by researchers for image recognition tasks can frequently have more than 30 layers [89], with ResNet-152 containing 152 layers [90]. These models can require huge amounts of computing power and resources to train; with ResNet-152 containing 60192808 parameters, all to be optimised [91]. Deep learning image models usually use a convolutional neural network (CNN) architecture, which additionally contain convolutional layers and pooling layers alongside fully connected layers. These are particually useful in computer vision applications since they allow the network to detect local patterns in images. For a more in depth guide, see [92]. For a famous application on classifying handwritten digits, see [93].

A.1 Gradient Based Optimisation

Alongside this increasing complexity of high performance deep neural networks, comes the need for efficient and scalable optimisation algorithms.

A.1.1 Backpropagation

Automatic Differentiation (AD) uses symbolic rules for differentiation which are more accurate than approximations within numerical differentiation. Unlike symbolic differentiation, automatic differentiation evaluates expressions numerically early in the computations rather than carrying out large symbolic

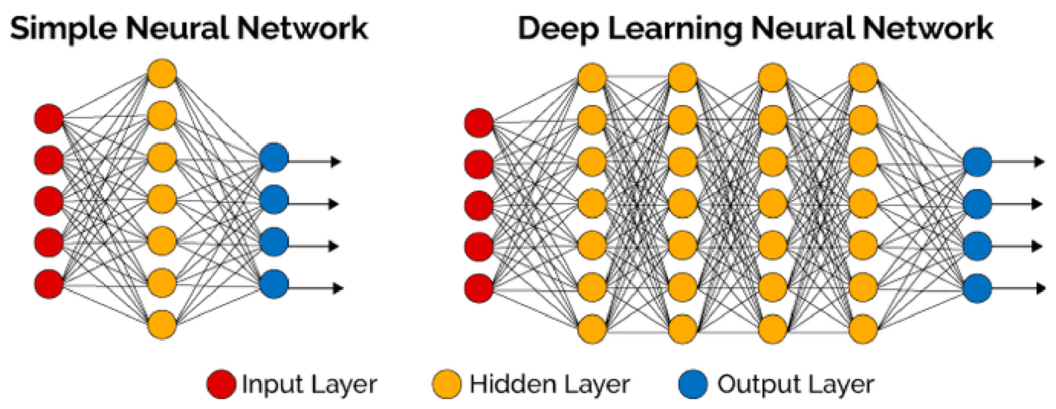


Figure 20: High level view of neural network architectures. Source: [88]

computations. In other words, automatic differentiation evaluates derivatives at particular numeric values and does not construct symbolic expressions for derivatives [75]. Backpropagation is a specific case of Reverse mode AD, this propagates derivatives backward from a given output.

Neural Networks naturally lend themselves to backpropagation because of their layered structure, differentiable activation functions, and chain rule of calculus [94]. Being careful to avoid the intricacies of automatic differntiation, backpropagation is a key part of the success of neural networks, enabling very efficient gradient calculations for networks with a large number parameters. This gives rise to the effective use of gradient based optimisation methods such as Gradient Descent. For more information on Automatic Differentiation and Backpropagation, see section 7.