

UNIVERSITY OF EDINBURGH

SCHOOL OF INFORMATICS

MSC BY RESEARCH IN DATA SCIENCE

---

# Distilling Model Knowledge

---

by

GEORGE PAPAMAKARIOS

August 2015

## Abstract

Top-performing machine learning systems, such as deep neural networks, large ensembles and complex probabilistic graphical models, can be expensive to store, slow to evaluate and hard to integrate into larger systems. Ideally, we would like to replace such cumbersome models with simpler models that perform equally well.

In this thesis, we study knowledge distillation, the idea of extracting the knowledge contained in a complex model and injecting it into a more convenient model. We present a general framework for knowledge distillation, whereby a convenient model of our choosing learns how to mimic a complex model, by observing the latter's behaviour and being penalized whenever it fails to reproduce it.

We develop our framework within the context of three distinct machine learning applications: (a) model compression, where we compress large discriminative models, such as ensembles of neural networks, into models of much smaller size; (b) compact predictive distributions for Bayesian inference, where we distil large bags of MCMC samples into compact predictive distributions in closed form; (c) intractable generative models, where we distil unnormalizable models such as RBMs into tractable models such as NADEs.

We contribute to the state of the art with novel techniques and ideas. In model compression, we describe and implement derivative matching, which allows for better distillation when data is scarce. In compact predictive distributions, we introduce online distillation, which allows for significant savings in memory. Finally, in intractable generative models, we show how to use distilled models to robustly estimate intractable quantities of the original model, such as its intractable partition function.

## Acknowledgements

This work was supervised by Dr Iain Murray. My long discussions with Iain have considerably expanded my knowledge of machine learning and his detailed feedback has greatly improved my writing. I would not have known how to recognize bad kerning had it not been for him. I consider myself fortunate to have been in the privileged group of individuals that have been supervised by Iain.

I am grateful to all fellow PhD students in the Institute for Adaptive and Neural Computation and the Centre for Doctoral Training in Data Science with whom I have had numerous discussions about machine learning (and beyond!). Special thanks go to Harri Edwards and Krzysztof Geras for regularly pointing me to good papers.

None of this would have existed had it not been for the directors of the Centre for Doctoral Training in Data Science, Prof. Chris Williams and Dr Charles Sutton. Many thanks to them for our regular interactions and making everything possible.

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh, and by Microsoft Research through its PhD Scholarship Programme.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                                 | <b>6</b>  |
| 1.1      | Model compression . . . . .                         | 7         |
| 1.2      | Compact predictive distributions . . . . .          | 7         |
| 1.3      | Intractable generative models . . . . .             | 8         |
| <b>2</b> | <b>Model Compression</b>                            | <b>9</b>  |
| 2.1      | A framework for model compression . . . . .         | 10        |
| 2.1.1    | Matching function values . . . . .                  | 10        |
| 2.1.2    | Matching function derivatives . . . . .             | 11        |
| 2.2      | Case study: compressing a neural ensemble . . . . . | 14        |
| 2.2.1    | The neural ensemble . . . . .                       | 14        |
| 2.2.2    | Loss functions . . . . .                            | 14        |
| 2.2.3    | Data generators . . . . .                           | 16        |
| 2.2.4    | Results and discussion . . . . .                    | 18        |
| 2.3      | Related work . . . . .                              | 23        |
| 2.3.1    | Optimal brain damage . . . . .                      | 23        |
| 2.3.2    | Model compression . . . . .                         | 24        |
| 2.3.3    | Mimicking neural networks . . . . .                 | 24        |
| 2.3.4    | Knowledge distillation . . . . .                    | 25        |
| 2.3.5    | FitNets . . . . .                                   | 25        |
| 2.3.6    | Matching derivatives . . . . .                      | 25        |
| 2.4      | Summary and conclusions . . . . .                   | 27        |
| <b>3</b> | <b>Compact Predictive Distributions</b>             | <b>28</b> |
| 3.1      | Distilling bags of MCMC samples . . . . .           | 29        |
| 3.2      | Bayesian density estimation . . . . .               | 30        |
| 3.2.1    | Batch distillation . . . . .                        | 30        |
| 3.2.2    | Online distillation . . . . .                       | 30        |
| 3.2.3    | Case study: Bayesian Mixture of Gaussians . . . . . | 31        |
| 3.3      | Bayesian binary classification . . . . .            | 36        |
| 3.3.1    | Loss functions . . . . .                            | 37        |
| 3.3.2    | Batch distillation . . . . .                        | 39        |
| 3.3.3    | Online distillation . . . . .                       | 39        |
| 3.3.4    | Case study: Bayesian logistic regression . . . . .  | 41        |
| 3.4      | Related work . . . . .                              | 46        |
| 3.4.1    | Distillation-based approaches . . . . .             | 46        |
| 3.4.2    | Deterministic approximate inference . . . . .       | 46        |
| 3.5      | Summary and conclusions . . . . .                   | 47        |

|   |           |
|---|-----------|
| <b>4 Intractable Generative Models</b>                                | <b>48</b> |
| 4.1 Distilling intractable into tractable generative models . . . . . | 49        |
| 4.1.1 Loss functions . . . . .  | 49        |
| 4.1.2 Stochastic gradient training . . . . .                          | 51        |
| 4.2 Case study: distilling RBM into NADE . . . . .                    | 53        |
| 4.2.1 The Restricted Boltzmann Machine . . . . .                      | 53        |
| 4.2.2 The Neural Autoregressive Distribution Estimator . . . . .      | 54        |
| 4.2.3 Distilling an RBM into NADE . . . . .                           | 56        |
| 4.2.4 Estimating the partition function of the RBM . . . . .          | 62        |
| 4.3 Related work and extensions . . . . .                             | 66        |
| 4.3.1 Variational inference . . . . .                                 | 66        |
| 4.3.2 Generative adversarial training . . . . .                       | 67        |
| 4.4 Summary and conclusions . . . . .                                 | 67        |
| <b>5 Conclusions</b>  | <b>69</b> |
| 5.1 Knowledge distillation: is it worth it? . . . . .                 | 69        |
| 5.2 What is there next? . . . . .                                     | 70        |
| <b>A The R Technique</b>  | <b>72</b> |
| <b>B Derivatives in Neural Networks</b>                               | <b>73</b> |
| B.1 Computing the output: forward propagation . . . . .               | 73        |
| B.2 Computing first derivatives: backward propagation . . . . .       | 73        |
| B.3 Computing second derivatives: R{backprop} . . . . .               | 74        |
| B.4 Common nonlinearities . . . . .                                   | 75        |
| B.4.1 Linear . . . . .  | 75        |
| B.4.2 Rectified linear . . . . .                                      | 76        |
| B.4.3 Logistic . . . . .  | 76        |
| B.4.4 Probit . . . . .  | 76        |
| B.4.5 Softmax and log-softmax . . . . .                               | 77        |
| B.5 Useful output functions . . . . .                                 | 77        |
| B.5.1 Square error . . . . .  | 78        |
| B.5.2 Cross entropy . . . . .   | 78        |
| B.5.3 Dot product . . . . .   | 78        |
| <b>C Derivatives in NADE</b>  | <b>79</b> |
| C.1 Computing the output: forward propagation . . . . .               | 79        |
| C.2 Computing first derivatives: backward propagation . . . . .       | 79        |
| C.3 Computing second derivatives: R{backprop} . . . . .               | 80        |
| <b>D Batch and Online EM for MoG</b>                                  | <b>82</b> |
| D.1 Batch EM . . . . .  | 82        |
| D.2 Online EM . . . . .   | 83        |
| <b>E Notations and Symbols</b>  | <b>85</b> |
| <b>F List of Abbreviations</b>  | <b>86</b> |
| <b>Bibliography</b>   | <b>87</b> |
| <b>Index</b>  | <b>91</b> |

# Chapter 1

## Introduction

Recent progress in machine learning has been phenomenal. Record after record is being broken, as models are becoming increasingly more accurate and better performing (Bojar et al., 2014; Russakovsky et al., 2015). This has been made possible by a series of theoretic and algorithmic breakthroughs in the last decade, and further facilitated by significant technological advances in computer hardware and software, the explosion of available data, and a renewed commercial interest in the field.

The above improvements have been accompanied by a notable increase in model size and complexity. Large ensembles consisting of several individual models, deep neural architectures with complex multilayer structure, large-scale probabilistic graphical models with complicated interactions between variables, and sophisticated generative models of data are only some examples of this ongoing trend. Clearly, increased model size and complexity has played a major role in the success of machine learning models.

Nevertheless, a large and complex model can give rise to significant practical challenges. Deep neural networks with several hidden layers can be slow to evaluate and expensive to store. Inference in large, hierarchical probabilistic models can be computationally demanding, even when done approximately. Adding sophisticated structure to generative models may compromise their tractability, and as a consequence reduce their interpretability and make them harder to integrate into larger systems. Size and complexity may present models with power, but they can also make them cumbersome and inconvenient to use.

Let us suppose for a moment that we have access to an accurate and powerful model, which however is cumbersome and inconvenient to use. Imagine that we could replace it with a convenient model that does the same job and is just as good. From then on, for all practical purposes, we could just get rid of the cumbersome model and use the convenient model instead. The question then becomes: *using the cumbersome model, can we actually construct a convenient model that does the same job and is just as good?*

In this thesis, we address the above question, and present a framework for constructing such convenient models from cumbersome ones. We call this procedure *knowledge distillation*; as if we could extract the knowledge hidden inside the cumbersome model, distil it, and inject it into the convenient model. From a high-level perspective, our framework is based on training the convenient model to mimic the cumbersome model, by getting it to observe its behaviour, and punish it when it fails to replicate it successfully.

Our work brings together ideas often referred to in the literature as knowledge distillation (Hinton et al., 2015), model compression (Bucilă et al., 2006), compact approximations (Snelson and Ghahramani, 2005a), mimicking models (Ba and Caruana, 2014), and teacher-student models (Romero et al., 2015). As a technical term, “knowledge distillation” was introduced only recently by Hinton et al. (2015), who used it to mean model compression. Here, we use it more broadly to refer to the act of transferring the knowledge of a cumbersome model into a convenient one. We believe that this usage of the term is more appropriate, since it permits a unifying description of tasks that are not necessarily limited to model compression.

The above description of knowledge distillation is very general; the words “cumbersome” and

“convenient” can mean different things in different contexts. For instance, for a deep neural network, cumbersome might mean “too large”, in which case a convenient model would be a small neural network. On the other hand, for a generative model, cumbersome might mean “intractable”, in which case a convenient model would be a tractable generative model.

In this work, we apply the aforementioned idea of knowledge distillation into three fairly general—and highly important—fields of machine learning: *discriminative models*, *Bayesian inference* and *generative models*. For each one, we define the meaning of “cumbersome” and “convenient”, exemplify our knowledge distillation framework, and apply it to solve an example problem within the field.

The rest of the thesis is organized into three chapters, one for each field in which we apply the knowledge distillation framework. Each chapter is standalone; that is, it can be read independently and does not require knowledge of the others. The following three sections provide an overview of each chapter and list our individual contributions in each one.

## 1.1 Model compression

In chapter 2, we apply knowledge distillation on discriminative models. In this particular context, knowledge distillation becomes synonymous with model compression. That is, the cumbersome model is a large discriminative model, such as an ensemble or deep neural networks, which, due to its size, is expensive to store and slow to evaluate. The objective is to distil it into a smaller model that would be cheaper to store and evaluate, but would not seriously compromise accuracy and performance. Our particular contributions in this chapter are listed below.

- (i) We present a generic and flexible framework for model compression, which is viewed as a special case of knowledge distillation. Our framework is general enough to be applied to a wide range of models, and it extends the work of Bucilă et al. (2006).
- (ii) We successfully apply the above framework in compressing an ensemble of neural networks into a single small neural network.
- (iii) We make a comprehensive review of the model compression literature and compare it to our approach.
- (iv) We extend the model compression literature in two distinct ways. Firstly, we show how to efficiently use derivative information in the distillation procedure. Secondly, we make use of generative models, such as NADE (Larochelle and Murray, 2011), for providing input data for distillation. We show that both approaches lead to better generalization performance when the input data is limited.

## 1.2 Compact predictive distributions

In chapter 3, we apply knowledge distillation on approximate Bayesian inference with Markov Chain Monte Carlo (Neal, 1993). Here, the cumbersome model is a Markov chain that explores a posterior distribution and generates MCMC samples from it. Averaging over those samples approximates the true predictive distribution. If high accuracy is required, then a large number of MCMC samples needs to be generated and stored, which is an inefficient and wasteful way to represent the predictive distribution. The convenient model that is distilled from the Markov chain is a compact closed-form approximation to the true predictive distribution, which it is cheaper to store and evaluate compared to a bag of MCMC samples. Our contributions in this chapter are listed below.

- (i) We present a framework for learning compact predictive distributions from Markov chains, which extends that of Snelson and Ghahramani (2005a).

- (ii) We apply our framework to the problems of Bayesian density estimation and Bayesian binary classification. We showcase it in the special cases of Bayesian Mixtures of Gaussians and Bayesian logistic regression respectively.
- (iii) We show how distillation in our framework can be performed in an online fashion, that is, while the Markov chain is being simulated. This leads to a form of online distillation that performs as well as batch distillation, but requires only a fraction of the memory.
- (iv) We show how to efficiently include derivatives within the distillation procedure. This way, we manage to achieve better distillation, by conveying more information about the true predictive distribution to the compact one.

### **1.3 Intractable generative models**

In chapter 4, we focus on generative models. Here, the cumbersome model is a generative model that cannot be normalized, hence tasks such as calculating its partition function, evaluating its probabilities, marginalizing and conditioning are all intractable. The task is to distil it into a tractable model, for which doing all the above would be tractable. This way, the convenient tractable model can be used within a larger system in ways that the cumbersome intractable model could not. Our particular contributions in this chapter are listed below.

- (i) We present a framework for distilling intractable generative models into tractable ones. Our framework manages to extract the knowledge contained in the intractable model, bypassing the need to calculate its intractable partition function.
- (ii) We apply our framework to successfully distil an intractable Restricted Boltzmann Machine (Smolensky, 1986) into a tractable NADE.
- (iii) We show that the distilled NADE can be successfully used to produce robust estimates of the intractable partition function of the RBM, that are comparable to the state of the art estimates provided by Salakhutdinov and Murray (2008).

## Chapter 2

# Model Compression

Discriminative models play a dominant role in machine learning. From a high-level point of view, these are models that take as input some datapoint  $\mathbf{x}$  and output some prediction  $\mathbf{y}$  that is a function  $t$  of  $\mathbf{x}$ , that is

$$\mathbf{y} = t(\mathbf{x}). \quad (2.1)$$

Variable  $\mathbf{x}$  can represent any sort of data, such as speech, images or video. In the simplest case, variable  $\mathbf{y}$  can be a numerical prediction (in which case the model is used for regression), or a categorical prediction (in which case the model is used for classification).

Increasing the capacity of a discriminative model typically makes it capable of learning more complex functions. Consider for example the case of *neural networks*; in the last decade, neural networks have significantly increased in size and have become deeper. This has been made possible by having access to bigger datasets, more powerful computers (such as GPUs) and the invention of strong regularization methods, such as dropout (Srivastava et al., 2014). As a result, deep neural networks have produced state of the art results in many applications, such as automatic speech recognition (Graves et al., 2013), machine translation (Bahdanau et al., 2015; Sutskever et al., 2014) and image classification (Ciresan et al., 2012; Szegedy et al., 2015).

A different way of improving discriminative models is to combine several of them into a single model, which outputs in some sense the “average” or the “consensus” across models. This sort of model is known as an *ensemble*. The intuition why this works is that each separate model might make errors on different inputs, but by combining predictions from various models these errors tend to be overruled, hence the final predictions tend to be better. A prominent example of a model that owes its success to being an ensemble is the *random forest* (Breiman, 2001). In a random forest, a large number of decision trees are combined together, each of which is a weak model but altogether make a strong model. In general, the individual models need not be of the same class, and forming ensembles can also be done by combining fundamentally different kinds of models (Caruana et al., 2004).

Increasing the capacity of a single model or combining several models into an ensemble typically improves predictions; however it comes at a cost. The final model might be too large to store, or too slow to evaluate at test time. It is natural then to ask the question, is the extra size really necessary, or could the same function in principle be represented by a simpler, smaller model? If this is indeed the case, could we then distil the knowledge of the large model into a smaller model that does the same job?

Motivated by the above questions, the field of *model compression* emerged, pioneered by Bucilă et al. (2006). Having a accurate but large model  $t(\mathbf{x})$ , the objective of model compression is to train a smaller model  $f(\mathbf{x} | \boldsymbol{\theta})$  to mimic  $t(\mathbf{x})$  as closely as possible. In this chapter, we study model compression as a special case of knowledge distillation. We describe a general framework for model compression, and we study the effect of various techniques within the framework. We contribute to the state of the art with two novel ideas: matching derivatives and guiding the distillation process with a generative model of the input data. Finally, we showcase our framework by compressing an ensemble of large neural networks into a single small neural network.

## 2.1 A framework for model compression

Suppose we are given a discriminative model which computes the function  $\mathbf{y} = t(\mathbf{x})$ . We shall refer to this model as the “teacher”. We then specify a family of models  $f(\mathbf{x} | \boldsymbol{\theta})$ , parameterized by a set of parameters  $\boldsymbol{\theta}$ , which we shall refer to as the “student”. Our goal is to train the student to be as similar as possible to the teacher; that is, we want to find the value of  $\boldsymbol{\theta}$  for which the approximation

$$t(\mathbf{x}) \approx f(\mathbf{x} | \boldsymbol{\theta}) \quad (2.2)$$

becomes as close to an equality as possible.

In the context of model compression,  $t(\mathbf{x})$  is typically significantly more complex than  $f(\mathbf{x} | \boldsymbol{\theta})$ , therefore, in general, we cannot expect the student to match the teacher everywhere. The hope instead is that the particular function that is represented by  $t(\mathbf{x})$  is simple enough to be closely approximated by  $f(\mathbf{x} | \boldsymbol{\theta})$  at least in the region of space we are interested in. In practical applications, the region of interest may be tiny or significantly lower-dimensional compared to the whole space. For instance, if  $\mathbf{x}$  represents greyscale images, and  $t(\mathbf{x})$  is meant to be used to classify images of handwritten digits, then we are indeed interested only in a minuscule fraction of all greyscale images imaginable.

Our framework for model compression, besides the teacher  $t(\mathbf{x})$  and the student  $f(\mathbf{x} | \boldsymbol{\theta})$ , also involves a data generator  $p(\mathbf{x})$  and a loss  $E(\mathbf{x}, \boldsymbol{\theta})$ . The data generator is a probability distribution over the input space; in practice it can be any procedure which can generate datapoints on demand. The loss  $E(\mathbf{x}, \boldsymbol{\theta})$  is a measure of discrepancy between the teacher and the student at input  $\mathbf{x}$  and for parameters  $\boldsymbol{\theta}$ . The training objective is to minimize the average loss with respect to  $\boldsymbol{\theta}$ , which can be written as

$$\min_{\boldsymbol{\theta}} \langle E(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}. \quad (2.3)$$

The choice of the data generator and of the loss is highly important. The data generator specifies the region of the input space we are interested in. Agreement between the teacher and the student on datapoints  $\mathbf{x}$  for which  $p(\mathbf{x})$  is high is prioritized, since the loss is weighted by  $p(\mathbf{x})$  in the training objective. On the other hand, the loss specifies what it means for the student to agree with the teacher. In our case study in section 2.2, we will study the effect of different choices of data generator and loss, when compressing an ensemble of neural networks. In the following two sections, we consider two broad families of losses, and describe a stochastic way of minimizing them.

### 2.1.1 Matching function values

The simplest choice for the loss is a function that directly measures the difference between the predictions of the teacher  $t(\mathbf{x})$  and the predictions of the student  $f(\mathbf{x} | \boldsymbol{\theta})$  at input  $\mathbf{x}$ . For a regression task, such a loss can be the square error, defined by

$$E_{\text{SE}}(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2} \|f(\mathbf{x} | \boldsymbol{\theta}) - t(\mathbf{x})\|^2. \quad (2.4)$$

For a classification task, where the outputs of both models are discrete probability distributions with  $I$  elements, a suitable loss is the cross entropy, defined by

$$E_{\text{CE}}(\mathbf{x}, \boldsymbol{\theta}) = - \sum_i t_i(\mathbf{x}) \log f_i(\mathbf{x} | \boldsymbol{\theta}). \quad (2.5)$$

Such losses only involve function values  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$ , and they encourage those values to be as similar as possible.

We will now describe a stochastic gradient training procedure for minimizing the average loss  $E(\boldsymbol{\theta}) = \langle E(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}$ , when the loss is of the above type. The gradient of  $E(\boldsymbol{\theta})$  with respect to the parameters  $\boldsymbol{\theta}$  is

$$\mathbf{g}(\boldsymbol{\theta}) = \left\langle \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta}) \right\rangle_{p(\mathbf{x})} \quad (2.6)$$

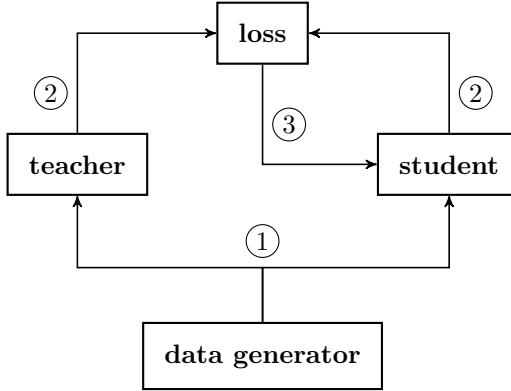


Figure 2.1: Overview of stochastic gradient training for model compression. The generator produces datapoints (1) on which both the teacher and the student are evaluated (2). The loss measures the discrepancy between the predictions of the teacher and the student, and its gradient with respect to the student’s parameters provides training signal for the student (3).

The above gradient can be stochastically approximated by

$$\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{S} \sum_s \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_s, \boldsymbol{\theta}), \quad (2.7)$$

where  $\{\mathbf{x}_s\}$  are samples generated from  $p(\mathbf{x})$ . It is easy to see that, in expectation, the stochastic gradient is equal to the original gradient. Using the above stochastic gradient, we can minimize  $E(\boldsymbol{\theta})$  using the following procedure, which is shown pictorially in Figure 2.1.

- (i) Generate a minibatch  $\{\mathbf{x}_s\}$  of size  $S$  from  $p(\mathbf{x})$ .
- (ii) Calculate  $t(\mathbf{x}_s)$  and  $f(\mathbf{x}_s | \boldsymbol{\theta})$  for all datapoints in the minibatch.
- (iii) Calculate the stochastic gradient  $\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{S} \sum_s \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_s, \boldsymbol{\theta})$ .
- (iv) Make an update on  $\boldsymbol{\theta}$  using  $\hat{\mathbf{g}}(\boldsymbol{\theta})$ .
- (v) Repeat until convergence.

The above algorithm falls into the category of stochastic optimization methods. Provided the samples generated from  $p(\mathbf{x})$  are exact and independent, and under certain regularity conditions, it is guaranteed to converge almost surely to a stationary point of  $E(\boldsymbol{\theta})$  (Bottou, 1998).

The above procedure is fairly general and assumes very little about  $p(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$ . Notice that we do not need to be able to evaluate  $p(\mathbf{x})$ , but only to be able to sample from it. Hence  $p(\mathbf{x})$  can be any data generating procedure, such as resampling a dataset. On the other hand, the only thing we assume for  $f(\mathbf{x} | \boldsymbol{\theta})$  is that it must be differentiable with respect to  $\boldsymbol{\theta}$ , so as to be able to calculate the gradient in step (iii). If  $f(\mathbf{x} | \boldsymbol{\theta})$  is a neural network, this can be done by backward propagation, as described in more detail in appendix B.

### 2.1.2 Matching function derivatives

In the previous section, we considered losses that only involve function values  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$ . We then described a training procedure, whereby the function represented by the teacher is probed at various input locations, and the student is then adjusted to match the teacher’s function values

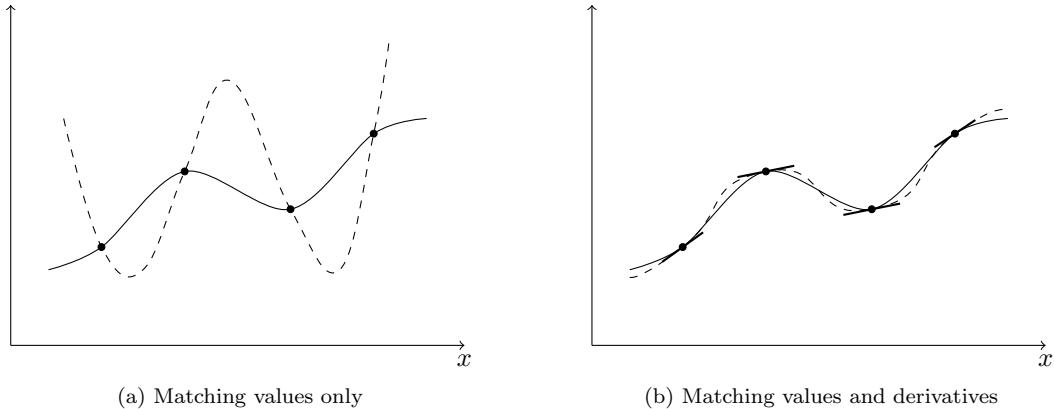


Figure 2.2: Motivational sketch for matching derivatives. Using the same number of input locations, matching derivatives further constrains the student (dashed line) to better match the teacher (solid line). Based on Figure 2 by Simard et al. (1992).

at those locations. In this sense, we reduced model compression to a regression problem, where train data were generated by  $p(\mathbf{x})$  and labelled by  $t(\mathbf{x})$ .

However, since we have access to the full function represented by the teacher, every time we probe it at some location to calculate its value, we can go one step further and also calculate its derivative with respect to  $\mathbf{x}$ . In principle, calculating the derivative of any function is possible at a cost which is at most a constant factor greater than the cost of calculating a single function value (Griewank, 2012).<sup>1</sup> We can then encourage the student to not only match the teacher’s function values at some input location, but also the whole tangent hyperplane specified by its derivative with respect to the input at the same location. Thus, we can further constrain the student using the same number of input locations, as illustrated in Figure 2.2.

In order to train the student by matching derivatives, we need to include  $\frac{\partial t}{\partial \mathbf{x}}$  and  $\frac{\partial f}{\partial \mathbf{x}}$  in the loss, so as to minimize the discrepancy between them. For instance, assuming that both the teacher and the student have  $I$  outputs, a loss that measures the discrepancy between derivatives is the derivative square error, defined as follows

$$E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2I} \sum_i \left\| \frac{\partial f_i}{\partial \mathbf{x}} - \frac{\partial t_i}{\partial \mathbf{x}} \right\|^2. \quad (2.8)$$

In order to apply the same stochastic gradient training procedure with losses that involve derivatives with respect to  $\mathbf{x}$  such as the above, we need to be able to calculate the derivatives of the loss with respect to  $\theta$ . As we will show, this is always possible at a cost which is at most a constant factor greater than the cost of using losses without derivatives. This is surprising, since one might expect that the cost of including derivatives would scale badly with the input dimensionality. However, it turns out that this is not the case.

To see why it is not the case, let both  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$  be functions of  $I$  outputs, and  $E(\mathbf{x}, \boldsymbol{\theta})$  be a function of only  $\frac{\partial t_i}{\partial \mathbf{x}}$  and  $\frac{\partial f_i}{\partial \mathbf{x}}$  for all  $i$ . Then, the derivative of the loss with respect to  $\boldsymbol{\theta}$  becomes

$$\frac{\partial}{\partial \theta} E(\mathbf{x}, \theta) = \sum_i \frac{\partial^2 f_i}{\partial \theta \partial \mathbf{x}} \frac{\partial E}{\partial \mathbf{d}_i}, \quad (2.9)$$

where  $\mathbf{d}_i = \frac{\partial f_i}{\partial \mathbf{x}}$ . In the above expression,  $\frac{\partial^2 f_i}{\partial \theta \partial \mathbf{x}}$  is part of the Hessian of  $f_i$  and  $\frac{\partial E}{\partial \mathbf{d}_i}$  is some vector that depends on  $\frac{\partial t}{\partial \mathbf{x}}$  and  $\frac{\partial f}{\partial \mathbf{x}}$ . The key thing to notice here is that the above expression only involves the Hessian in the form of Hessian-vector products. Despite the full Hessian needing quadratic time to be calculated, Hessian-vector products can be directly calculated in only linear

---

<sup>1</sup>This result is commonly known as the *cheap gradient principle*.

time, using a method known as the *R technique* (Pearlmutter, 1994). The R technique, as described in detail in appendix A, avoids calculating the full Hessian by introducing the operator  $\mathcal{R}\{\cdot\}$ , which is defined as

$$\mathcal{R}\{\mathbf{g}\} = \left( \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^T \mathbf{v}. \quad (2.10)$$

Making use of the R operator and identifying  $\mathbf{g} = \frac{\partial f_i}{\partial \boldsymbol{\theta}}$  and  $\mathbf{v} = \frac{\partial E}{\partial \mathbf{d}_i}$ , we can write the derivative of  $E(\mathbf{x}, \boldsymbol{\theta})$  with respect to  $\boldsymbol{\theta}$  as

$$\frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta}) = \sum_i \mathcal{R}\left\{ \frac{\partial f_i}{\partial \boldsymbol{\theta}} \right\}. \quad (2.11)$$

For example, when the loss is the derivative square error, we have

$$\frac{\partial}{\partial \boldsymbol{\theta}} E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{I} \sum_i \frac{\partial^2 f_i}{\partial \boldsymbol{\theta} \partial \mathbf{x}} \left( \frac{\partial f_i}{\partial \mathbf{x}} - \frac{\partial t_i}{\partial \mathbf{x}} \right) = \frac{1}{I} \sum_i \mathcal{R}\left\{ \frac{\partial f_i}{\partial \boldsymbol{\theta}} \right\}. \quad (2.12)$$

In the above, the vector each Hessian  $\frac{\partial^2 f_i}{\partial \boldsymbol{\theta} \partial \mathbf{x}}$  is multiplied by is

$$\mathbf{v}_i = \frac{\partial f_i}{\partial \mathbf{x}} - \frac{\partial t_i}{\partial \mathbf{x}}. \quad (2.13)$$

Putting all the above together, the stochastic gradient training procedure for losses that involve derivatives with respect to the input is described below.

- (i) Generate a minibatch  $\{\mathbf{x}_s\}$  of size  $S$  from  $p(\mathbf{x})$ .
- (ii) Calculate  $\frac{\partial t}{\partial \mathbf{x}}$  and  $\frac{\partial f}{\partial \mathbf{x}}$  for all datapoints in the minibatch.
- (iii) For each output  $i$  and for each  $\mathbf{x}_s$ , calculate  $\mathbf{v}_i = \frac{\partial E}{\partial \mathbf{d}_i}$  and make use of the R technique (appendix A) to calculate the Hessian-vector product  $\mathcal{R}\left\{ \frac{\partial f_i}{\partial \boldsymbol{\theta}} \right\} = \frac{\partial^2 f_i}{\partial \boldsymbol{\theta} \partial \mathbf{x}} \mathbf{v}_i$ .
- (iv) Calculate the stochastic gradient  $\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{S} \sum_s \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_s, \boldsymbol{\theta})$ .
- (v) Make an update on  $\boldsymbol{\theta}$  using  $\hat{\mathbf{g}}(\boldsymbol{\theta})$ .
- (vi) Repeat until convergence.

Under certain regularity conditions, and provided the samples from  $p(\mathbf{x})$  are exact and independent, the above algorithm is guaranteed to converge almost surely to a stationary point of the average loss (Bottou, 1998).

Notice that the above algorithm is fairly general, since the only requirement it has on the student is that  $f(\mathbf{x} | \boldsymbol{\theta})$  must be differentiable. If  $f(\mathbf{x} | \boldsymbol{\theta})$  is a neural network, then each  $\frac{\partial f_i}{\partial \boldsymbol{\theta}}$  can be calculated using backward propagation and each  $\mathcal{R}\left\{ \frac{\partial f_i}{\partial \boldsymbol{\theta}} \right\}$  can be calculated using  $\mathcal{R}\{\text{backprop}\}$ , as described in detail in appendix B.

Since backprop and  $\mathcal{R}\{\text{backprop}\}$  have roughly the same cost, calculating  $\frac{\partial f_i}{\partial \boldsymbol{\theta}}$  and  $\mathcal{R}\left\{ \frac{\partial f_i}{\partial \boldsymbol{\theta}} \right\}$  for all  $i$  can be done at the cost of  $2I$  backprops. In contrast, loss functions that only match function values require a single backprop per datapoint. Hence, including derivatives of functions with  $I$  outputs in the loss increases the computational cost roughly  $2I$  times compared to losses that do not involve derivatives. As a consequence, even though matching derivatives does not scale badly with the input dimensionality, it does scale linearly with the output dimensionality, therefore it is better suited to models with relatively few outputs.

## 2.2 Case study: compressing a neural ensemble

In this section, we put our model compression framework into practice, in order to compress an ensemble of neural networks into a single small neural network. The framework asks for the implementation of (a) a loss for measuring the discrepancy between the teacher and the student, and (b) an appropriate data generator for the input data. In the following, we describe the neural ensemble to be compressed, the losses and data generators we employed, and we present and discuss experimental results.

### 2.2.1 The neural ensemble

We trained an ensemble of 30 neural networks for handwritten digit classification on the MNIST dataset. The MNIST dataset (Le Cun et al.) is a dataset of greyscale  $28 \times 28$  images of handwritten digits (0 to 9), and it is commonly used as a benchmark for classification algorithms. The dataset is partitioned into a train set of 60,000 images, which we used to train the ensemble, and a test set of 10,000 images. The objective is to classify each image as being one of the 10 digits. Figure 2.3a shows some random samples from the MNIST train set.

The ensemble is composed of 30 neural networks of the same architecture. Each neural network has two hidden layers with 500 and 300 hidden units respectively, and an output layer of 10 units. The hidden layers use ReLU activation functions (as described in section B.4.2) and the output layer uses softmax activation functions (as described in section B.4.5). Each network is therefore of the following type

$$784 \xrightarrow{\text{ReLU}} 500 \xrightarrow{\text{ReLU}} 300 \xrightarrow{\text{softmax}} 10. \quad (2.14)$$

Each network in the ensemble was trained separately on a different dataset of 60,000 images, that was created by resampling the MNIST train set with replacement. We trained each network by minimizing cross entropy with stochastic gradient descent, using minibatches of 20 datapoints, and performing 20 passes over the dataset in total. We used ADADELTA (Zeiler, 2012) as a learning rate strategy, which uses a different learning rate for each network parameter and adapts them during training. ADADELTA is controlled by two hyperparameters, for which we used the values suggested in the original paper.

Given an image  $\mathbf{x}$ , each individual network  $n$  outputs a discrete probability distribution  $t^{(n)}(\mathbf{x})$  with 10 elements, corresponding to how likely the network thinks it is that  $\mathbf{x}$  belongs to each class. The predictions from all networks are then averaged together, in order to give a final prediction as follows

$$t(\mathbf{x}) = \frac{1}{N} \sum_n t^{(n)}(\mathbf{x}). \quad (2.15)$$

On the test set, the above ensemble achieves an accuracy of  $98.67\% \pm 0.23\%$  and an average log probability of  $-0.053 \pm 0.011$  nats. The error bars correspond to two standard deviations. These results are comparable to those reported in the literature for models of similar architecture (Le Cun et al.).

### 2.2.2 Loss functions

Our model compression framework requires the specification of a suitable loss  $E(\boldsymbol{\theta}) = \langle E(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}$  in order to measure the discrepancy between the neural ensemble  $t(\mathbf{x})$  and the student network  $f(\mathbf{x} | \boldsymbol{\theta})$ . As presented in section 2.1, our framework supports two kinds of losses; losses that (a) try to match function values, and losses that (b) try to match function derivatives. In our case study, we used two losses, namely *cross entropy*, which matches function values, and *derivative square error*, which matches function derivatives. The following two sections discuss these two losses in detail and establish their validity.

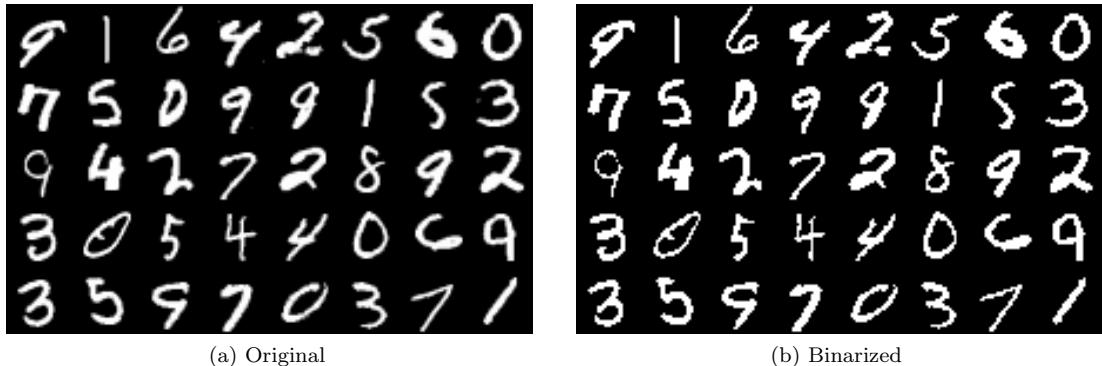


Figure 2.3: Samples from the MNIST dataset of handwritten digits. White corresponds to 1 and black to 0. The binary version was only used to train NADE.

### Cross entropy

In our case study, both the teacher  $t(\mathbf{x})$  and the student  $f(\mathbf{x}|\boldsymbol{\theta})$ , for each input location  $\mathbf{x}$ , output a vector of probabilities, which corresponds to how likely the models think it is that  $\mathbf{x}$  belongs to each class. A natural measure of discrepancy between  $t(\mathbf{x})$  and  $f(\mathbf{x}|\boldsymbol{\theta})$  would then be the KL divergence from the teacher to the student. For some input  $\mathbf{x}$ , this KL divergence can be written as

$$D_{\text{KL}}(t(\mathbf{x}) \parallel f(\mathbf{x}|\boldsymbol{\theta})) = \sum_i t_i(\mathbf{x}) \log t_i(\mathbf{x}) - \sum_i t_i(\mathbf{x}) \log f_i(\mathbf{x}|\boldsymbol{\theta}). \quad (2.16)$$

In the above expression, the term  $\sum_i t_i(\mathbf{x}) \log t_i(\mathbf{x})$  is the negative entropy of  $t(\mathbf{x})$  and is a constant with respect to  $\boldsymbol{\theta}$ . Therefore, minimizing the above KL divergence is equivalent to minimizing the quantity

$$E_{\text{CE}}(\mathbf{x}, \boldsymbol{\theta}) = - \sum_i t_i(\mathbf{x}) \log f_i(\mathbf{x}|\boldsymbol{\theta}), \quad (2.17)$$

which is known as *cross entropy*.

Cross entropy is a loss commonly used in training for classification problems. It only involves function values, so it can be optimized as described in section 2.1.1. It is well known that the KL divergence is always non-negative, and that it becomes zero if and only if  $t(\mathbf{x}) = f(\mathbf{x}|\boldsymbol{\theta})$  (MacKay, 2002, section 2.6). Therefore, assuming  $f(\mathbf{x}|\boldsymbol{\theta})$  has enough capacity,  $\langle E_{\text{CE}}(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}$  will achieve its global minimum if and only if  $t(\mathbf{x}) = f(\mathbf{x}|\boldsymbol{\theta})$  everywhere in the support of  $p(\mathbf{x})$ .

### Derivative square error

A different approach for making  $t(\mathbf{x})$  and  $f(\mathbf{x}|\boldsymbol{\theta})$  agree everywhere is by making sure that their derivatives with respect to  $\mathbf{x}$  match. To measure the discrepancy between these derivatives, we herein propose the *derivative square error* loss, which we define as follows

$$E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2I} \sum_i \left\| \frac{\partial}{\partial \mathbf{x}} \log f_i(\mathbf{x}|\boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log t_i(\mathbf{x}) \right\|^2. \quad (2.18)$$

Note that since both  $t(\mathbf{x})$  and  $f(\mathbf{x}|\boldsymbol{\theta})$  output probabilities, we choose to convert them into the log domain before taking their derivatives. This loss contains only derivatives, and therefore it can be optimized using the framework described in section 2.1.2.

In the general case, matching the derivatives of two arbitrary—but differentiable—functions makes them equal only up to an additive constant. Therefore, in general, only matching derivatives is not sufficient in itself to make the functions equal. However, in our case, where the functions are probability distributions, we can show that, under very mild conditions, matching derivatives

is actually sufficient for making the functions equal. The following proposition proves this claim, and provides the conditions that need to apply.

**Proposition 2.1.** *Assuming that  $f(\mathbf{x} | \boldsymbol{\theta})$  has sufficient capacity, and that there exist points  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_I$  in the support of  $p(\mathbf{x})$  for which  $t(\mathbf{x}_1), t(\mathbf{x}_2), \dots, t(\mathbf{x}_I)$  are linearly independent, then the global minimum of  $\langle E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}$  is achieved if and only if  $f(\mathbf{x} | \boldsymbol{\theta}) = t(\mathbf{x})$  everywhere in the support of  $p(\mathbf{x})$ .*

*Proof.* Let  $S_p$  be the support of  $p(\mathbf{x})$ , that is, the set of points  $\mathbf{x}$  for which  $p(\mathbf{x}) > 0$ . Notice that  $\langle E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} \geq 0$ , as the expectation of a non-negative quantity. If  $f(\mathbf{x} | \boldsymbol{\theta}) = t(\mathbf{x})$  in  $S_p$ , then obviously  $\langle E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} = 0$ , hence it is globally minimized. Conversely, if  $\langle E_{\text{DSE}}(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} = 0$ , then for all  $i$  and all  $\mathbf{x}$  in  $S_p$  we have

$$\frac{\partial}{\partial \mathbf{x}} \log f_i(\mathbf{x} | \boldsymbol{\theta}) = \frac{\partial}{\partial \mathbf{x}} \log t_i(\mathbf{x}) \Rightarrow f_i(\mathbf{x} | \boldsymbol{\theta}) = c_i t_i(\mathbf{x}), \quad (2.19)$$

for some positive constants  $c_i$ . Since both  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$  are probability distributions, we have that

$$\sum_i c_i t_i(\mathbf{x}) = \sum_i f_i(\mathbf{x} | \boldsymbol{\theta}) = 1 = \sum_i t_i(\mathbf{x}) \Rightarrow \sum_i (c_i - 1) t_i(\mathbf{x}) = 0. \quad (2.20)$$

The above can be written as  $\boldsymbol{\alpha}^T t(\mathbf{x}) = 0$ , where  $\boldsymbol{\alpha}$  is defined to be a vector of length  $I$  that has  $(c_i - 1)$  as its elements. Assume that  $\boldsymbol{\alpha} \neq \mathbf{0}$ . Since  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_I$  are all in  $S_p$ , we have that  $\boldsymbol{\alpha}$  is orthogonal to all  $t(\mathbf{x}_i)$ , hence the collection of all  $t(\mathbf{x}_i)$  together with  $\boldsymbol{\alpha}$  forms a set of linearly independent vectors. This is a contradiction, since it implies the existence of  $(I + 1)$  linearly independent vectors in an  $I$ -dimensional space. We have thus shown that  $\boldsymbol{\alpha} = \mathbf{0}$ , hence  $c_i = 1$  for all  $i$ , therefore  $f(\mathbf{x} | \boldsymbol{\theta}) = t(\mathbf{x})$  for all  $\mathbf{x}$  in  $S_p$ .  $\square$

The above proposition says that, as long as we can find  $I$  input locations for which  $t(\mathbf{x})$  are linearly independent, then derivative square error is a valid loss. For most practical problems of interest, this is quite a reasonable assumption to make. For example, in our case study, the neural ensemble was trained on MNIST with data from all 10 classes. Therefore, there must be 10 regions in the input space for which a different one of the outputs is close to 1, while all others are close to 0. If this is not the case, then the model would be a poor classifier and we would probably not want to compress it in the first place.

### 2.2.3 Data generators

In our model compression framework, we need to specify a data generator  $p(\mathbf{x})$ , whose job is to generate the input data used during training. The choice of data generator is highly important, as it determines the region of input space for which agreement between the teacher and the student is sought. In the following sections, we describe the three data generators we employed in our case study, namely *resampling the dataset*, *NADE* and *random noise*.

#### Resampling the dataset

If we have access to a dataset of input locations, then an obvious thing to do is to use it as is. The way this works is that during training, in each iteration, we sample a minibatch of points from the dataset.

Sampling from the dataset can be done either with or without replacement. With replacement, each point in the minibatch is independently and uniformly at random selected from the whole dataset. In this case, the data generator takes the following form

$$p(\mathbf{x}) = \frac{1}{N} \sum_n \delta(\mathbf{x} - \mathbf{x}_n), \quad (2.21)$$

where  $\delta(\cdot)$  is the delta function and  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$  are the points in the dataset. Without replacement, each point is sampled uniformly at random, but then removed from the dataset. The

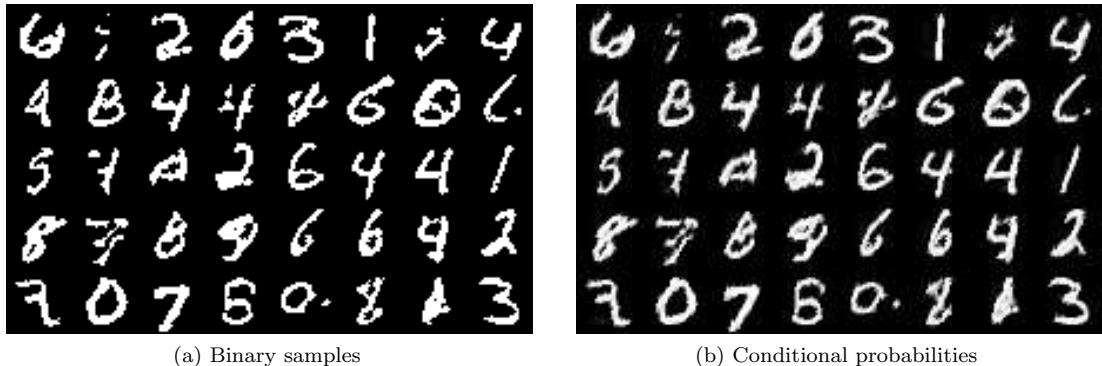


Figure 2.4: Samples from NADE trained on the full MNIST train set and the corresponding conditional probabilities. For model compression, the conditional probabilities were used instead of the actual samples.

sampling process resumes from the beginning when all the dataset has been sampled. This ensures that we see every other point in the dataset before seeing any point twice. In our experiments, we sampled the dataset without replacement.

In many real-life applications, labelled datasets are an expensive and potentially scarce resource, since it typically takes time and manual effort to do the labelling. However, unlabelled datasets may be abundant, especially in domains like vision and speech. Model compression is well-suited for such domains, since it does not require the dataset to be labelled; instead, all the labelling is done by the teacher model. In our experiments, we used the MNIST train set (or part of it), on which the neural ensemble was trained in the first place.

## NADE

Having access to a dataset is good, however it might not always be ideal. The dataset might be too small to cover the input space sufficiently, or if it is large, it might be expensive to store. And, no matter how large it is, it will always be finite. On the other hand, if we have access to a generative model from which we can generate data, we have essentially access to an infinite dataset, for limited storage cost.

We can imagine having a dataset to begin with, which is not sufficiently large to fully cover the input space, but is large enough to train a generative model on it. Then, we can use this generative model for model compression. Essentially, this way we achieve massive dataset expansion, while at the same time never having to store more than a minibatch of data, since minibatches are generated on the fly and thrown away after being used.

Motivated by the above, in our experiments we used the Neural Autoregressive Distribution Estimator (NADE for short), which was introduced by Larochelle and Murray (2011). NADE is a generative model whose likelihood is fully tractable, therefore it can be trained on a dataset using maximum likelihood. Also, generating from NADE produces samples that are exact and independent, therefore it is well-suited for our framework. Furthermore, NADE is fairly flexible, so it can fit well datasets like MNIST. NADE is extensively used in chapter 4, so we refer the reader to section 4.2.2 for more details on it.

In our experiments, we trained a NADE with 500 hidden units on the MNIST train set (or part of it). We used stochastic gradient descent, minibatches of size 20, and performed 10 passes over the train set. To set the learning rate strategy, we used ADADELTA (Zeiler, 2012), with its two hyperparameters set to the values suggested in the original paper. NADE is sensitive to the ordering of the input variables (which in this case are the pixel intensities). We used a raster columnwise ordering instead of a random ordering, which we found in practice to give a better generative model.

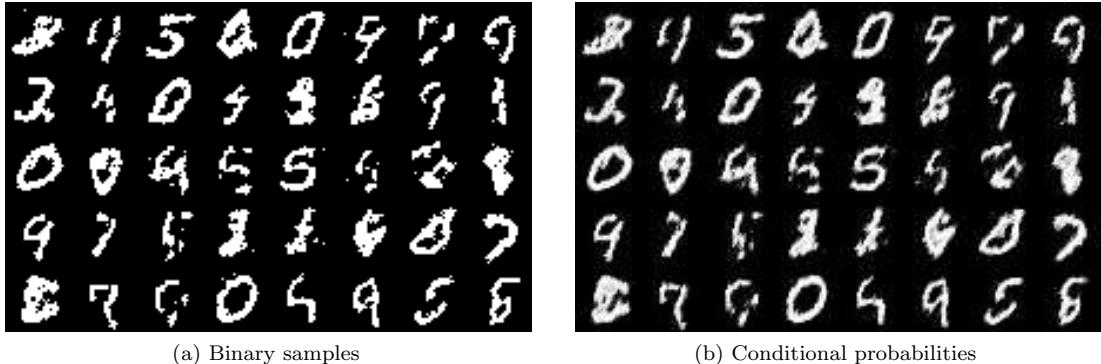


Figure 2.5: Samples from NADE trained on 10% of the MNIST train set and the corresponding conditional probabilities. For model compression, the conditional probabilities were used instead of the actual samples.

We used the binary version of NADE, which works only with binary inputs. To train it, we first binarized the MNIST train set by rounding its pixel intensities (which normally vary from 0 to 1). Figure 2.3b shows some binarized images from the MNIST train set. A problem with binary NADE is that the samples it generates are also binary. If these samples are used during model compression, this would limit severely the input space covered to the corners of a hypercube. However, together with samples, NADE also outputs the conditional probabilities of each pixel being turned on, given all generated pixels before it. These probabilities, varying from 0 to 1 typically resemble a “smoothed” version of the actual binary sample (see Figures 2.4 and 2.5). In all our experiments, we used the images formed by the conditional probabilities instead of the actual binary samples, since this gives a better coverage of the input space.<sup>2</sup>

### Random noise

In case we have neither a dataset nor a generative model of the input, we can always resort to using random noise as input. Nevertheless, generating inputs randomly is rarely a good idea. Typically, most randomly generated inputs will lie outside the regions of input space we are interested in, especially in high-dimensional spaces. In practice, it is rarely—if ever—the case that we are unable to build a data generator that produces inputs at least somewhat similar to those we are interested in, so we typically would not use random noise as input.

In our experiments, we used random Gaussian noise. That is, each image pixel was generated by a zero-mean, unit-variance Gaussian distribution. The main purpose of using random noise in our experiments is to demonstrate the importance of having a proper data generator when doing model compression.

#### 2.2.4 Results and discussion

We used our model compression framework to compress the neural ensemble into a single small neural network of the same type as the networks forming the ensemble, but with only 10% the number of hidden units. In other words, the student model is a neural network of type

$$784 \xrightarrow{\text{ReLU}} 50 \xrightarrow{\text{ReLU}} 30 \xrightarrow{\text{softmax}} 10. \quad (2.22)$$

We used the 2 losses (cross entropy and derivative square error) and the 3 data generators (resampling the dataset, NADE and random noise) as described in the previous sections, resulting into 6 different models. In all cases, we used minibatches of 20 samples, and the equivalent of 40 passes over the train data. We used ADADELTA (Zeiler, 2012) to adaptively set the learning

<sup>2</sup>Another hacky option would have been to postprocess the samples with a low-pass filter to smoothen them.

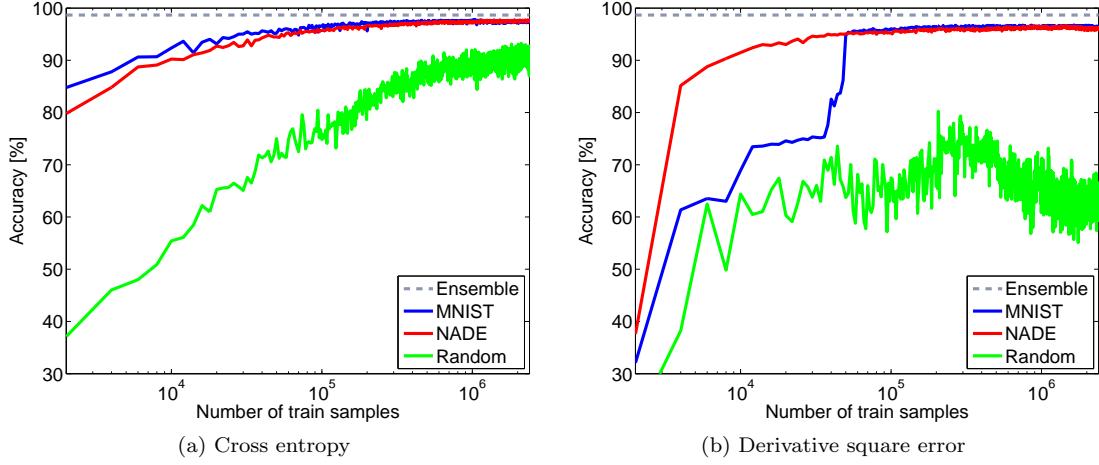


Figure 2.6: Training progress during compression of the neural ensemble, using the full MNIST train set. The plots show the accuracy of the student network on the MNIST test set for each loss and each data generator, as it progresses with the number of train samples. Note that the horizontal axis is in log scale.

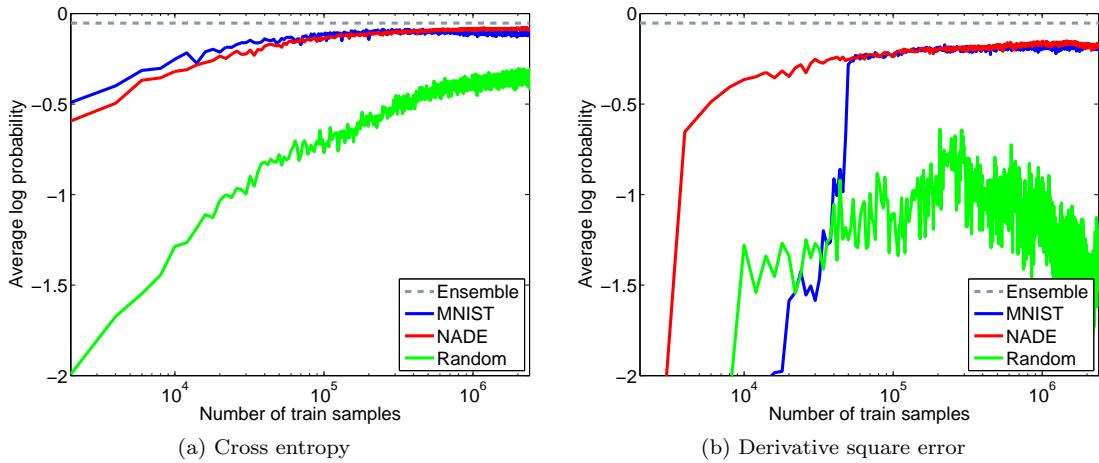


Figure 2.7: Training progress during compression of the neural ensemble, using the full MNIST train set. The plots show the average log probability of the student network on the MNIST test set for each loss and each data generator, as it progresses with the number of train samples. Note that the horizontal axis is in log scale.

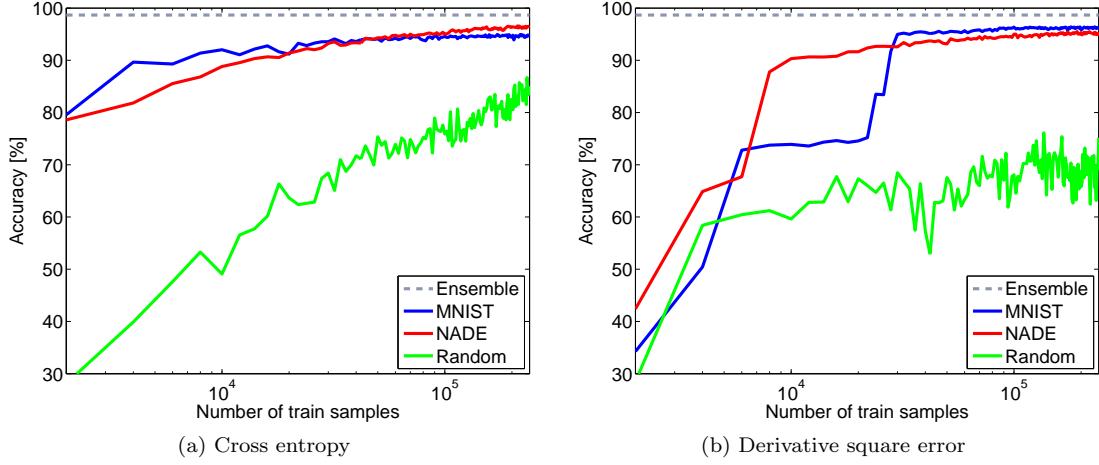


Figure 2.8: Training progress during compression of the neural ensemble, using 10% of the MNIST train set. The plots show the accuracy of the student network on the MNIST test set for each loss and each data generator, as it progresses with the number of train samples. Note that the horizontal axis is in log scale.

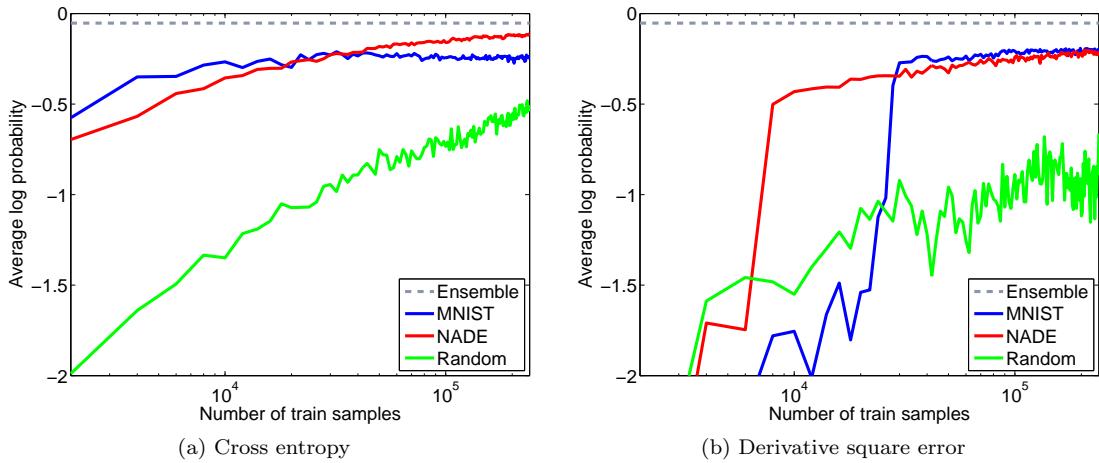


Figure 2.9: Training progress during compression of the neural ensemble, using 10% of the MNIST train set. The plots show the average log probability of the student network on the MNIST test set for each loss and each data generator, as it progresses with the number of train samples. Note that the horizontal axis is in log scale.

| Data generator        | Train data | Cross entropy    | Deriv. sq. error |
|-----------------------|------------|------------------|------------------|
| <b>MNIST</b>          | All        | $97.46 \pm 0.31$ | $96.50 \pm 0.37$ |
|                       | 10%        | $94.63 \pm 0.45$ | $96.36 \pm 0.37$ |
| <b>NADE</b>           | All        | $97.60 \pm 0.31$ | $96.27 \pm 0.38$ |
|                       | 10%        | $96.64 \pm 0.36$ | $95.33 \pm 0.42$ |
| <b>Gaussian noise</b> | All        | $91.98 \pm 0.54$ | $63.98 \pm 0.96$ |
|                       | 10%        | $85.04 \pm 0.71$ | $75.17 \pm 0.86$ |

Table 2.1: Performance of each compressed network, as measured by accuracy [%] on the MNIST test set. The accuracy of the original ensemble is  $98.67\% \pm 0.23\%$ . Error bars correspond to 2 standard deviations.

| Data generator        | Train data | Cross entropy      | Deriv. sq. error   |
|-----------------------|------------|--------------------|--------------------|
| <b>MNIST</b>          | All        | $-0.110 \pm 0.017$ | $-0.184 \pm 0.025$ |
|                       | 10%        | $-0.232 \pm 0.024$ | $-0.198 \pm 0.026$ |
| <b>NADE</b>           | All        | $-0.083 \pm 0.011$ | $-0.177 \pm 0.022$ |
|                       | 10%        | $-0.113 \pm 0.012$ | $-0.209 \pm 0.024$ |
| <b>Gaussian noise</b> | All        | $-0.323 \pm 0.011$ | $-1.328 \pm 0.044$ |
|                       | 10%        | $-0.507 \pm 0.014$ | $-0.662 \pm 0.025$ |

Table 2.2: Performance of each compressed network, as measured by average log probability on the MNIST test set. The average log probability of the original ensemble is  $-0.053 \pm 0.011$  nats. Error bars correspond to 2 standard deviations.

rate, with its two hyperparameters set to the values suggested in the original paper. Note that ADADELTA provides no convergence guarantees, however we found that it works well in practice, while it requires minimal tuning.

As already discussed, the MNIST train set was used both as a data generator and in order to train NADE. To measure the effect the size of the train set has on model compression, we performed the set of experiments twice. In the first set of experiments, we used the full MNIST train set, whereas in the second set, we only used 10% of it. We measure how effectively the neural ensemble was compressed by evaluating the accuracy and the average log probability of each compressed neural network on the MNIST test set. Figures 2.6, 2.7, 2.8 and 2.9 show the progress during training, and Tables 2.1 and 2.2 show the final results.

From the results we can see that, in all cases, using random Gaussian noise as data generator performs significantly worse than using MNIST or using NADE. This highlights the importance of the data generator for the model compression process. For the student model to be trained properly, it is vital that during training it sees input samples similar to those that it is going to be evaluated on. Still though, when cross entropy is used as a loss, even with random noise the student network classifies correctly about 9 in 10 images, and its performance appears to continue increasing with more training samples.

When using the MNIST train set or a NADE trained on it as data generators, the performance appears to be good. To confirm that this is indeed the case, we also trained two neural networks of the same type as the student network, but this time directly on the MNIST train set, using the true train labels. These networks were trained with the same hyperparameters we used for training the ensemble components. One was trained on the full MNIST train set and the other on

| Data generator        | Train data | Cross entropy    | Deriv. sq. error  |
|-----------------------|------------|------------------|-------------------|
| <b>MNIST</b>          | All        | $0.32 \pm 0.33$  | $-0.64 \pm 0.35$  |
|                       | 10%        | $0.20 \pm 0.39$  | $1.93 \pm 0.40$   |
| <b>NADE</b>           | All        | $0.46 \pm 0.30$  | $-0.87 \pm 0.38$  |
|                       | 10%        | $2.21 \pm 0.40$  | $0.90 \pm 0.44$   |
| <b>Gaussian noise</b> | All        | $-5.16 \pm 0.53$ | $-33.16 \pm 0.98$ |
|                       | 10%        | $-9.39 \pm 0.71$ | $-19.26 \pm 0.88$ |

Table 2.3: Average difference in accuracy [%] across examples in the MNIST test set, between compressed networks and networks trained directly on data. A positive number indicates that the compressed network has higher accuracy. Error bars correspond to 2 standard deviations.

| Data generator        | Train data | Cross entropy      | Deriv. sq. error   |
|-----------------------|------------|--------------------|--------------------|
| <b>MNIST</b>          | All        | $0.072 \pm 0.023$  | $-0.002 \pm 0.024$ |
|                       | 10%        | $0.089 \pm 0.024$  | $0.124 \pm 0.027$  |
| <b>NADE</b>           | All        | $0.099 \pm 0.022$  | $0.005 \pm 0.024$  |
|                       | 10%        | $0.209 \pm 0.029$  | $0.112 \pm 0.029$  |
| <b>Gaussian noise</b> | All        | $-0.141 \pm 0.026$ | $-1.146 \pm 0.048$ |
|                       | 10%        | $-0.185 \pm 0.031$ | $-0.341 \pm 0.034$ |

Table 2.4: Average difference in log probability across examples in the MNIST test set, between compressed networks and networks trained directly on data. A positive number indicates that the compressed network has higher log probability. Error bars correspond to 2 standard deviations.

10% of it. Their performance is shown in Table 2.5, and a paired comparison with the compressed networks is shown in Tables 2.3 and 2.4. We can see that in most cases the compressed networks have similar or better performance than the networks trained directly on data, especially when only 10% of the dataset was used. It is important to note that even when the accuracy appears to be similar, the log probability of the compressed networks is usually higher, suggesting that the networks trained directly on data are more prone to overfitting. Indeed, using the teacher model to perform the labelling provides soft instead of hard labels, effectively reducing the risk of overfitting.

Comparing cross entropy to derivative square error, we can see that when the full train set was used, cross entropy tends to perform better. However, when the amount of data was limited to 10%, the performance of cross entropy dropped significantly, whereas the performance of derivative square error dropped only slightly (as seen from Tables 2.1 and 2.2, most prominently when MNIST was used as a data generator). This suggests that using derivatives improves the generalization potential of the compressed network. Indeed, by matching derivatives, the student network tries to match a whole tangent hyperplane of 784 dimensions instead of a single scalar value. This means that every time the teacher model is probed, a lot more information is transferred to the student network about the shape of the function surface it represents. Furthermore, by passing the information about the derivatives, the teacher model effectively informs the student about how the output should change if the input is slightly perturbed in every possible direction. Due to this, the student network can generalize better without having to see more data. On the other hand, when the data is sufficient, it appears to be preferable to use function values instead.

| Train data | Accuracy [%]     | Av. log probability |
|------------|------------------|---------------------|
| All        | $97.14 \pm 0.33$ | $-0.182 \pm 0.028$  |
| 10%        | $94.43 \pm 0.46$ | $-0.321 \pm 0.035$  |

Table 2.5: Performance on the MNIST test set of two networks of the same type as the student network that were trained with cross entropy directly on data. Error bars correspond to 2 standard deviations.

It is interesting to compare NADE to MNIST as a data generators. When using MNIST, the student network sees the same input data over and over again. If the full dataset is used, then this is less of a problem, as shown by the fact that in this case the networks compressed with NADE perform similarly to the networks compressed with MNIST. However, when the size of the dataset is reduced, NADE appears to have a clear advantage. When using NADE, the student sees novel input data each time, and it does not recycle the same dataset over and over again. Note that NADE is also affected by the reduction in the train set, since it relies on it to be trained. This can be seen in Figures 2.4 and 2.5, which show random samples from the NADE trained on the full train set and the NADE trained on 10% of it. Clearly, the quality of samples deteriorates with the reduction in the train set. However, it appears that this affects NADE less than it affects the compression process.

To summarize our conclusions, (a) the choice of data generator is highly important, as shown by the poor performance of random noise, (b) having the teacher provide the labels reduces the risk of overfitting, (c) cross entropy works better than derivative square error if a lot of data is available, (d) derivative square error generalizes better than cross entropy when data is limited, and (e) model compression using NADE has a clear advantage over using the dataset when the latter is small.

## 2.3 Related work

There is a fairly rich literature related to model compression. In this section, we present the most important approaches to model compression that have been proposed so far, and we compare them to the approach adopted in the present thesis. We also review past work that provided inspiration for the methods discussed in this chapter.

### 2.3.1 Optimal brain damage

Optimal brain damage (Le Cun et al., 1990) is one of the first methods proposed for reducing the size of a neural network. The main idea is to take a trained neural network and start deleting parameters from it, by setting them to zero. This is equivalent to removing connections and/or neurons from the network, hence the term “brain damage”. Brain damage is done “optimally” in the sense that those parameters that matter less are deleted first. After deletion of a few parameters, the network is retrained and the process can be iterated as many times as desired.

In the context of optimal brain damage, how much a parameter matters is measured by the change the deletion of the parameter causes to the network’s loss function. In general, this change would be costly to measure exactly, however, it can be efficiently approximated by using the second derivatives of the loss function with respect to the parameters. The second derivatives can be easily calculated by a procedure similar to backward propagation (Becker and Le Cun, 1989).

Optimal brain damage is significantly different to knowledge distillation, and it is constrained by the fact that it can only compress neural networks to neural networks. In contrast, knowledge distillation can be used to compress any type of model to a model not necessarily of the same type. The only requirement of knowledge distillation is that the student model must be differentiable, which is a fairly weak one.

### 2.3.2 Model compression

The term “model compression” in the sense used in this chapter was first introduced in a seminal paper by Bucilă et al. (2006). Their work focuses on compressing large ensembles of a multitude of different models into a single neural network that has the capacity to approximate well the function represented by the ensemble. The main idea is to collect or create a large amount of unlabelled data, label it using the ensemble, and then treat it as train data to train the neural network on. As the authors mention, with this setup the neural network hardly runs any risk of overfitting, a fact we also observed in our experiments in section 2.2.4.

One of the main focuses of Bucilă et al. (2006) is how to obtain the large unlabelled data set needed. They discuss both the importance and the difficulty of ensuring that this dataset comes from the correct manifold of the input space, as we also discussed in this chapter. To achieve this, Bucilă et al. (2006) propose a method they call MUNGE, which, given a dataset, creates new input datapoints by combining and perturbing the original ones. They also experiment with generative models, but they use a weak Naive Bayes density estimator that is found to underperform.

Our model compression framework contains their method as a special case and it is largely inspired from it. Instead of creating synthetic data by modifying existing datasets however, in our framework we take a more principled approach and we propose to use flexible density estimators such as NADE trained on the original input data. Also, our implementation has low storage requirements, since labelling the input data by the teacher is done in minibatches, thanks to the stochastic way the student is trained.

### 2.3.3 Mimicking neural networks

In the field of automatic speech recognition, hidden Markov models that use large deep neural networks in calculating their emission probabilities have become considerably successful in recent years (Hinton et al., 2012). Such networks typically have a softmax output layer that produces a vector of probabilities, which correspond to the states of the Markov model. However, deploying large networks on devices like smart phones is impractical, due to limitations on both memory and processing power.

Motivated by this problem, Li et al. (2014) propose replacing the large deep neural networks found in existing automatic speech recognition systems with smaller ones that have been trained specifically to mimic them. Their method is based on training the small network by minimizing the KL divergence from the output of the large network to the output of the small one, averaged over an unlabelled dataset. Essentially, their method reduces to training the small network using cross entropy, with labels provided by the large network. Note that this method is not necessarily limited to neural networks, but can still only be applied to models that output discrete probability distributions. From this perspective, this work can be seen as a special case both of the model compression framework of Bucilă et al. (2006) and of our model compression framework described in this chapter, specifically applied to the domain of automatic speech recognition.

Though not strictly on model compression, another work related to training models to mimic other models is that of Ba and Caruana (2014). The focus there is to provide insights as to whether it is possible for shallow neural networks with a similar number of parameters to represent the functions learnt by deep, possibly convolutional, neural networks. In attempting to answer this question, they train shallow neural networks to mimic deep neural networks, by minimizing the discrepancy between them averaged on a large unlabelled dataset. They conclude that it is indeed possible to represent the same functions with shallow networks, and that the reason shallow networks are not typically used in practice is their tendency to overfit when trained directly on data.

In their work, they found that for networks with softmax outputs, matching the logits by minimizing square error was more effective than matching the outputs by minimizing cross entropy. This is a particularly interesting insight, as it indicates new potential ways of performing knowledge distillation. However, this method is only limited to a specific type of networks, namely those with softmax outputs.

### 2.3.4 Knowledge distillation

The term “knowledge distillation” originated from the recent work of Hinton et al. (2015). Whereas in this thesis we are using it more broadly to mean “transferring knowledge from a cumbersome model to a convenient model”, in their work they use it only in the context of model compression. They focus on neural networks, or ensembles thereof, that have a softmax output. Their goal is to compress them to smaller neural networks with a softmax output. Their method is in principle similar to the standard one of Bucilă et al. (2006), however they come up with a novel insight that leads to a slightly different approach.

Their main observation is that, in the output of the softmax layer, the relative magnitudes of the different components, even of those that are very small, contain a significant amount of knowledge. However, when training with cross entropy, the training signal is typically dominated by the largest component, which is often very close to 1. In order to extract the knowledge hidden in the small differences between the tiny components, they propose scaling the logits by a factor of  $1/T$ . Parameter  $T$  can be interpreted as a “temperature”, and the higher it is set the smoother the softmax output becomes, thus making the small differences in the probability components become more pronounced. The drawback of their approach is that  $T$ , as a newly introduced free hyperparameter, needs to be tuned.

Increasing the temperature is not the only way of enhancing the differences between tiny probability components. The same effect can be achieved by transforming the probabilities to a different domain. For example, Ba and Caruana (2014) use the logit domain, and in our derivative square error loss we use the log domain, both of which succeed in enhancing the small differences. Indeed, Hinton et al. (2015) argue that in the limit of setting the temperature infinitely high, their method approximates a transformation to the logit domain.

### 2.3.5 FitNets

FitNets is an enhancement to the techniques used for model compression, proposed in a recent work by Romero et al. (2015). The authors argue that depth is highly important for the representation power of neural networks and therefore it should not be sacrificed when compressing them. Even more so, they argue that the compressed networks should become even deeper, so that they can compensate with depth for the loss in parameters. This leads to deep and thin compressed networks (playfully termed FitNets), on which, according to the authors, knowledge distillation approaches such as that of Hinton et al. (2015) tend to underperform.

In order to make the training of FitNets more successful, they propose having the teacher network provide supervision not only to the final layer of the student network, but also to some of its intermediate layers. That is, in addition to matching the outputs, they also attempt to match the intermediate representations of the teacher and the student. Unlike the output layers which are the same, the intermediate layers of the teacher and the student are of different size and typically learn different representations. Therefore, for making their matching possible, they propose having a regressor that tries to predict the teacher’s intermediate representations from the student’s intermediate representations. Training the student to make the regressor predict the teacher’s intermediate representations well becomes then part of the training objective during compression.

This approach departs from using only the output of models, which has so far been the dominant approach in model compression, including our framework presented in this thesis. Therefore, it is interesting to see what new research directions it may give rise to in the near future. Its drawback however is that it is specific to neural networks and it is not obvious how it can be generalized to other models.

### 2.3.6 Matching derivatives

In this chapter, we argued that knowing the derivatives of the function represented by the teacher with respect to the input provides a significant amount of information for the student. However, in machine learning, training models to learn input-output mappings has traditionally been done

on labelled datasets, from which it is not obvious how to extract derivative information. Model compression is a rare example of a setup where the whole function to be learnt is readily available, and therefore its derivatives can be computed. To the best of our knowledge, our work is the first in the literature that proposed using derivatives in the context of model compression. In the following, we review three works that have used derivatives in related contexts.

### Tangent prop

The importance of knowing the derivatives of the function to be learnt was also highlighted by Simard et al. (1992). In their work, they made a keen observation that for certain transformations of input points, the network’s predictions should not change. For example, a slightly rotated image of the digit “3” should still be classified as a “3”. This implies that the directional derivative to the direction of the transformation should be zero.

To introduce this invariance to the training process, they propose adding an extra term to the training objective that encourages these directional derivatives to be zero. This extra term is similar to our derivative square error loss, except that it does not use the log domain and it contains only certain directional derivatives. This extra term can be differentiated with respect to the model parameters using a modified version of backprop, termed *tangent prop*, which is similar to  $R\{\text{backprop}\}$  as used by us (see section B.3). The main difference with our work is that we can evaluate the full derivative with respect to the input, and by matching this, we can essentially match the full tangent hyperplane and not just a few selected directional derivatives.

### Contractive auto-encoders

Encouraging derivatives to be zero for introducing invariance was also proposed by Rifai et al. (2011) in the context of auto-encoders. Auto-encoders (Bengio, 2009, section 5.2) are neural networks consisting of two cascaded phases: the encoding phase, which transforms the input to an intermediate representation, and the decoding phase, which reconstructs the input from the intermediate representation. Auto-encoders are typically trained by minimizing the discrepancy between the input and its reconstruction.

Rifai et al. (2011) argue that the intermediate representation of an auto-encoder should be invariant to small changes in the input. In other words, small perturbations of the input should yield similar intermediate representations. This is equivalent to saying that the derivatives of the intermediate representation with respect to the input should be small. Motivated by this, they introduce *contractive auto-encoders*, which are trained by adding an extra regularization term to their loss function, consisting of the sum of the squares of all partial derivatives of the intermediate representation with respect to the input. Viewed from our framework’s perspective, this is equivalent to matching all derivatives to zero by derivative square error.

### Score matching

The idea of matching derivatives was also used by Hyvärinen (2005), where it was termed *score matching*. There, the focus is on learning unnormalizable generative models. Evaluating the exact probability distribution of an unnormalizable generative model is intractable, making maximum likelihood learning hard. However, the derivative of the log probability is tractable, because it does not involve the intractable normalization constant. The main idea of score matching is to train the intractable model by matching its tractable derivatives with the derivatives of the distribution that we want it to learn.

Score matching is similar to our derivative square error loss, the difference being that in score matching the derivatives are with respect to the distribution’s random variable, whereas in derivative square error they are with respect to the input. Also, score matching is used in a different context to that of model compression. Nevertheless, score matching is used in our framework for distilling intractable models in chapter 4, and more details about it can be found in section 4.1.1.

## 2.4 Summary and conclusions

In this chapter we presented a framework for model compression. We view model compression as a special case of knowledge distillation, where the knowledge of a large model is distilled into a small model. Our framework is fairly general, and it can in principle be applied to any sort of model, as long as the student model is differentiable.

The two key elements of our framework are the data generator and the loss. We commented extensively on the vital importance of the data generator, an importance which we also demonstrated in our experiments. We showed that a flexible generative model like NADE can be successfully used as a data generator, one which is capable of providing as much train data as desired.

To the best of our knowledge, we are the first to have used derivative matching in the model compression literature. We argued that knowing the derivatives of the teacher model with respect to the input provides significant information for the student, and we showed that in some cases knowing only the derivatives is sufficient for training. In our experiments, we demonstrated that matching derivatives has a greater generalization potential than matching function values when the input data is limited.

The literature on model compression is quite rich, and we made a detailed review of it. The main ideas found in the literature are rather similar, and have been renamed over the years as “model compression”, “mimicking” and “knowledge distillation”. We hope our work has added new ideas to this literature, and a practical implementation of them.

## Chapter 3

# Compact Predictive Distributions

Bayesian inference is a particular way of doing machine learning, based on using nothing but the rules of probability (Barber, 2012; MacKay, 2002). In Bayesian inference, every quantity of interest, regardless of what it corresponds to, admits a probability distribution, which represents our uncertainty about its value. Then, both learning and making predictions are done by simply using the rules of probability throughout the process.

Bayesian inference is elegant, principled and conceptually simple, and it is based on the well-founded and well-understood theory of probability. Nevertheless, the rules of probability, despite their seemingly innocuous simplicity, lead to intractable computations in all but the most trivial cases. This computational difficulty of Bayesian inference is one of the main factors that have hindered its widespread adoption in real-life applications.

One way of tackling the computational difficulty of Bayesian inference is via Monte Carlo<sup>1</sup> (Bishop, 2006, chapter 11). The idea behind Monte Carlo is to use a bag of samples generated from a probability distribution as a proxy for it. Having access to such a bag of samples, any expectation over the original probability distribution can be approximated by the arithmetic average over the samples. Moreover, this approximation can be made arbitrarily good by sufficiently increasing the number of samples. Hence, expectations that might be intractable to calculate analytically, can be effectively approximated via Monte Carlo.

However, the quality of the Monte Carlo approximation improves very slowly with the number of samples. Assuming samples are independent, the standard deviation of a Monte Carlo estimate decreases as  $\mathcal{O}(1/\sqrt{S})$ , where  $S$  is the number of samples. If the samples are not independent, which is more often the case, the standard deviation of the estimate decreases even more slowly. For ballpark estimates this low precision might not be a problem, but for high accuracy computations a very large number of samples may be needed. Furthermore, if the distribution we wish to sample from is defined over e.g. the half-million parameter vector of a large deep neural network, then the Monte Carlo approximation can become prohibitively expensive in both computation time and storage space.

It is natural then to ask the question, having been given a large bag of samples, is it possible to distil the knowledge about the original distribution contained in the samples to a more compact form? That is, to a form that takes little time to evaluate and limited space to store? If the answer is yes, then we can throw away the original bulky bag of samples and use the compact model instead.

In this chapter we describe a set of techniques for distilling bags of samples to a compact model. Our framework is based on that of Snelson and Ghahramani (2005a), but viewed from the perspective of knowledge distillation. In addition, we propose novel extensions to the framework, allowing us to avoid storing the full set of samples and to take advantage of derivative information about the target prediction surface. We validate our framework in two Bayesian inference problems: Bayesian density estimation and Bayesian binary classification.

---

<sup>1</sup>Named after the eponymous casino in the tiny principality of Monaco.

### 3.1 Distilling bags of MCMC samples

A typical setup of a problem in Bayesian inference is as follows. Suppose we have observed some data  $D = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ , which we know were independently generated by some model  $p(\mathbf{x} | \mathbf{w})$ , parameterized by a set of unknown parameters  $\mathbf{w}$ . That is, we do not know what the value of  $\mathbf{w}$  is for the model that actually generated  $D$ . The question we want to answer is, given that we observed  $D$ , how likely do we believe  $\mathbf{x}$  to be? That is, what is our prediction for the next point to be generated by the model?

Essentially, what the question asks is to calculate the probability  $p(\mathbf{x} | D)$ , which is known as the *predictive distribution*. In order to do this, we need to have some idea of what the parameters  $\mathbf{w}$  were before we observed  $D$ . That is, we need to know—or assume— $p(\mathbf{w})$ , which is known as the *prior distribution*. Finally, for simplicity, let us assume that if we actually knew the true value of  $\mathbf{w}$ , then our observations  $D$  would not influence our predictions about  $\mathbf{x}$ . Based on the above assumptions, and using nothing but the rules of probability, it is easy to show that the predictive distribution is given by

$$p(\mathbf{x} | D) = \langle p(\mathbf{x} | \mathbf{w}) \rangle_{p(\mathbf{w} | D)}, \quad (3.1)$$

where

$$p(\mathbf{w} | D) \propto p(\mathbf{w}) \prod_n p(\mathbf{x}_n | \mathbf{w}). \quad (3.2)$$

In the above equations,  $p(\mathbf{w} | D)$  is known as the *posterior distribution* and  $p(D | \mathbf{w}) = \prod_n p(\mathbf{x}_n | \mathbf{w})$  as the *likelihood*. The notation  $\langle p(\mathbf{x} | \mathbf{w}) \rangle_{p(\mathbf{w} | D)}$  denotes expectation of  $p(\mathbf{x} | \mathbf{w})$  with respect to  $p(\mathbf{w} | D)$ .

In most practical cases, the computation described by the above equations is intractable, even for tractable prior distribution and likelihood. The posterior  $p(\mathbf{w} | D)$  is typically unnormalizable in practice, that is, it can only be evaluated up to a multiplicative constant. Moreover, the expectation over the posterior that defines the predictive distribution  $p(\mathbf{x} | D)$  is typically high-dimensional and intractable to compute, either analytically or by numerical integration.

One way of approximating the above computation is via Markov Chain Monte Carlo (or MCMC for short). MCMC works by setting up a Markov chain whose equilibrium distribution is the posterior  $p(\mathbf{w} | D)$ . Then, by simulating this chain, a set of samples  $\{\mathbf{w}_s\}$  from  $p(\mathbf{w} | D)$  can be obtained. The advantage of using MCMC over other sampling methods is that (a) MCMC only needs to know  $p(\mathbf{w} | D)$  up to a multiplicative constant, which as we already discussed is usually the case, and (b) MCMC scales better than other sampling methods to high-dimensional spaces, which the  $\mathbf{w}$  space often is. There are numerous different methods in the MCMC literature, which is extensively reviewed by Neal (1993) and Murray (2007).

Having sampled the posterior using MCMC and having obtained a set of parameter samples  $\{\mathbf{w}_s\}$ , the predictive distribution can be approximated by the following Monte Carlo estimate

$$p(\mathbf{x} | D) \approx p_{MC}(\mathbf{x}) = \frac{1}{S} \sum_s p(\mathbf{x} | \mathbf{w}_s). \quad (3.3)$$

The above however is an expensive and inefficient representation of the predictive distribution, especially if highly accurate estimates are needed and therefore a large number of samples has to be collected. Indeed, every time a prediction needs to be made for some particular value of  $\mathbf{x}$ , the full collection of samples needs to be scanned. Storing the samples can also be expensive, especially if  $\mathbf{w}$  is high-dimensional.

To alleviate this problem and make MCMC inference more efficient, we can distil the knowledge contained in  $\{\mathbf{w}_s\}$  into a compact representation of the predictive distribution. Let  $p(\mathbf{x} | \boldsymbol{\theta})$  be a family of distributions, parameterized by a set of parameters  $\boldsymbol{\theta}$ , that we have chosen to be compact but at the same time flexible enough to approximate the true predictive distribution. The goal of distillation is to find the particular value of  $\boldsymbol{\theta}$  for which  $p(\mathbf{x} | \boldsymbol{\theta})$  is as good an approximation to  $p_{MC}(\mathbf{x})$  as possible. That is, distillation trains  $p(\mathbf{x} | \boldsymbol{\theta})$  in order to make

$$p_{MC}(\mathbf{x}) \approx p(\mathbf{x} | \boldsymbol{\theta}) \quad (3.4)$$

become as close to an equality as possible.

In the rest of this chapter, we will describe a set of techniques for performing knowledge distillation of bags of MCMC samples in the context Bayesian inference. Following Snelson and Ghahramani (2005a), we consider two different problems, Bayesian density estimation and Bayesian binary classification.

## 3.2 Bayesian density estimation

In this section, we study the problem of Bayesian density estimation, where  $p(\mathbf{x} | \mathbf{w})$  is a density model of  $\mathbf{x}$ , such as a mixture of Gaussians. We wish to learn a compact model  $p(\mathbf{x} | \boldsymbol{\theta})$  to approximate the predictive distribution  $p(\mathbf{x} | D)$ , given a Markov chain that explores the posterior  $p(\mathbf{w} | D)$ .

We will start by defining a loss function that measures the discrepancy between  $p(\mathbf{x} | D)$  and  $p(\mathbf{x} | \boldsymbol{\theta})$ . Learning the compact model amounts to minimizing this loss function with respect to  $\boldsymbol{\theta}$ . A natural, information-theoretic measure of discrepancy between distributions is the KL divergence from  $p(\mathbf{x} | D)$  to  $p(\mathbf{x} | \boldsymbol{\theta})$ , defined as follows

$$D_{\text{KL}}(p(\mathbf{x} | D) \| p(\mathbf{x} | \boldsymbol{\theta})) = \langle \log p(\mathbf{x} | D) \rangle_{p(\mathbf{x} | D)} - \langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | D)}. \quad (3.5)$$

The term  $\langle \log p(\mathbf{x} | D) \rangle_{p(\mathbf{x} | D)}$  is the negative entropy of  $p(\mathbf{x} | D)$ , and it is a constant with respect to  $\boldsymbol{\theta}$ . Hence, minimizing the above KL divergence is equivalent to maximizing  $\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | D)}$ . This can be interpreted as fitting  $p(\mathbf{x} | \boldsymbol{\theta})$  using maximum likelihood to an infinite amount of data generated by  $p(\mathbf{x} | D)$ .

Our basic assumption is that all our knowledge about  $p(\mathbf{x} | D)$  comes from a Markov chain that generates MCMC samples from it. In the following two sections, we describe two ways of maximizing  $\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | D)}$  using this Markov chain: a “batch” way, first proposed by Snelson and Ghahramani (2005a), and a new “online” way proposed herein.

### 3.2.1 Batch distillation

By simulating the Markov chain, we can generate a set of MCMC samples  $\{\mathbf{w}_s\}$  from the posterior  $p(\mathbf{w} | D)$ . The Monte Carlo estimate

$$p_{\text{MC}}(\mathbf{x}) = \frac{1}{S} \sum_s p(\mathbf{x} | \mathbf{w}_s) \quad (3.6)$$

can then be used as an approximation to the true predictive distribution  $p(\mathbf{x} | D)$ . Hence, maximizing  $\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | D)}$  can be approximately done by (a) generating a set of samples  $\{\mathbf{x}_m\}$  from  $p_{\text{MC}}(\mathbf{x})$  and (b) fitting  $p(\mathbf{x} | \boldsymbol{\theta})$  to  $\{\mathbf{x}_m\}$  using maximum likelihood. This procedure is outlined in the algorithm below.

- (i) Generate a bag of samples  $\{\mathbf{w}_s\}$  of size  $S$  from  $p(\mathbf{w} | D)$  by simulating the Markov chain.
- (ii) Generate a bag of samples  $\{\mathbf{x}_m\}$  of size  $M$  from  $p_{\text{MC}}(\mathbf{x})$ .
- (iii) Train  $p(\mathbf{x} | \boldsymbol{\theta})$  by maximizing its average log likelihood on  $\{\mathbf{x}_m\}$ .

### 3.2.2 Online distillation

The batch algorithm works well in practice (as we shall see later in the experiments), but requires the full sets of samples  $\{\mathbf{w}_s\}$  and  $\{\mathbf{x}_m\}$  to be generated and stored before  $\boldsymbol{\theta}$  is optimized. This is problematic when a lot of samples are required and  $\mathbf{w}$  or  $\mathbf{x}$  (or both) are high-dimensional.

We will now describe an alternative online method for learning  $p(\mathbf{x} | \boldsymbol{\theta})$  that iteratively updates  $\boldsymbol{\theta}$  on the fly as the Markov chain is simulated. Firstly, notice that

$$\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | D)} = \langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x}, \mathbf{w} | D)}, \quad (3.7)$$

where  $p(\mathbf{x}, \mathbf{w} | D) = p(\mathbf{x} | \mathbf{w})p(\mathbf{w} | D)$ . Hence, a pair of samples  $(\mathbf{x}_s, \mathbf{w}_s)$  can be generated by first simulating the Markov chain for a single step to obtain  $\mathbf{w}_s$  and then sampling  $\mathbf{x}_s$  from  $p(\mathbf{x} | \mathbf{w}_s)$ . Having generated a minibatch  $\{(\mathbf{x}_s, \mathbf{w}_s)\}$  of size  $S$ , a stochastic update to  $\boldsymbol{\theta}$  can be made. Hence, optimizing  $\boldsymbol{\theta}$  can be done by iterating the above procedure, as outlined in the algorithm below.

- (i) Generate a minibatch of pairs of samples  $\{(\mathbf{x}_s, \mathbf{w}_s)\}$  of size  $S$  from  $p(\mathbf{w} | D)$  by simulating the Markov chain.
  - (ii) Make a stochastic update to  $\boldsymbol{\theta}$  using only the minibatch.
  - (iii) Repeat until  $\boldsymbol{\theta}$  converges.

In the above algorithm, the size of the minibatch  $S$  can be made as small as desired. The big advantage of this algorithm is that its storage requirements are  $\mathcal{O}(S)$ , since every minibatch can be thrown away after having been used. Compare this to the batch algorithm, where all the samples have to be stored in advance.

The downside of the online algorithm is that it requires care in arranging the stochastic updates, so that the algorithm remains stable. This is especially important when the minibatch is chosen to be small, since in this case the updates to  $\boldsymbol{\theta}$  have higher variance. A stochastic update can be, for instance, a single gradient update, such as

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{S} \sum_s \frac{\partial}{\partial \boldsymbol{\theta}} \log p(\mathbf{x}_s | \boldsymbol{\theta}), \quad (3.8)$$

where  $\alpha > 0$  is an appropriately chosen, possibly adaptive, learning rate. Alternatively,  $p(\mathbf{x} | \boldsymbol{\theta})$  can be fitted on the minibatch, and the  $\boldsymbol{\theta}$  can be interpolated between its new and its old value. That is

$$\boldsymbol{\theta} \leftarrow (1 - \alpha) \boldsymbol{\theta} + \alpha \boldsymbol{\theta}', \quad (3.9)$$

where

$$\boldsymbol{\theta}' = \arg \max_{\boldsymbol{\theta}} \frac{1}{S} \sum_s \log p(\mathbf{x}_s | \boldsymbol{\theta}) \quad (3.10)$$

and  $0 < \alpha < 1$  an appropriately chosen, possibly adaptive, learning rate. In our case study in section 3.2.3 we will provide more details on our particular choice of update strategy.

### 3.2.3 Case study: Bayesian Mixture of Gaussians

We will now showcase our framework for Bayesian density estimation on a toy inference task involving Mixtures of Gaussians (or MoG for short). The goal of the task is, given a dataset of observations that have been generated by a MoG with unknown means, to infer the predictive density.

#### The setup

The true model that generated the observations is a one-dimensional MoG with 3 components, given by

$$p(x | \mathbf{w}) = \frac{1}{3} \mathcal{N}(x | m_1, s_1^2) + \frac{1}{3} \mathcal{N}(x | m_2, s_2^2) + \frac{1}{3} \mathcal{N}(x | m_3, s_3^2), \quad (3.11)$$

where

$$\begin{aligned} m_1 &= -3 & m_2 &= 0 & m_3 &= 2 \\ s_1^2 &= 2 & s_2^2 &= 5 & s_3^2 &= 1 \end{aligned} . \quad (3.12)$$

We assume that the variances  $s_1^2, s_2^2, s_3^2$  are known, but the means are not. We therefore identify the unknown model parameters to be  $\mathbf{w} = \{m_1, m_2, m_3\}$ , over which we place the following broad Gaussian prior

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, 100\mathbf{I}). \quad (3.13)$$

In our experiments, we generated a dataset of observations  $D = \{x_1, x_2, \dots, x_N\}$  from the true model. We varied  $N$  across experiments. Pretending not to know the true means of the MoG, the task is to calculate  $p(x | D)$ . Note that even in this toy setting, the true posterior  $p(\mathbf{w} | D)$  is a sum of  $3^N$  Gaussians. Therefore, exact inference becomes hard even for a small set of observations, and an approximation like MCMC is typically needed.

### Distillation details

To generate samples from the posterior  $p(\mathbf{w} | D)$ , we used slice sampling (Neal, 2000). Slice sampling is a general MCMC method, whose main advantage is that it requires minimal tuning. We used linear stepping out, and we performed univariate updates to each parameter in turn. The chain was initialized at  $\mathbf{0}$ , and burned in for 1000 iterations. No thinning was used.

The compact model into which we chose to distil the Markov chain is a MoG with 3 components, given by

$$p(x | \boldsymbol{\theta}) = \pi_1 \mathcal{N}(x | \mu_1, \sigma_1^2) + \pi_2 \mathcal{N}(x | \mu_2, \sigma_2^2) + \pi_3 \mathcal{N}(x | \mu_3, \sigma_3^2), \quad (3.14)$$

whose free parameters are  $\boldsymbol{\theta} = \{\pi_1, \pi_2, \pi_3, \mu_1, \mu_2, \mu_3, \sigma_1^2, \sigma_2^2, \sigma_3^2\}$ . For batch distillation, we first generated 10,000 MCMC samples  $\{\mathbf{w}_s\}$  in order to form the approximate posterior  $p_{MC}(x)$ . Note that in this case  $p_{MC}(x)$  is a MoG with 30,000 components, so we can exactly sample from it. From  $p_{MC}(x)$ , we generated 1000 samples  $\{x_m\}$ . We then used the EM algorithm (Dempster et al., 1977) to fit  $p(x | \boldsymbol{\theta})$  on  $\{x_m\}$  using maximum likelihood. We ran EM until convergence of the likelihood value. More details on the EM algorithm are given in the appendix, section D.1.

For online distillation, we used minibatches of size 100. That is, in each iteration, 100 MCMC samples  $\{\mathbf{w}_s\}$  and 100 corresponding data samples  $\{x_s\}$  were generated. In each iteration,  $\boldsymbol{\theta}$  was updated using an online version of the EM algorithm (Cappé and Moulines, 2009). This version of EM makes a soft update on  $\boldsymbol{\theta}$  in each iteration, by interpolating between the sufficient statistics calculated from the minibatch  $\{x_s\}$  and the sufficient statistics calculated so far. More details on how this particular version of online EM works are given in the appendix, section D.2. Note that online EM needs a learning rate to be specified for each iteration. In our experiments, we used a learning rate that was initialized to 1, and then was decayed linearly until it reached 0 in the last iteration. We performed 100 iterations in total.

### Results and discussion

We ran the experiment described above for various dataset sizes  $N$ . Figures 3.1–3.9 show the results for  $N \in \{2, 5, 10, 20, 50, 100, 200, 500, 1000\}$ . On the left, the histograms of  $m_1, m_2$  and  $m_3$  are shown, calculated from all 10,000 MCMC samples obtained during batch distillation, with vertical lines indicating the true parameter values. These histograms are a good approximation of the true marginal posteriors  $p(m_1 | D), p(m_2 | D)$  and  $p(m_3 | D)$  respectively. On the right, we show the pdf of the true model  $p(x | \mathbf{w})$ , from which the dataset was generated and whose means we pretend not to know, superpositioned onto the three approximations of the predictive distribution  $p(x | D)$ : the Monte Carlo approximation  $p_{MC}(x)$  given by averaging over all 10,000 MCMC samples collected during batch distillation, the compact model  $p(x | \boldsymbol{\theta})$  trained using batch distillation, and the compact model  $p(x | \boldsymbol{\theta})$  trained using online distillation.

We can see that the larger a dataset we use, the closer the predictive distribution  $p(x | D)$  is to the true model  $p(x | \mathbf{w})$ . At the same time, the posterior  $p(\mathbf{w} | D)$  becomes more and more concentrated on the true parameter values. With only few datapoints, the posterior is broad, and

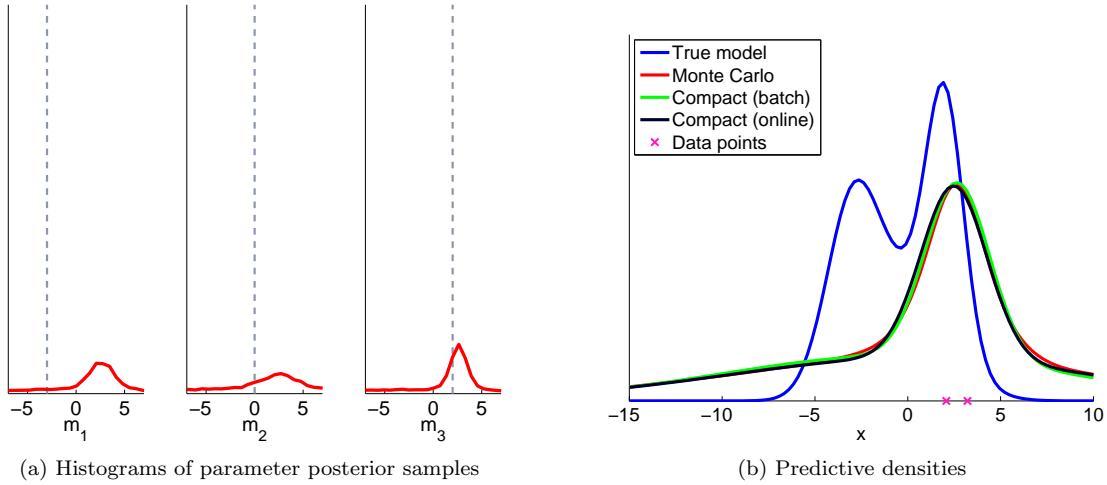


Figure 3.1: Bayesian MoG with 2 datapoints.

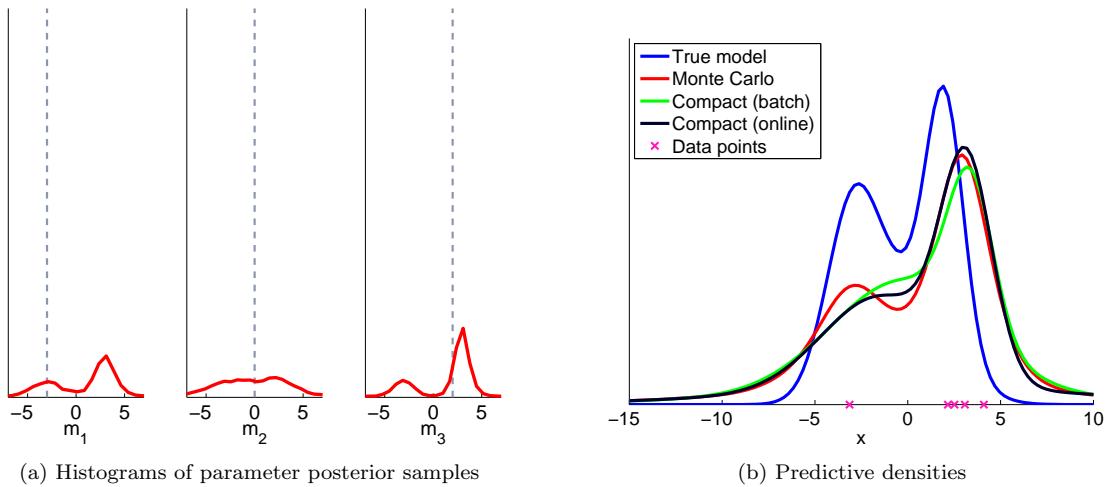


Figure 3.2: Bayesian MoG with 5 datapoints.

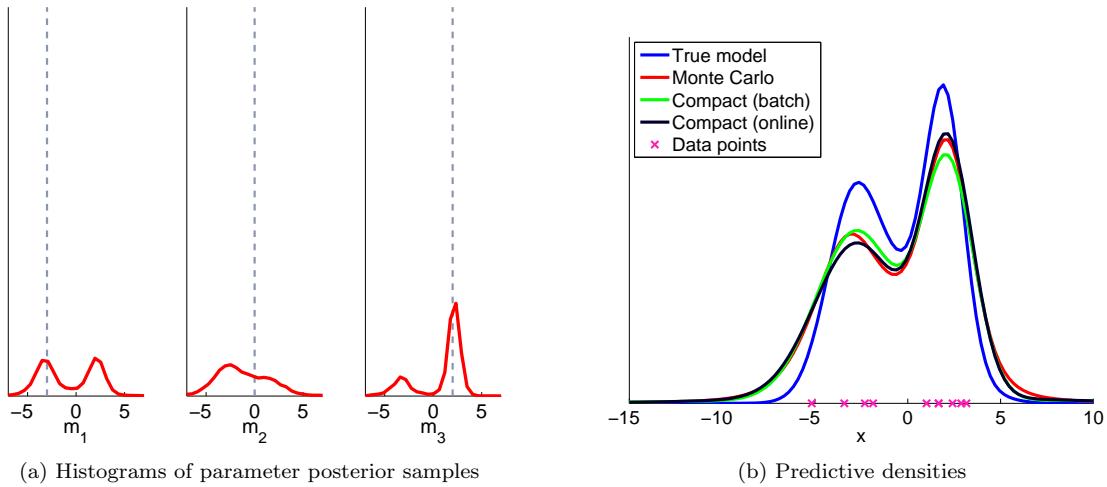


Figure 3.3: Bayesian MoG with 10 datapoints.

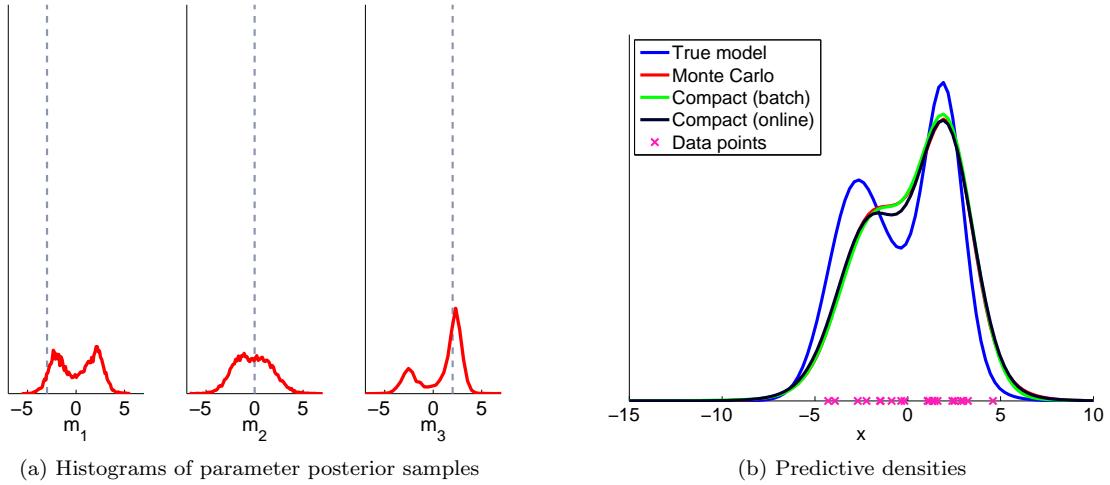


Figure 3.4: Bayesian MoG with 20 datapoints.

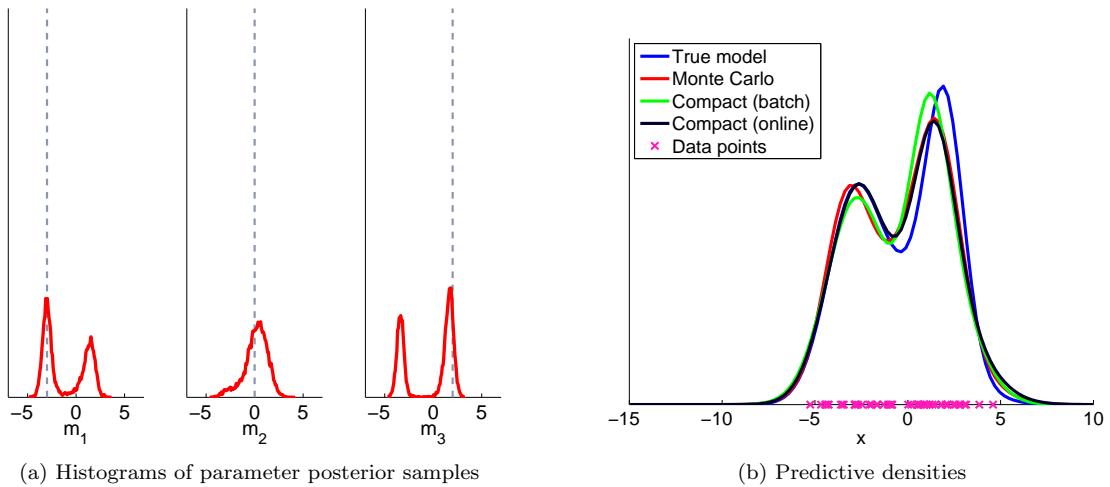


Figure 3.5: Bayesian MoG with 50 datapoints.

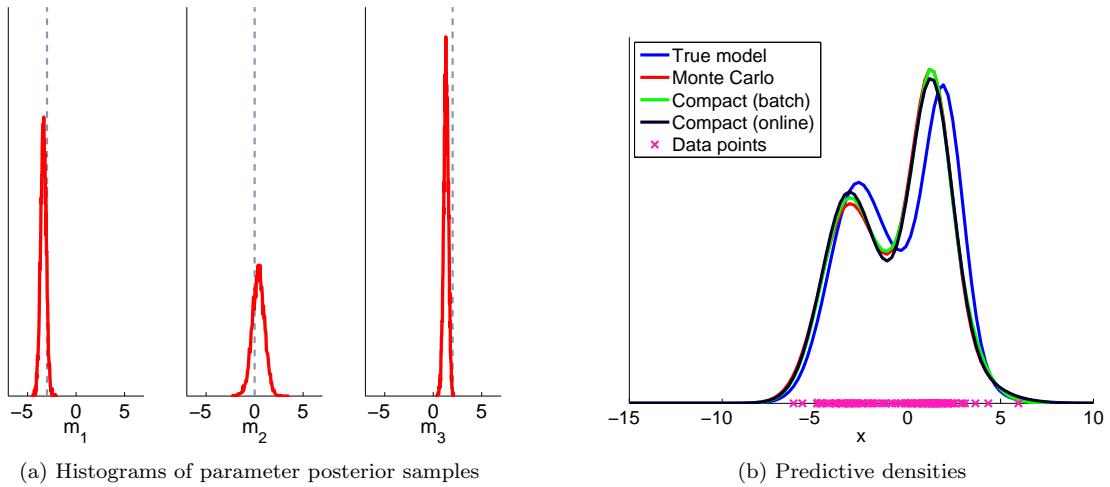


Figure 3.6: Bayesian MoG with 100 datapoints.

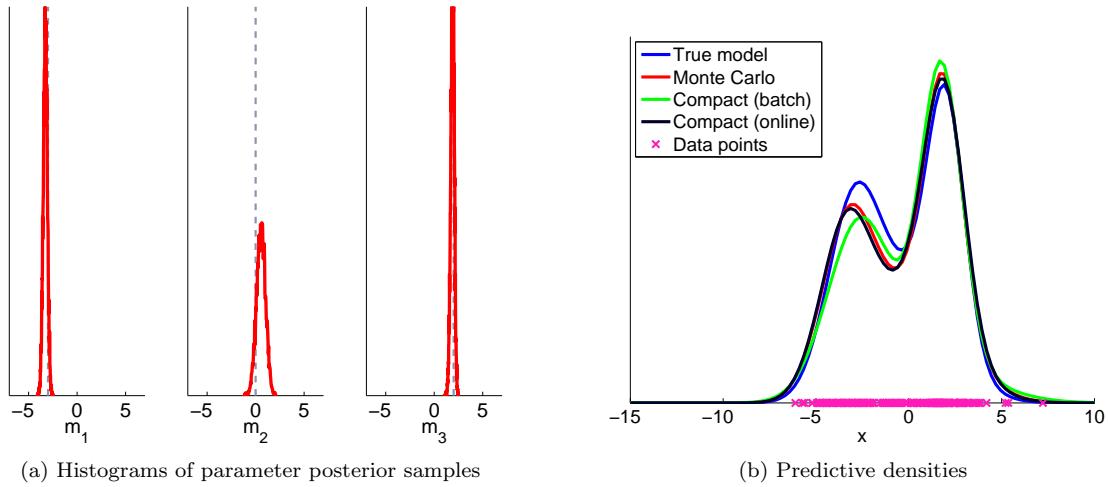


Figure 3.7: Bayesian MoG with 200 datapoints.

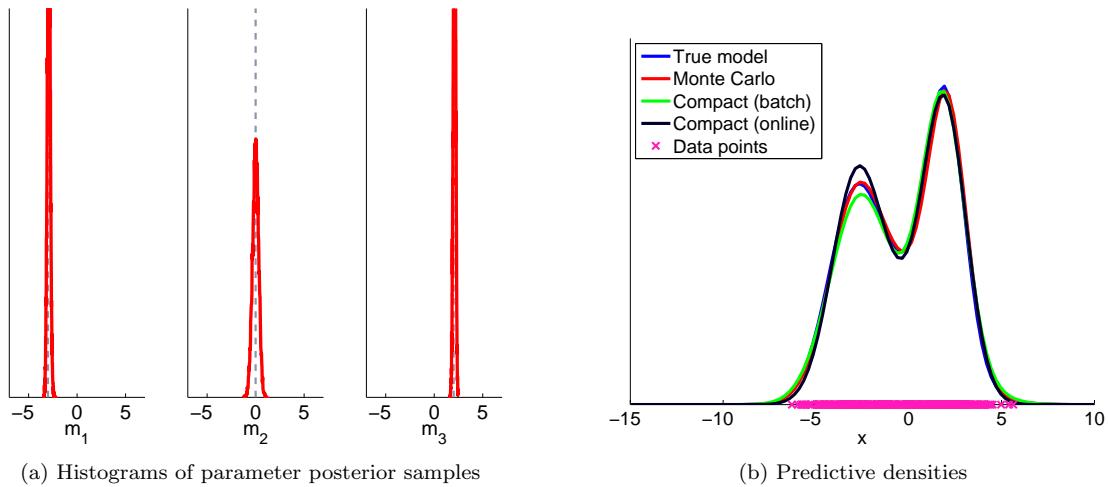


Figure 3.8: Bayesian MoG with 500 datapoints.

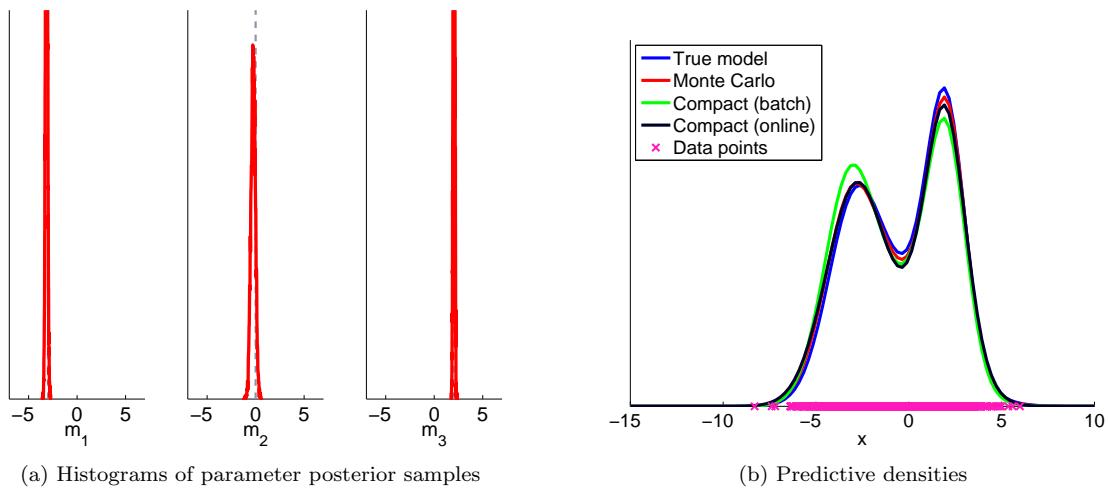


Figure 3.9: Bayesian MoG with 1000 datapoints.

as a consequence the predictive distribution is broader than the true model. This is because with only a few datapoints, the predictive distribution does not have enough evidence to rule out the possibility of future datapoints being far away from those that it has already seen. This is the big advantage of Bayesian modelling: the model is aware of its own uncertainty.

The plots show that both compact predictive distributions are close to the Monte Carlo estimate  $p_{MC}(x)$ . Note that the latter was given by averaging over 10,000 MCMC samples, therefore we assume that it is a good approximation to the true predictive distribution  $p(x | D)$ . However, even though  $p_{MC}(x)$  contains 30,000 parameters in total (3 means per sample), its functional form is fairly simple. This is why it can be faithfully represented by the compact model  $p(x | \theta)$ , which has only 9 parameters.

We can see that online distillation has produced compact predictive distributions that are close to those produced by batch distillation. This shows that the distillation process can be done successfully with little storage requirements. In fact, batch distillation needs to store 10,000 MCMC samples and 1000 data samples, whereas online distillation only needs to store 100 MCMC samples and 100 data samples at any time. This can be particularly useful when the dimensionality of either the weights or the datapoints is large, in which case it would be prohibitively expensive to store a lot of samples. In terms of computation cost, we found that batch and online distillation performed comparably.

For this experiment we used minibatches of size 100, but online distillation can work with any minibatch size, in order to adapt to any storage limitations. Empirically, we found that relatively large minibatches work better, since small minibatches can suffer from high variance. When using small minibatches, we found that thinning the Markov chain is beneficial, since this reduces the correlation of weight samples within the same minibatch. Another approach for reducing correlation within minibatches would be to have independent Markov chains running in parallel, each contributing with a single weight sample in the minibatch. In any case, the learning rate strategy is highly important for good performance. The need to specify and tune a learning rate strategy is the main downside of online distillation compared to batch distillation.

### 3.3 Bayesian binary classification

In this section, we will extend the framework described for density estimation to handle conditional distributions as well. This will allow us to apply the distillation framework in tasks such as regression and classification.

We assume that the data takes the form of input-output pairs  $(\mathbf{x}, \mathbf{y})$ , where  $\mathbf{x}$  is the input and  $\mathbf{y}$  is the output. We assume that the input is generated by some known distribution  $p(\mathbf{x})$ , and that the corresponding output is generated by a conditional distribution  $p(\mathbf{y} | \mathbf{x}, \mathbf{w})$ , parameterized by an unknown set of parameters  $\mathbf{w}$ . We further assume that we have a prior idea  $p(\mathbf{w})$  of what the parameters are and that we observe a dataset  $D = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_N, \mathbf{y}_N)\}$  of independently generated pairs. The task is to calculate the conditional predictive distribution  $p(\mathbf{y} | \mathbf{x}, D)$ .

Similarly to the density estimation case, by applying the rules of probability, we obtain the predictive distribution as follows

$$p(\mathbf{y} | \mathbf{x}, D) = \langle p(\mathbf{y} | \mathbf{x}, \mathbf{w}) \rangle_{p(\mathbf{w} | D)}, \quad (3.15)$$

where

$$p(\mathbf{w} | D) \propto p(\mathbf{w}) \prod_n p(\mathbf{y}_n | \mathbf{x}_n, \mathbf{w}). \quad (3.16)$$

By sampling weights  $\{\mathbf{w}_s\}$  from the posterior  $p(\mathbf{w} | D)$  using MCMC, we can construct a Monte Carlo estimate of the predictive distribution as follows

$$p(\mathbf{y} | \mathbf{x}, D) \approx p_{MC}(\mathbf{y} | \mathbf{x}) = \frac{1}{S} \sum_s p(\mathbf{y} | \mathbf{x}, \mathbf{w}_s). \quad (3.17)$$

Our goal is to distil a compact model  $p(\mathbf{y} | \mathbf{x}, \boldsymbol{\theta})$  from the MCMC samples, that is as close to the true predictive distribution as possible.

For simplicity, in the rest of this section we will focus on binary classification. In other words, we will assume that output  $\mathbf{y}$  is a binary variable that can be either 0 or 1 (and we will denote it as  $y$ ). This will allow expectations over  $y$  to be calculated analytically. However, in principle there is no need to restrict ourselves to binary classification, and the framework can be easily extended to handle more general types of output  $\mathbf{y}$ .

### 3.3.1 Loss functions

The first thing we need to do is define an appropriate loss function to measure the discrepancy between the true predictive distribution  $p(y | \mathbf{x}, D)$  and the compact predictive distribution  $p(y | \mathbf{x}, \boldsymbol{\theta})$ . Minimizing this loss function with respect to  $\boldsymbol{\theta}$  will train  $p(y | \mathbf{x}, \boldsymbol{\theta})$  to be as close an approximation to  $p(y | \mathbf{x}, D)$  as possible. In the following, we describe two loss functions that achieve this goal: *cross entropy* and *derivative square error*.

#### Cross entropy

In the density estimation case, we used KL divergence as a loss function. To use KL divergence as a loss function for binary classification too, we need to take care of the input variable  $\mathbf{x}$ . Following Snelson and Ghahramani (2005a), we average the KL divergence over all possible values of  $\mathbf{x}$ , which gives the following loss function

$$E_{\text{KL}}(\boldsymbol{\theta}) = \langle D_{\text{KL}}(p(y | \mathbf{x}, D) \| p(y | \mathbf{x}, \boldsymbol{\theta})) \rangle_{p(\mathbf{x})}. \quad (3.18)$$

We can rewrite the above as follows

$$E_{\text{KL}}(\boldsymbol{\theta}) = \langle \log p(y | \mathbf{x}, D) \rangle_{p(y | \mathbf{x}, D)p(\mathbf{x})} - \langle \log p(y | \mathbf{x}, \boldsymbol{\theta}) \rangle_{p(y | \mathbf{x}, D)p(\mathbf{x})}. \quad (3.19)$$

The first term in the above is a constant with respect to  $\boldsymbol{\theta}$  so we can simply ignore it. The second term is the cross entropy between  $p(y | \mathbf{x}, D)$  and  $p(y | \mathbf{x}, \boldsymbol{\theta})$  averaged over  $p(\mathbf{x})$ , which we shall denote as  $E_{\text{CE}}(\boldsymbol{\theta})$ . Minimizing  $E_{\text{CE}}(\boldsymbol{\theta})$  is then equivalent to minimizing  $E_{\text{KL}}(\boldsymbol{\theta})$ .

Since  $y$  is binary, we can do the expectations with respect to it analytically. For convenience, we define the following two functions

$$t(\mathbf{x}) = p(y = 1 | \mathbf{x}, D) \quad (3.20)$$

$$f(\mathbf{x} | \boldsymbol{\theta}) = p(y = 1 | \mathbf{x}, \boldsymbol{\theta}). \quad (3.21)$$

Using these, we can rewrite cross entropy as

$$E_{\text{CE}}(\boldsymbol{\theta}) = - \left\langle t(\mathbf{x}) \log f(\mathbf{x} | \boldsymbol{\theta}) + (1 - t(\mathbf{x})) \log(1 - f(\mathbf{x} | \boldsymbol{\theta})) \right\rangle_{p(\mathbf{x})}. \quad (3.22)$$

It is easy to see that cross entropy is a valid loss function for knowledge distillation. It is well known that the KL divergence is always non-negative, and that it becomes zero if and only if  $p(y | \mathbf{x}, D) = p(y | \mathbf{x}, \boldsymbol{\theta})$  (MacKay, 2002, section 2.6). Hence,  $E_{\text{KL}}(\boldsymbol{\theta})$ —and equivalently  $E_{\text{CE}}(\boldsymbol{\theta})$ —is minimized if and only if  $t(\mathbf{x}) = f(\mathbf{x} | \boldsymbol{\theta})$  everywhere in the support of  $p(\mathbf{x})$ .

#### Derivative square error

Functions  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$ , as defined above, correspond to the probability of  $y$  being turned on at input location  $\mathbf{x}$ , as given by the true predictive distribution and the compact predictive distribution respectively. Geometrically,  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$  are surfaces over the input space. The process of distillation can be viewed as trying to make the surface represented by  $f(\mathbf{x} | \boldsymbol{\theta})$  match as closely as possible the surface represented by  $t(\mathbf{x})$ , especially at input locations  $\mathbf{x}$  for which  $p(\mathbf{x})$  is high.

Minimizing the cross entropy  $E_{\text{CE}}(\boldsymbol{\theta})$  is one way of making the surfaces match, by matching function values  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$  directly. However, assuming the surfaces are smooth, we can take

this one step further and try to match function derivatives with respect to the input  $\mathbf{x}$  as well. By matching derivatives as well as function values, we encourage the two surfaces to not only have the same height, but also have the same slope. Since matching derivatives at input location  $\mathbf{x}$  is equivalent to matching a whole tangent hyperplane to the surface at  $\mathbf{x}$ , a loss function based on derivatives can constrain the surfaces with fewer evaluations. This can become a great advantage especially in high-dimensional input spaces.

In fact, for the task of binary classification, we can match the two surfaces only by using derivatives, without the need to use function values at all. We propose herein a loss function that involves derivatives with respect to  $\mathbf{x}$  and achieves exactly that. This loss function, which we call *derivative square error*, is defined as follows

$$E_{\text{DSE}}(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \left\| \frac{\partial}{\partial \mathbf{x}} \log p(y | \mathbf{x}, \boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log p(y | \mathbf{x}, D) \right\|^2 \right\rangle_{p(y | \mathbf{x}, D)p(\mathbf{x})}. \quad (3.23)$$

Notice that we first transform the probabilities into the log domain, and then take their derivatives. Expanding the expectation over  $y$ , and writing the above in terms of  $t(\mathbf{x})$  and  $f(\mathbf{x} | \boldsymbol{\theta})$ , we get the following

$$E_{\text{DSE}}(\boldsymbol{\theta}) = \langle t(\mathbf{x}) E_1(\mathbf{x}, \boldsymbol{\theta}) + (1 - t(\mathbf{x})) E_0(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}, \quad (3.24)$$

where

$$E_1(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2} \left\| \frac{\partial}{\partial \mathbf{x}} \log f(\mathbf{x} | \boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log t(\mathbf{x}) \right\|^2 \quad (3.25)$$

$$E_0(\mathbf{x}, \boldsymbol{\theta}) = \frac{1}{2} \left\| \frac{\partial}{\partial \mathbf{x}} \log (1 - f(\mathbf{x} | \boldsymbol{\theta})) - \frac{\partial}{\partial \mathbf{x}} \log (1 - t(\mathbf{x})) \right\|^2. \quad (3.26)$$

The validity of derivative square error as a loss function is established by the following proposition, which states that, unless  $t(\mathbf{x})$  is the constant function, minimizing  $E_{\text{DSE}}(\boldsymbol{\theta})$  is sufficient for making the surfaces match.

**Proposition 3.1.** *Assuming that  $f(\mathbf{x} | \boldsymbol{\theta})$  has enough capacity and that  $t(\mathbf{x})$  is not constant in the support of  $p(\mathbf{x})$ ,  $E_{\text{DSE}}(\boldsymbol{\theta})$  is minimized if and only if  $f(\mathbf{x} | \boldsymbol{\theta}) = t(\mathbf{x})$  for all  $\mathbf{x}$  in the support of  $p(\mathbf{x})$ .*

*Proof.* Let  $S_p$  be the support of  $p(\mathbf{x})$ , i.e. the set of points  $\mathbf{x}$  for which  $p(\mathbf{x}) > 0$ . Notice that  $E_{\text{DSE}}(\boldsymbol{\theta}) \geq 0$ , as the expectation of a norm. Obviously, if  $f(\mathbf{x} | \boldsymbol{\theta}) = t(\mathbf{x})$  for all  $\mathbf{x} \in S_p$ , we have that  $E_{\text{DSE}}(\boldsymbol{\theta}) = 0$ . Conversely, if  $E_{\text{DSE}}(\boldsymbol{\theta}) = 0$ , it has to be the case that for all  $\mathbf{x} \in S_p$  the following hold

$$\frac{\partial}{\partial \mathbf{x}} \log f(\mathbf{x} | \boldsymbol{\theta}) = \frac{\partial}{\partial \mathbf{x}} \log t(\mathbf{x}) \quad (3.27)$$

$$\frac{\partial}{\partial \mathbf{x}} \log (1 - f(\mathbf{x} | \boldsymbol{\theta})) = \frac{\partial}{\partial \mathbf{x}} \log (1 - t(\mathbf{x})). \quad (3.28)$$

The above two constraints imply the existence of positive constants  $c_1$  and  $c_2$  such that

$$f(\mathbf{x} | \boldsymbol{\theta}) = c_1 t(\mathbf{x}) \quad (3.29)$$

$$1 - f(\mathbf{x} | \boldsymbol{\theta}) = c_2 (1 - t(\mathbf{x})), \quad (3.30)$$

which combine to give

$$(c_2 - c_1) t(\mathbf{x}) = c_2 - 1. \quad (3.31)$$

The above implies that either  $t(\mathbf{x})$  is constant in  $S_p$  or  $c_1 = c_2 = 1$ . Since we have assumed that  $t(\mathbf{x})$  is not constant in  $S_p$ , it follows that  $f(\mathbf{x} | \boldsymbol{\theta}) = t(\mathbf{x})$  everywhere in  $S_p$ .  $\square$

### 3.3.2 Batch distillation

In order to be in a position to minimize either  $E_{\text{CE}}(\boldsymbol{\theta})$  or  $E_{\text{DSE}}(\boldsymbol{\theta})$ , we need to have access to the true predictive distribution  $t(\mathbf{x})$ . In our framework, we have approximate access to  $t(\mathbf{x})$  via MCMC sampling. By collecting a large number of MCMC samples  $\{\mathbf{w}_s\}$ , we can approximate  $t(\mathbf{x})$  using the following Monte Carlo estimate

$$t(\mathbf{x}) \approx p_{\text{MC}}(y = 1 | \mathbf{x}) = \frac{1}{S} \sum_s p(y = 1 | \mathbf{x}, \mathbf{w}_s). \quad (3.32)$$

Provided  $S$  is large enough, the above estimate can be used as a proxy for  $t(\mathbf{x})$ .

We will now describe a stochastic gradient optimization method for minimizing  $E_{\text{CE}}(\boldsymbol{\theta})$  and  $E_{\text{DSE}}(\boldsymbol{\theta})$ . First, note that both loss functions can be written in the form

$$E(\boldsymbol{\theta}) = \langle E(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} \quad (3.33)$$

for an appropriate instantiation of  $E(\mathbf{x}, \boldsymbol{\theta})$ . The derivative of the above with respect to  $\boldsymbol{\theta}$  can be written as

$$\mathbf{g}(\boldsymbol{\theta}) = \left\langle \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta}) \right\rangle_{p(\mathbf{x})}. \quad (3.34)$$

If we have a minibatch  $\{\mathbf{x}_m\}$  of samples from  $p(\mathbf{x})$ , then we can stochastically approximate the above derivative by

$$\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{M} \sum_m \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_m, \boldsymbol{\theta}). \quad (3.35)$$

It is easy to see that  $\hat{\mathbf{g}}(\boldsymbol{\theta})$  is an unbiased estimate of  $\mathbf{g}(\boldsymbol{\theta})$ . By repeatedly sampling minibatches from  $p(\mathbf{x})$ , calculating  $\hat{\mathbf{g}}(\boldsymbol{\theta})$  and then using it to update  $\boldsymbol{\theta}$ , we can effectively minimize  $E(\boldsymbol{\theta})$ . This is outlined in the procedure below.

- (i) Generate a bag of samples  $\{\mathbf{w}_s\}$  of size  $S$  from  $p(\mathbf{w} | D)$  by simulating the Markov chain. Use  $p_{\text{MC}}(y = 1 | \mathbf{x})$  as an approximation to  $t(\mathbf{x})$ .
  - (ii) Generate a minibatch  $\{\mathbf{x}_m\}$  of size  $M$  from  $p(\mathbf{x})$ .
  - (iii) Calculate the stochastic gradient  $\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{M} \sum_m \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_m, \boldsymbol{\theta})$ .
  - (iv) Make an update on  $\boldsymbol{\theta}$  using  $\hat{\mathbf{g}}(\boldsymbol{\theta})$ .
  - (v) If  $\boldsymbol{\theta}$  has not converged yet, go back to step (ii).

The above algorithm relies on three conditions. Firstly, the number of MCMC samples  $S$  has to be large enough so that  $p_{\text{MC}}(y = 1 | \mathbf{x})$  can be effectively used in place of  $t(\mathbf{x})$ . Secondly, we need to be able to calculate the derivative  $\frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta})$ . In our case study in section 3.3.4, we discuss how to calculate  $\frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta})$  in the particular setting of logistic regression. Thirdly, the update strategy of  $\boldsymbol{\theta}$  using  $\hat{\mathbf{g}}(\boldsymbol{\theta})$  has to be suitably chosen, so that the algorithm converges. Bottou (1998) provides sufficient conditions for stochastic gradient algorithms of the above type that guarantee almost sure convergence to a stationary point of  $E(\boldsymbol{\theta})$  (provided samples from  $p(\mathbf{x})$  are drawn independently).

### 3.3.3 Online distillation

The downside of batch distillation as previously considered by Snelson and Ghahramani (2005a) is that a large number of MCMC samples  $\{\mathbf{w}_s\}$  has to be generated and stored in advance. This can be problematic, especially if  $\mathbf{w}$  is high-dimensional. To alleviate this problem, in this section

we present an online version of the distillation procedure. In online distillation, the updates to  $\boldsymbol{\theta}$  take place on the fly as the Markov chain is simulated, and hence there is no need to store the previously generated MCMC samples.

Online distillation relies on using a single MCMC sample as a stochastic estimate of  $t(\mathbf{x})$ . To see how this is done, define the following

$$t(\mathbf{x}, \mathbf{w}) = p(y = 1 | \mathbf{x}, \mathbf{w}). \quad (3.36)$$

It is easy to see that the above is an unbiased estimate of  $t(\mathbf{x})$  since

$$t(\mathbf{x}) = \langle t(\mathbf{x}, \mathbf{w}) \rangle_{p(\mathbf{w} | D)}. \quad (3.37)$$

The idea of online distillation is to use  $t(\mathbf{x}, \mathbf{w})$  instead of  $t(\mathbf{x})$  in the loss function and then average the whole loss function with respect to  $p(\mathbf{w} | D)$ . This way, the loss function takes the following form

$$E(\boldsymbol{\theta}) = \langle E(\mathbf{x}, \mathbf{w}, \boldsymbol{\theta}) \rangle_{p(\mathbf{w} | D)p(\mathbf{x})}. \quad (3.38)$$

Having the loss function in the above form, we can now minimize it using stochastic gradients not only with respect to  $\mathbf{x}$ , but also with respect to  $\mathbf{w}$ . That is, by simulating the Markov chain  $M$  times to get a minibatch of samples  $\{\mathbf{w}_m\}$ , and at the same time drawing  $M$  samples  $\{\mathbf{x}_m\}$  from  $p(\mathbf{x})$ , we obtain an unbiased stochastic gradient as follows

$$\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{M} \sum_m \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_m, \mathbf{w}_m, \boldsymbol{\theta}). \quad (3.39)$$

Hence, by repeatedly sampling minibatches from  $p(\mathbf{w} | D)$  and  $p(\mathbf{x})$ , calculating  $\hat{\mathbf{g}}(\boldsymbol{\theta})$  and using it to update  $\boldsymbol{\theta}$ , we can minimize  $E(\boldsymbol{\theta})$ . The above procedure is outlined in the algorithm below.

- (i) Generate a minibatch  $\{\mathbf{w}_m\}$  of size  $M$  from  $p(\mathbf{w} | D)$  by simulating the Markov chain.
  - (ii) Generate a minibatch  $\{\mathbf{x}_m\}$  of size  $M$  from  $p(\mathbf{x})$ .
  - (iii) Calculate the stochastic gradient  $\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{M} \sum_m \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_m, \mathbf{w}_m, \boldsymbol{\theta})$ .
  - (iv) Make an update on  $\boldsymbol{\theta}$  using  $\hat{\mathbf{g}}(\boldsymbol{\theta})$ .
  - (v) If  $\boldsymbol{\theta}$  has not converged yet, go back to step (i).

Note that, in the above algorithm, MCMC samples are not independent. Convergence of stochastic approximation algorithms of the above type, where samples are not independent, has been studied by Younes (1999). If desired, dependence between samples can be reduced by thinning or using parallel Markov chains.

To summarize, in order to make the loss function suitable for online distillation,  $t(\mathbf{x})$  was replaced by  $t(\mathbf{x}, \mathbf{w})$  and then the whole loss function was averaged with respect to  $p(\mathbf{w} | D)$ . It is natural to ask what effect this change to the loss function has. For cross entropy, it turns out that the two versions of the loss function are actually equivalent. To see why, rewrite cross entropy as follows

$$\begin{aligned} \langle E_{CE}(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} &= - \left\langle t(\mathbf{x}) \log f(\mathbf{x} | \boldsymbol{\theta}) + (1 - t(\mathbf{x})) \log(1 - f(\mathbf{x} | \boldsymbol{\theta})) \right\rangle_{p(\mathbf{x})} \\ &= - \left\langle \langle t(\mathbf{x}, \mathbf{w}) \rangle_{p(\mathbf{w} | D)} \log f(\mathbf{x} | \boldsymbol{\theta}) + \left(1 - \langle t(\mathbf{x}, \mathbf{w}) \rangle_{p(\mathbf{w} | D)}\right) \log(1 - f(\mathbf{x} | \boldsymbol{\theta})) \right\rangle_{p(\mathbf{x})} \\ &= - \left\langle t(\mathbf{x}, \mathbf{w}) \log f(\mathbf{x} | \boldsymbol{\theta}) + (1 - t(\mathbf{x}, \mathbf{w})) \log(1 - f(\mathbf{x} | \boldsymbol{\theta})) \right\rangle_{p(\mathbf{w} | D)p(\mathbf{x})} \\ &= \langle E_{CE}(\mathbf{x}, \mathbf{w}, \boldsymbol{\theta}) \rangle_{p(\mathbf{w} | D)p(\mathbf{x})}. \end{aligned} \quad (3.40)$$

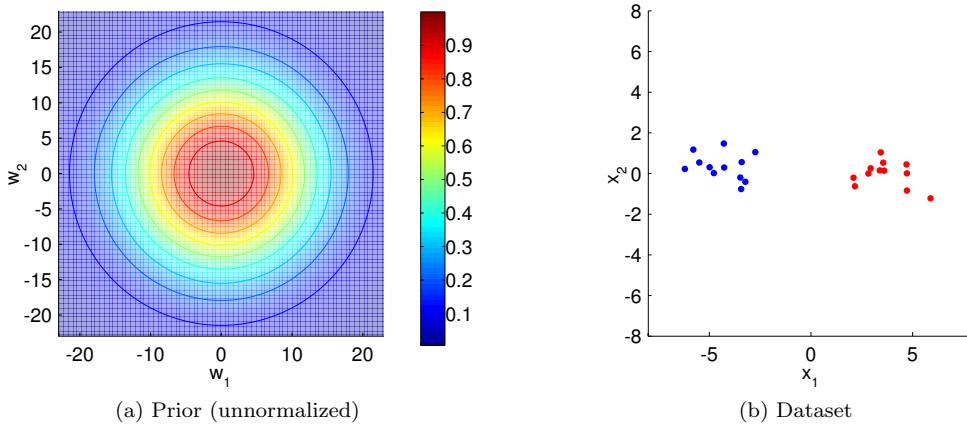


Figure 3.10: Setup for Bayesian logistic regression. The Gaussian prior over the weights is shown in (a). The dataset is shown in (b), with red corresponding to  $y = 1$  and blue corresponding to  $y = 0$ .

Hence, when using cross entropy, both batch and online distillation minimize the same loss function. The same is not true when using derivative square error though. In derivative square error, the term  $t(\mathbf{x})$  appears inside a log and a square, both of which are non-linear functions. Therefore, expectations with respect to  $p(\mathbf{w} | D)$  cannot be moved outside the loss function without changing it. In our case study in section 3.3.4, we will assess experimentally whether this change in derivative square error affects distillation performance.

### 3.3.4 Case study: Bayesian logistic regression

In this section, we apply the distillation framework to the problem of Bayesian logistic regression. Given a dataset with known binary labels, the task in Bayesian logistic regression is to infer the class probability for every possible input location.

#### The setup

Logistic regression is a probabilistic linear model for binary classification. Under a logistic regression model with weights  $\mathbf{w}$ , the probability of a datapoint  $\mathbf{x}$  having  $y = 1$  is given by

$$p(y = 1 | \mathbf{x}, \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}), \quad (3.41)$$

where  $\sigma(\cdot)$  is the logistic sigmoid function. In our experiments, we used a broad Gaussian prior for the weights (shown in Figure 3.10a) that is given by

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w} | \mathbf{0}, 100\mathbf{I}). \quad (3.42)$$

We created a toy dataset  $D$  of 24 two-dimensional input points  $\{\mathbf{x}_n\}$ , and we labelled 12 of them with  $y = 1$  and the other 12 with  $y = 0$ . The datapoints are shown in Figure 3.10b, colour-coded according to their labels. Given the dataset, the task is to calculate the predictive probability  $p(y = 1 | \mathbf{x}, D)$  for every input location  $\mathbf{x}$ . Despite the simplicity of the logistic regression model, calculating the predictive distribution is analytically intractable due to the sigmoid likelihood factor, and typically an approximation like MCMC is needed.

#### Distillation details

In our experiments, we used slice sampling (Neal, 2000) to generate MCMC samples from the posterior  $p(\mathbf{w} | D)$ . Slice sampling has the advantage over other MCMC methods that it is broadly

applicable and requires minimal tuning. We used linear stepping out, and we performed univariate updates to each parameter in turn. In all experiments, we initialized the Markov chain at  $\mathbf{0}$ , burned it in for 1000 iterations and used no thinning.

Having collected a set of MCMC samples  $\{\mathbf{w}_s\}$ , the predictive distribution is given by the following Monte Carlo estimate

$$t(\mathbf{x}) = p(y = 1 | \mathbf{x}, D) \approx \frac{1}{S} \sum_s \sigma(\mathbf{w}_s^T \mathbf{x}). \quad (3.43)$$

For batch distillation, we used  $S = 10,000$  MCMC samples to form the above estimate. Following Snelson and Ghahramani (2005a), the compact model we used in our experiments is of the same form as the above Monte Carlo estimate, but with only few components. That is, the compact model is defined to be

$$f(\mathbf{x} | \boldsymbol{\theta}) = p(y = 1 | \mathbf{x}, \boldsymbol{\theta}) = \frac{1}{S'} \sum_{s'} \sigma(\mathbf{w}_{s'}^T \mathbf{x}), \quad (3.44)$$

with  $S' \ll S$  and  $\boldsymbol{\theta} = \{\mathbf{w}_{s'}\}$ . In all our experiments, we used  $S' = 10$ . Note that the parameters  $\{\mathbf{w}_{s'}\}$  of the compact model can be viewed as a small, compact set of 10 weight samples. Hence, distillation in this case can be interpreted as trying to compress the knowledge of the whole Markov chain into only 10 samples. The locations of those 10 samples are optimized during training, in order to yield a predictive distribution that is as close as possible to the predictive distribution obtained by the whole chain.

As discussed in the previous sections, in order to train the compact model  $f(\mathbf{x} | \boldsymbol{\theta})$ , we need to be in a position to calculate the gradient of the loss function with respect to  $\boldsymbol{\theta}$ . Both the MCMC predictor  $t(\mathbf{x})$  and the compact predictor  $f(\mathbf{x} | \boldsymbol{\theta})$  are given by a linear combination of sigmoid terms. Hence, they can be viewed as feedforward neural networks with a single hidden layer of logistic sigmoid units, and a linear output layer. Therefore, the gradients of the loss functions can be easily calculated using standard backprop and R{backprop}, which are described in detail in appendix B.

For optimizing the loss functions, in all our experiments we performed 5000 iterations, using minibatches of size 10. We used learning rates that were initially set at 1 and linearly decayed to reach 0 in the last iteration. Finally, minibatches of input locations were independently drawn from the following input distribution

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x} | \mathbf{0}, 100\mathbf{I}). \quad (3.45)$$

## Results and discussion

Figure 3.11 shows graphically the MCMC predictor that was used in batch distillation, which was formed by 10,000 MCMC samples drawn from  $p(\mathbf{w} | D)$ . The left hand side plot shows the posterior  $p(\mathbf{w} | D)$  and 10% of all MCMC samples drawn from it. The right hand side plot shows the predictive distribution  $p_{MC}(y = 1 | \mathbf{x})$ , formed by averaging over all MCMC samples. Since the number of MCMC samples is large, we assume that this predictive distribution is a close approximation to the true one.

The shape of the MCMC predictive distribution shows how Bayesian logistic regression works. In the regions of input space that are densely populated by datapoints, Bayesian logistic regression is fairly confident about the value of  $y$ , which agrees with the true data labels. However, as we move away from the datapoints, the model becomes more and more uncertain about the value of  $y$ . This uncertainty is due to the small size of the dataset and the fact that it can be linearly separated by a wide range of linear separators. This is also reflected by the shape of the posterior, which shows that the model considers quite a large region of the weight space to be plausible. This region corresponds to the linear separators which Bayesian logistic regression has not yet received enough evidence to rule out.

Despite the fact that the MCMC predictive distribution is given by a sum of 10,000 logistic sigmoid terms, it corresponds to a rather simple and smooth surface. Hence, it is reasonable to

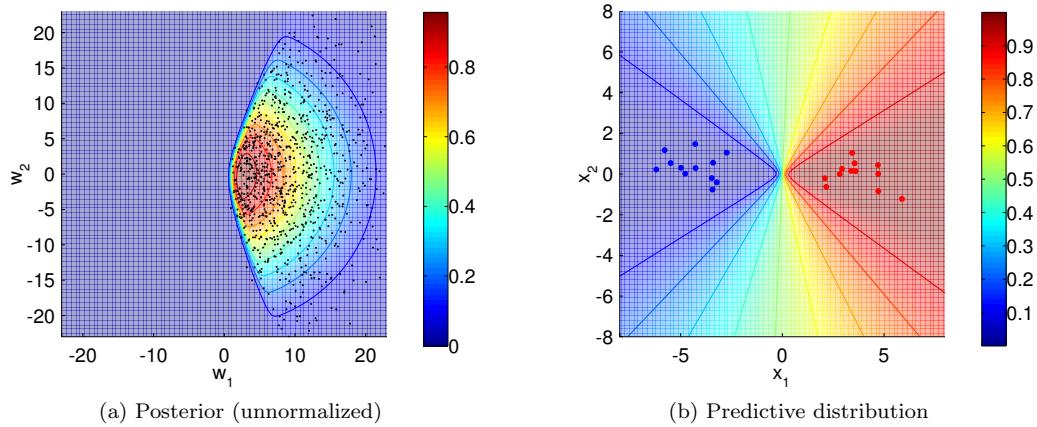


Figure 3.11: Bayesian logistic regression using MCMC. Black dots in (a) correspond to 10% of all MCMC samples.

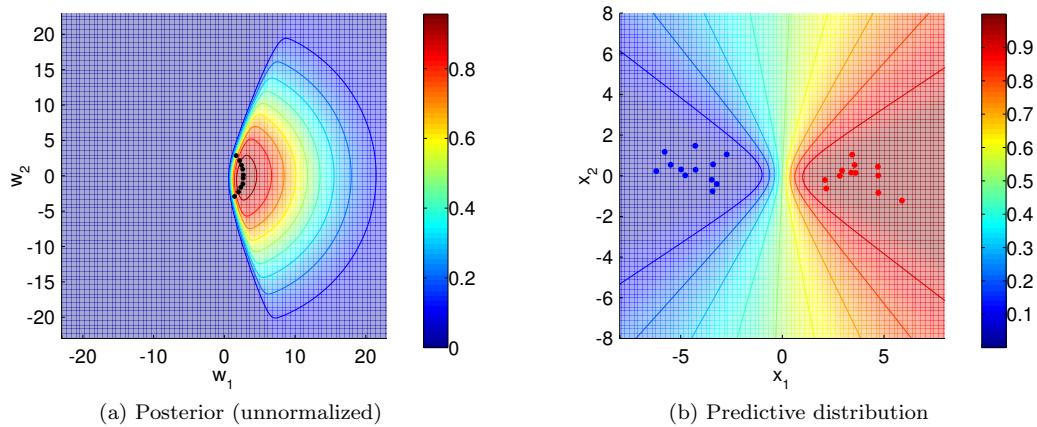


Figure 3.12: Batch distillation using cross entropy. Black dots in (a) correspond to the compact model's weights.

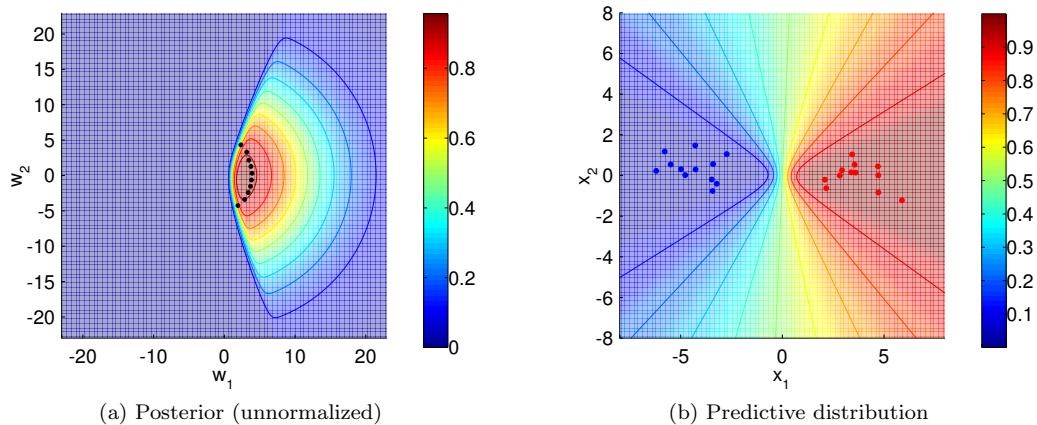


Figure 3.13: Batch distillation using derivative square error. Black dots in (a) correspond to the compact model's weights.

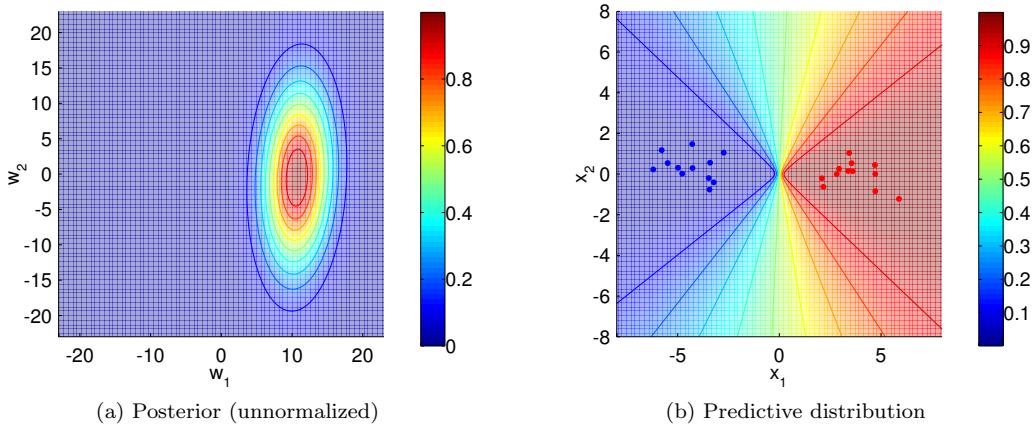


Figure 3.14: Bayesian logistic regression using EP. The posterior in (a) is the Gaussian approximation to the true posterior.

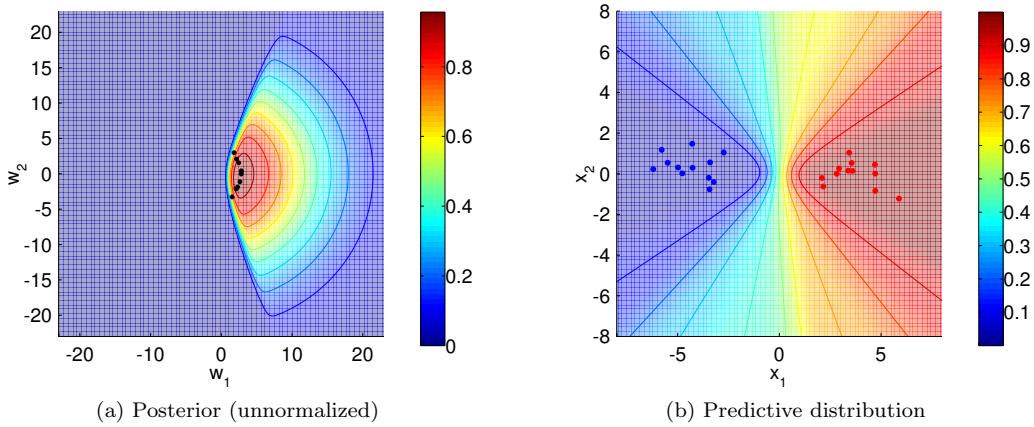


Figure 3.15: Online distillation using cross entropy. Black dots in (a) correspond to the compact model's weights.

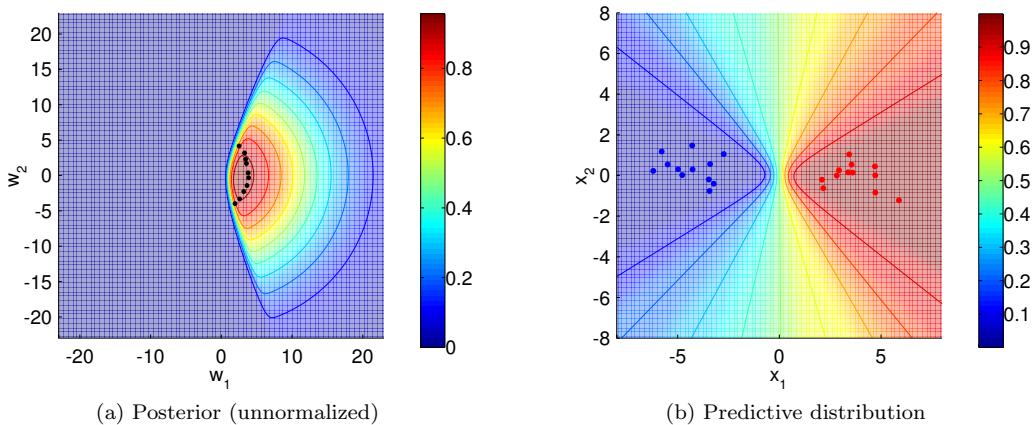


Figure 3.16: Online distillation using derivative square error. Black dots in (a) correspond to the compact model's weights.

expect that this surface can be accurately captured by a more compact model. Using the compact model described in Equation (3.44), we performed both batch and online distillation, using both cross entropy and derivative square error, resulting in 4 compact models in total. Figures 3.12, 3.13, 3.15 and 3.16 show graphically the 4 compact models. The left hand side plot shows the 10 weights  $\{\mathbf{w}_{s'}\}$  of each compact model, overlaid on top of the posterior  $p(\mathbf{w} | D)$ . The right hand side plot shows the predictive distribution  $p(y = 1 | \mathbf{x}, \boldsymbol{\theta})$ .

It can be seen that all 4 compact models yield a predictive distribution that is similar to that of the MCMC model. The region where the compact models and the MCMC model appear to be most dissimilar is around the origin, between the red and blue datapoints. In this region, the predictive surface of the compact models appears to be smoother, whereas the MCMC model has a steeper slope. Still, all compact models correctly capture the uncertainty Bayesian logistic regression should have in regions away from the datapoints. As can be seen by the plots on the left hand side, the compact model chose an elegant way to approximate the MCMC model by placing its weights in a semicircle, spanning the region of plausible weights.

Compared to cross entropy, derivative square error appears to yield a better approximation of the MCMC model. From the left hand side plots, it can be seen that the weights found by derivative square error have larger magnitude than those found by cross entropy. As a result, the predictive distribution of derivative square error better captures the steep slope of the MCMC model around the origin. This suggests that using derivatives in the loss function can convey more information about the shape of the predictive surface, and hence lead to more accurate compact models.

We can see that online distillation produced similar results to batch distillation. Note that batch distillation needs to store the full set of 10,000 MCMC samples, whereas online distillation only needs to store 10 MCMC samples at any time (i.e. as many as the minibatch size). This shows that compact models of the same quality can be learnt using only a fraction of the memory. This can be particularly useful in situations where the weights  $\mathbf{w}$  are high-dimensional, and storing more than few of them becomes prohibitively expensive.

For comparison, we also used Expectation Propagation to calculate an approximation to the predictive distribution. Expectation Propagation (Minka, 2001) works by first finding a Gaussian approximation to the posterior

$$p(\mathbf{w} | D) \approx \mathcal{N}(\mathbf{w} | \mathbf{m}, \mathbf{S}), \quad (3.46)$$

and then using this Gaussian approximation instead of the true posterior in calculating the predictive distribution. For Bayesian logistic regression, it is easy to show that the EP predictive distribution is given by

$$p(y = 1 | \mathbf{x}, D) \approx \int \sigma(z) \mathcal{N}(z | \mathbf{m}^T \mathbf{x}, \mathbf{x}^T \mathbf{S} \mathbf{x}) dz. \quad (3.47)$$

The above integral is always one-dimensional, regardless of the dimensionality of  $\mathbf{x}$ , but cannot be solved analytically. There are two ways to approximate it: (a) using quadrature, and (b) using the following approximation, due to MacKay (1992)

$$\int \sigma(z) \mathcal{N}(z | \mathbf{m}^T \mathbf{x}, \mathbf{x}^T \mathbf{S} \mathbf{x}) dz \approx \sigma(z^*) \quad \text{where} \quad z^* = \frac{\mathbf{m}^T \mathbf{x}}{\sqrt{1 + \frac{\pi}{8} \mathbf{x}^T \mathbf{S} \mathbf{x}}}. \quad (3.48)$$

In our implementation of EP, we used the MacKay approximation for integrals of the above form in calculating the Gaussian approximate posterior, and we used global adaptive quadrature (Shampine, 2008) for calculating predictions. Minka (2008) provides all the details of how to implement EP for Bayesian logistic regression.

Figure 3.14 shows the approximate Gaussian posterior (left plot) and the predictive distribution (right plot) as computed by EP. Compared to the compact models, we can see that the EP predictive distribution manages to better capture the steepness of the MCMC prediction surface around the origin, but also appears to slightly underestimate the correct uncertainty away from the datapoints. In terms of compactness, EP only needs to store the mean  $\mathbf{m}$  and the covariance  $\mathbf{S}$  of the Gaussian approximate posterior, so it is competitive to distillation. However, distillation

has a significant advantage over EP in terms of prediction speed. Indeed, for every location  $\mathbf{x}$ , EP needs to numerically calculate a one-dimensional integral, which is significantly more expensive compared to the compact model, whose predictive distribution is given in closed form.

## 3.4 Related work

In this section, we review work related to compact Bayesian inference, and compare it to our knowledge distillation framework presented in this chapter. We first review approaches that are based on knowledge distillation and are the most similar to our work. Then, we briefly review deterministic approximate inference, which provides an alternative to knowledge distillation for compact Bayesian inference.

### 3.4.1 Distillation-based approaches

Snelson and Ghahramani (2005a) were the first who proposed training compact models to approximate Bayesian predictive distributions. In their framework, the true predictive distribution is approximated either by a large bag of MCMC samples or by deterministic methods such as EP. Training is done by minimizing the KL divergence from the approximate predictive distribution to the one given by the compact model.

Our knowledge distillation framework described in this chapter includes and extends that of Snelson and Ghahramani (2005a). Their methods have been incorporated into our framework under the name “batch distillation with KL divergence”. Also, we followed their experiments in our case studies and confirmed their results. Moreover, our work extends theirs by (a) showing how distillation can be performed in an online fashion and (b) incorporating derivative information into the distillation procedure.

Recently, Korattikara et al. (2015) proposed what they call *Bayesian dark knowledge*, which is also a framework for distilling compact predictive distributions from MCMC samples. Their work focuses on Bayesian neural networks (Neal, 1996), which are neural networks that make predictions by taking into account the uncertainty in their parameters. Bayesian neural networks pose a hard inference problem; the typical MCMC-based solution is to sample a large collection of networks using MCMC, and then make predictions by averaging the predictions of all networks in the collection.

Bayesian dark knowledge aims at distilling the whole collection of MCMC-sampled networks into a single compact network, by minimizing the KL divergence from the true predictive distribution to the one given by the compact network. Distillation takes place on the fly as the networks are generated by the Markov chain, thanks to which there is no need for storing more than one MCMC-sampled network at any time. Every time a network is generated by the Markov chain, the parameters of the compact network are updated using a stochastic gradient step.

The distillation procedure used in Bayesian dark knowledge corresponds to “online distillation with cross entropy” in our framework. Note that Bayesian dark knowledge was developed at the same time as us but independently. It was published only after we had finished with our own work, so we were not aware of it when developing our framework.

### 3.4.2 Deterministic approximate inference

Knowledge distillation attempts to construct a compact approximation directly to the predictive distribution. However, this is not the only approach to making Bayesian inference compact. A different approach is to construct a compact approximation not to the predictive distribution but to the posterior. Then, every time a prediction needs to be made, the compact approximate posterior can be used to calculate it.

A general framework for approximating complex posteriors with simpler distributions is *variational inference* (Jordan et al., 1999; Wainwright and Jordan, 2008). In variational inference, the approximate posterior is chosen such that expectations over it are easy to compute. Nevertheless, there is nothing in the framework stopping us to also choose it to be compact.

Two general purpose implementations of variational inference that typically lead to compact approximate posteriors are *variational message passing* (Winn and Bishop, 2005) and *Expectation Propagation* (Minka, 2001). Both of them work by locally passing messages in a graphical model, and their main difference lies in the variational objective they choose to minimize. The approximate posterior they compute is an exponential family distribution, hence it can be compactly represented by the vector of its natural parameters. EP was showcased in our experiments in section 3.3.4, where a Gaussian approximation to the true posterior was computed.

Having a posterior in a compact form does provide savings in storage, but it does not necessarily provide savings in computation at prediction time. This is because every time a prediction needs to be made, an expectation over the approximate posterior has to be computed. If this expectation can be computed analytically, then typically making predictions is fast. However, if numerical integration has to be used (such as in our EP example in section 3.3.4), then making predictions can be slow. In contrast, knowledge distillation provides a closed form approximation directly to the predictive distribution, which leads to fast predictions by design.

## 3.5 Summary and conclusions

In this chapter we presented a framework for constructing compact approximations to Bayesian predictive distributions. Our approach is based on distilling the knowledge contained in a Markov chain designed to generate samples from the posterior. We successfully applied our framework in the problems of Bayesian density estimation and Bayesian binary classification.

Our framework extends the state of the art in two different ways. Firstly, we showed how knowledge distillation can be performed on the fly as the Markov chain generates samples. This technique, which we called online distillation, can significantly reduce memory requirements. Secondly, in the case of Bayesian binary classification, we showed how to use derivatives to convey more information about the true predictive surface to the compact model, which can lead to more effective distillation.

Our framework relies on the Markov chain being capable of effectively exploring the space of plausible weights, and hence the compact predictive distribution it computes will be only as good as this Markov chain. For our experiments slice sampling was sufficiently good, however more sophisticated MCMC methods can be used to improve the performance in harder inference tasks. For instance, Hamiltonian dynamics (Neal, 2010) can be used to reduce dependence between samples, and stochastic gradient Langevin dynamics (Welling and Teh, 2011) can be used to make MCMC scale to large datasets.

## Chapter 4

# Intractable Generative Models

A generative model is a probability distribution  $p(\mathbf{x})$  over some variable  $\mathbf{x}$ . In general,  $\mathbf{x}$  can be any sort of data, such as text, sound, images, or other models. Being able to represent, store and compute with  $p(\mathbf{x})$  is perhaps the most general problem in machine learning; if we know the probability of everything, then we can condition on the observations, integrate out the unknowns, and thus infer any quantity of interest.

Nevertheless, in practice all but the simplest generative models are intractable. In this chapter, we focus our attention on intractable generative models of the form

$$p(\mathbf{x}) = \frac{1}{Z} \bar{p}(\mathbf{x}), \quad (4.1)$$

where  $\bar{p}(\mathbf{x})$  is some tractable non-negative function (sometimes called a potential) and  $Z$  is an intractable partition function (also called the normalizing constant) given by

$$Z = \int \bar{p}(\mathbf{x}) d\mathbf{x}. \quad (4.2)$$

We will assume that we can easily evaluate  $\bar{p}(\mathbf{x})$  but not  $Z$ , and therefore not  $p(\mathbf{x})$ .

Distributions of the above type are particularly common in Bayesian inference. If  $\mathbf{x}$  is some quantity we wish to infer given some data  $D$ , the posterior over  $\mathbf{x}$  is given by Bayes rule

$$p(\mathbf{x} | D) = \frac{p(D | \mathbf{x}) p(\mathbf{x})}{p(D)}. \quad (4.3)$$

Even if the likelihood  $p(D | \mathbf{x})$  and the prior  $p(\mathbf{x})$  are tractable, the marginal likelihood  $p(D)$  and hence the posterior  $p(\mathbf{x} | D)$  can be intractable. Apart from the posterior, the marginal likelihood  $p(D)$  is also a quantity of interest, since it measures how well the model fits the data and so it can be used for model comparison.

Another example where distributions of the above type are commonplace is undirected graphical models. An undirected graphical model (Koller and Friedman, 2009) defines a distribution as

$$p(\mathbf{x}) = \frac{1}{Z} \exp(-U(\mathbf{x})), \quad (4.4)$$

where  $U(\mathbf{x})$  is some tractable energy function. Even if the energy is tractable,  $Z$  and therefore  $p(\mathbf{x})$  can still be intractable. The partition function  $Z$  is also a quantity of interest, since useful properties of the model can be obtained from the derivatives of  $\log Z$  with respect to the model parameters—for models in the exponential family, for instance, these derivatives correspond exactly to the expected sufficient statistics under the model.

In this chapter, we present a framework for distilling intractable generative models such as the above into tractable generative models of our choosing. After successful distillation, we can use the tractable model in place of the intractable one in order to perform computations

that were previously infeasible. We showcase our framework by distilling an intractable RBM into a tractable NADE, and we demonstrate that the distilled NADE can be successfully used in combination with simple sampling schemes for robustly estimating the RBM’s intractable partition function.

## 4.1 Distilling intractable into tractable generative models

Let  $p(\mathbf{x})$  be some generative model we are interested in but can only evaluate up to a multiplicative constant; that is,  $p(\mathbf{x})$  is given by

$$p(\mathbf{x}) = \frac{1}{Z} \bar{p}(\mathbf{x}), \quad (4.5)$$

where  $\bar{p}(\mathbf{x})$  is easy to calculate but  $Z$  is not. Assume we can construct a tractable generative model  $p(\mathbf{x} | \boldsymbol{\theta})$ , parameterized by a set of parameters  $\boldsymbol{\theta}$ , which is flexible enough to represent complex distributions. Our goal is to distil the knowledge contained in  $p(\mathbf{x})$  into  $p(\mathbf{x} | \boldsymbol{\theta})$ ; that is, we want to train  $p(\mathbf{x} | \boldsymbol{\theta})$  to do the same job as  $p(\mathbf{x})$ .

Our approach is based on minimizing an appropriate loss function  $E(\boldsymbol{\theta})$  that measures the discrepancy between  $p(\mathbf{x})$  and  $p(\mathbf{x} | \boldsymbol{\theta})$  as a function of  $\boldsymbol{\theta}$ . The challenge is to specify a loss function that only involves tractable quantities and find a way to minimize it efficiently. In the following two sections, we describe and analyze our choice of appropriate loss functions, together with a stochastic way of minimizing them. As we shall see, our information about  $p(\mathbf{x})$  will come from MCMC samples and the values of  $\bar{p}(\mathbf{x})$  at sample locations.

### 4.1.1 Loss functions

In measuring the discrepancy between  $p(\mathbf{x})$  and  $p(\mathbf{x} | \boldsymbol{\theta})$ , the challenge is to come up with a loss function that avoids intractable quantities such as  $p(\mathbf{x})$  and  $Z$ , but at the same time captures as much information about  $p(\mathbf{x})$  as possible. In the following, we describe three loss functions that achieve this goal; *KL divergence*, *square error* and *score matching*.

#### KL divergence

A natural measure of discrepancy is the KL divergence from  $p(\mathbf{x})$  to  $p(\mathbf{x} | \boldsymbol{\theta})$ , which is defined as

$$E_{\text{KL}}(\boldsymbol{\theta}) = D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x} | \boldsymbol{\theta})) = \left\langle \log \frac{p(\mathbf{x})}{p(\mathbf{x} | \boldsymbol{\theta})} \right\rangle_{p(\mathbf{x})}. \quad (4.6)$$

It is well known that the KL divergence is non-negative, and is equal to zero if and only if  $p(\mathbf{x}) = p(\mathbf{x} | \boldsymbol{\theta})$  (MacKay, 2002, section 2.6). Thus, if  $p(\mathbf{x} | \boldsymbol{\theta})$  has sufficient capacity, globally minimizing  $E_{\text{KL}}(\boldsymbol{\theta})$  will result in  $p(\mathbf{x} | \boldsymbol{\theta})$  exactly matching  $p(\mathbf{x})$ .

In order to efficiently minimize  $E_{\text{KL}}(\boldsymbol{\theta})$  with respect to  $\boldsymbol{\theta}$  while avoiding intractable quantities, note that

$$E_{\text{KL}}(\boldsymbol{\theta}) = \langle \log p(\mathbf{x}) \rangle_{p(\mathbf{x})} - \langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} = -\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} + \text{const.} \quad (4.7)$$

Therefore, minimizing  $E_{\text{KL}}(\boldsymbol{\theta})$  is equivalent to maximizing  $\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}$ . This can be interpreted as fitting  $p(\mathbf{x} | \boldsymbol{\theta})$  using maximum likelihood to an infinite amount of data generated from  $p(\mathbf{x})$ . Later on, we will use MCMC to generate such data.

The KL divergence has a nice information-theoretic interpretation; it measures the extra number of nats (or bits, if the logs are base 2) that are needed in order to encode samples  $\mathbf{x}$  from  $p(\mathbf{x})$  using  $p(\mathbf{x} | \boldsymbol{\theta})$  (MacKay, 2002, section 5.4). In other words, it is the encoding cost we must pay for using the wrong distribution.

### Square error

Another approach to measuring discrepancy, is to directly measure how much the values of  $p(\mathbf{x} | \boldsymbol{\theta})$  differ from the values of  $p(\mathbf{x})$ . A natural choice is the average square error of the logarithms, given by<sup>1</sup>

$$E_{\text{SE}}^0(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \|\log p(\mathbf{x} | \boldsymbol{\theta}) - \log p(\mathbf{x})\|^2 \right\rangle_{p(\mathbf{x})}. \quad (4.8)$$

Obviously, directly working with the above is not possible, since we have assumed that directly evaluating  $\log p(\mathbf{x})$  is intractable. However, it is possible to circumvent this problem by using the following tractable loss function instead

$$E_{\text{SE}}(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \|\log p(\mathbf{x} | \boldsymbol{\theta}) - \log \bar{p}(\mathbf{x}) + c\|^2 \right\rangle_{p(\mathbf{x})}, \quad (4.9)$$

where  $c$  is an appropriately chosen constant. The following proposition establishes the range of  $c$  values for which minimizing  $E_{\text{SE}}(\boldsymbol{\theta})$  correctly trains  $p(\mathbf{x} | \boldsymbol{\theta})$  to match  $p(\mathbf{x})$ .

**Proposition 4.1.** *Assuming  $p(\mathbf{x} | \boldsymbol{\theta})$  has sufficient capacity and  $c \leq \log Z$ ,  $E_{\text{SE}}(\boldsymbol{\theta})$  is globally minimized if and only if  $p(\mathbf{x} | \boldsymbol{\theta}) = p(\mathbf{x})$ .*

*Proof.* Using the fact that  $\bar{p}(\mathbf{x}) = Zp(\mathbf{x})$  and the definition of the KL divergence,  $E_{\text{SE}}(\boldsymbol{\theta})$  can be decomposed as follows

$$E_{\text{SE}}(\boldsymbol{\theta}) = E_{\text{SE}}^0(\boldsymbol{\theta}) + (\log Z - c)E_{\text{KL}}(\boldsymbol{\theta}) + \frac{1}{2} \|\log Z - c\|^2, \quad (4.10)$$

where

$$E_{\text{SE}}^0(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \|\log p(\mathbf{x} | \boldsymbol{\theta}) - \log p(\mathbf{x})\|^2 \right\rangle_{p(\mathbf{x})} \quad (4.11)$$

and

$$E_{\text{KL}}(\boldsymbol{\theta}) = D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x} | \boldsymbol{\theta})). \quad (4.12)$$

Note that both  $E_{\text{SE}}^0(\boldsymbol{\theta})$  and  $E_{\text{KL}}(\boldsymbol{\theta})$  are non-negative. Therefore, assuming  $c \leq \log Z$ , the global minimum of  $E_{\text{SE}}(\boldsymbol{\theta})$  is achieved if and only if  $E_{\text{SE}}^0(\boldsymbol{\theta}) = E_{\text{KL}}(\boldsymbol{\theta}) = 0$ , which in turn is achieved if and only if  $p(\mathbf{x} | \boldsymbol{\theta}) = p(\mathbf{x})$ .  $\square$

The above proposition tells us that minimizing  $E_{\text{SE}}(\boldsymbol{\theta})$  is equivalent to minimizing  $E_{\text{SE}}^0(\boldsymbol{\theta})$  regularized by  $E_{\text{KL}}(\boldsymbol{\theta})$ , with regularizer  $(\log Z - c)$ . Therefore, as long as the regularizer is non-negative,  $E_{\text{SE}}(\boldsymbol{\theta})$  is a sensible loss.

In order to set  $c$ , we do not necessarily need to know  $\log Z$ , but we only need to lower bound it. If  $\mathbf{x}$  is discrete, this becomes trivial since for any  $\mathbf{x}$

$$p(\mathbf{x}) \leq 1 \Rightarrow \log \bar{p}(\mathbf{x}) \leq \log Z. \quad (4.13)$$

Thus, setting  $c$  less or equal to  $\max_{\mathbf{x}} \log \bar{p}(\mathbf{x})$  is guaranteed to work. More generally, lower bounding  $\log Z$  can be done using the following inequality based on the variational free energy

$$\langle \log q(\mathbf{x}) \rangle_{q(\mathbf{x})} - \langle \log \bar{p}(\mathbf{x} | \boldsymbol{\theta}) \rangle_{q(\mathbf{x})} \leq \log Z. \quad (4.14)$$

The above inequality holds for any distribution  $q(\mathbf{x})$ , so  $q(\mathbf{x})$  can be freely chosen to be convenient. Finally, note that in the limit case where  $c \rightarrow -\infty$ , minimizing the square error loss becomes equivalent to minimizing the KL divergence loss.

We conclude our analysis with a final remark about the relationship between the square error loss and the KL divergence loss. Using Jensen's inequality and the convexity of norms, we can easily show that

$$E_{\text{SE}}(\boldsymbol{\theta}) \geq \frac{1}{2} \|E_{\text{KL}}(\boldsymbol{\theta}) + (\log Z - c)\|^2. \quad (4.15)$$

Hence, as long as  $c \leq \log Z$ , minimizing the square error loss corresponds to minimizing an upper bound of the squared KL divergence loss plus a positive constant.

---

<sup>1</sup>Using a norm here is not necessary, since the quantities are scalar, but we find it aesthetically pleasing.

### Score matching

Score matching was first proposed by Hyvärinen (2005), where it was used to learn unnormalizable generative models. It is based on the clever trick that the derivative of the log probability does not depend on  $Z$ , that is

$$\frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x}) = \frac{\partial}{\partial \mathbf{x}} \log \bar{p}(\mathbf{x}). \quad (4.16)$$

Therefore, even though  $\log p(\mathbf{x})$  can be intractable, its derivative with respect to  $\mathbf{x}$  is not.

In the context of knowledge distillation, the intuition behind score matching is to train  $p(\mathbf{x} | \boldsymbol{\theta})$  to mimic  $p(\mathbf{x})$  by matching their derivatives with respect to  $\mathbf{x}$ . Since they are both distributions, and therefore have to integrate to 1, matching derivatives provides sufficient information for training. The score matching loss function is defined as

$$E_{SM}(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \left\| \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x} | \boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x}) \right\|^2 \right\rangle_{p(\mathbf{x})}. \quad (4.17)$$

The following proposition, adapted from Hyvärinen (2005, theorem 2), establishes the validity of the score matching loss for the continuous case.

**Proposition 4.2.** *Assuming  $p(\mathbf{x} | \boldsymbol{\theta})$  has sufficient capacity,  $\mathbf{x}$  is continuous and  $p(\mathbf{x}) > 0$  everywhere,  $E_{SM}(\boldsymbol{\theta})$  is globally minimized if and only if  $p(\mathbf{x} | \boldsymbol{\theta}) = p(\mathbf{x})$ .*

*Proof.* First notice that  $E_{SM}(\boldsymbol{\theta}) \geq 0$  as the expectation of a norm. If if  $p(\mathbf{x} | \boldsymbol{\theta}) = p(\mathbf{x})$  then trivially  $E_{SM}(\boldsymbol{\theta}) = 0$  and thus it is globally minimized. Conversely, if  $E_{SM}(\boldsymbol{\theta}) = 0$  then we have that for all  $\mathbf{x}$

$$\frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x} | \boldsymbol{\theta}) = \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x}) \Rightarrow p(\mathbf{x} | \boldsymbol{\theta}) = c p(\mathbf{x}), \quad (4.18)$$

for some constant  $c > 0$ . But since both  $p(\mathbf{x} | \boldsymbol{\theta})$  and  $p(\mathbf{x})$  must integrate to 1, we have

$$c = \int c p(\mathbf{x}) d\mathbf{x} = \int p(\mathbf{x} | \boldsymbol{\theta}) d\mathbf{x} = 1, \quad (4.19)$$

hence  $p(\mathbf{x} | \boldsymbol{\theta}) = p(\mathbf{x})$ .  $\square$

It is important to note that score matching works only in the continuous case and therefore it cannot be used for discrete distributions. Indeed, even though we can in principle calculate the derivatives of  $\log p(\mathbf{x} | \boldsymbol{\theta})$  and  $\log p(\mathbf{x})$  at every location in the discrete domain of  $\mathbf{x}$ , making them equal does not necessarily make the distributions equal. To handle this, an extension to score matching termed *ratio matching* was proposed by Hyvärinen (2007) for learning unnormalizable discrete generative models. Ratio matching is based on matching conditional distributions, which do not depend on  $Z$ . However, it is not obvious how to adapt it within the context of knowledge distillation.

Another caveat of score matching is that it requires the support of  $p(\mathbf{x})$  to be the whole space. To see why, let the support of  $p(\mathbf{x})$  be  $S_p$  and assume it is not the whole space. Then, a distribution satisfying  $p(\mathbf{x} | \boldsymbol{\theta}) = c p(\mathbf{x})$  with  $0 < c < 1$  for all  $\mathbf{x}$  in  $S_p$  will globally minimize  $E_{SM}(\boldsymbol{\theta})$  without perfectly matching  $p(\mathbf{x})$ . In other words, score matching does not constrain the behaviour of  $p(\mathbf{x} | \boldsymbol{\theta})$  outside  $S_p$  and it only guarantees proportionality within  $S_p$ , hence if  $S_p$  is not the whole space,  $p(\mathbf{x} | \boldsymbol{\theta})$  is allowed to “leak” mass outside  $S_p$ .

#### 4.1.2 Stochastic gradient training

In the previous section, we described and analyzed the following three loss functions for measuring the discrepancy between  $p(\mathbf{x} | \boldsymbol{\theta})$  and  $p(\mathbf{x})$ .

(i) **KL divergence**

$$E_{KL}(\boldsymbol{\theta}) = -\langle \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} + \text{const} \quad (4.20)$$

## (ii) Square error

$$E_{\text{SE}}(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \|\log p(\mathbf{x} | \boldsymbol{\theta}) - \log \bar{p}(\mathbf{x}) + c\|^2 \right\rangle_{p(\mathbf{x})} \quad (4.21)$$

## (iii) Score matching

$$E_{\text{SM}}(\boldsymbol{\theta}) = \left\langle \frac{1}{2} \left\| \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x} | \boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x}) \right\|^2 \right\rangle_{p(\mathbf{x})} \quad (4.22)$$

All three loss functions exhibit a common pattern; they contain a tractable quantity within an intractable expectation over  $p(\mathbf{x})$ . That is, they all have the following form

$$E(\boldsymbol{\theta}) = \langle E(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}, \quad (4.23)$$

for an appropriate instantiation of  $E(\mathbf{x}, \boldsymbol{\theta})$ .

We will describe a stochastic gradient learning procedure for optimizing loss functions of the above form. The gradient of  $E(\boldsymbol{\theta})$  with respect to  $\boldsymbol{\theta}$  is

$$\mathbf{g}(\boldsymbol{\theta}) = \left\langle \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta}) \right\rangle_{p(\mathbf{x})} \quad (4.24)$$

Even though  $\mathbf{g}(\boldsymbol{\theta})$  is intractable, we can stochastically approximate it by

$$\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{S} \sum_s \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_s, \boldsymbol{\theta}), \quad (4.25)$$

where  $\{\mathbf{x}_s\}$  are samples generated from  $p(\mathbf{x})$ . Typically, we can generate  $\{\mathbf{x}_s\}$  using MCMC. It is easy to see that, in expectation, the stochastic gradient is equal to the original gradient. Using the above stochastic gradient, we can minimize  $E(\boldsymbol{\theta})$  using the following procedure.

- (i) Generate a minibatch  $\{\mathbf{x}_s\}$  of size  $S$  from  $p(\mathbf{x})$  using MCMC.
- (ii) Calculate the stochastic gradient  $\hat{\mathbf{g}}(\boldsymbol{\theta}) = \frac{1}{S} \sum_s \frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}_s, \boldsymbol{\theta})$ .
- (iii) Make an update on  $\boldsymbol{\theta}$  using  $\hat{\mathbf{g}}(\boldsymbol{\theta})$ .
- (iv) Repeat until convergence.

Stochastic optimization algorithms of the above type are analyzed by Bottou (1998), where it is proved that—under certain regularity conditions—they converge almost surely to a stationary point of the objective function. However, their analysis assumes that samples generated from  $p(\mathbf{x})$  are exact and independent, therefore it does not apply when samples are generated by MCMC. Sufficient conditions for convergence even when samples are generated by MCMC are provided by Younes (1999).

To use the above algorithm in practice, the user has to specify (a) what MCMC algorithm to use to generate from  $p(\mathbf{x})$  and how to tune it, (b) what step size strategy to use in the updates on  $\boldsymbol{\theta}$  and (c) how to determine convergence. We will address these questions in the specific context of our case study in section 4.2.

To complete our discussion, we will provide some details on how to calculate  $\frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta})$  for each of the three loss functions under consideration.

## (i) KL divergence

$$\frac{\partial}{\partial \boldsymbol{\theta}} E_{\text{KL}}(\mathbf{x}, \boldsymbol{\theta}) = -\frac{\partial}{\partial \boldsymbol{\theta}} \log p(\mathbf{x} | \boldsymbol{\theta}) \quad (4.26)$$

(ii) **Square error**

$$\frac{\partial}{\partial \theta} E_{SE}(\mathbf{x}, \boldsymbol{\theta}) = \frac{\partial}{\partial \theta} \log p(\mathbf{x} | \boldsymbol{\theta}) (\log p(\mathbf{x} | \boldsymbol{\theta}) - \log \bar{p}(\mathbf{x}) + c) \quad (4.27)$$

(iii) **Score matching**

$$\frac{\partial}{\partial \theta} E_{SM}(\mathbf{x}, \boldsymbol{\theta}) = \frac{\partial^2}{\partial \theta \partial \mathbf{x}} \log p(\mathbf{x} | \boldsymbol{\theta}) \left( \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x} | \boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x}) \right) = \mathcal{R} \left\{ \frac{\partial}{\partial \theta} \log p(\mathbf{x} | \boldsymbol{\theta}) \right\} \quad (4.28)$$

As can be seen by the above equations, the only quantities we need to be in a position to calculate are

$$\frac{\partial}{\partial \theta} \log p(\mathbf{x} | \boldsymbol{\theta}) \quad \text{and} \quad \mathcal{R} \left\{ \frac{\partial}{\partial \theta} \log p(\mathbf{x} | \boldsymbol{\theta}) \right\}. \quad (4.29)$$

Note that in the case of score matching,  $\frac{\partial}{\partial \theta} E_{SM}(\mathbf{x}, \boldsymbol{\theta})$  takes the form of a Hessian-vector product, where the Hessian is that of  $\log p(\mathbf{x} | \boldsymbol{\theta})$  and the vector is  $(\frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x} | \boldsymbol{\theta}) - \frac{\partial}{\partial \mathbf{x}} \log p(\mathbf{x}))$ . Hence, we can efficiently calculate it in linear time using the R technique (as described in appendix A).

## 4.2 Case study: distilling RBM into NADE

In the previous section we described a general framework for distilling an intractable generative model  $p(\mathbf{x})$  into a tractable generative model  $p(\mathbf{x} | \boldsymbol{\theta})$ . In this section, we put the framework into practice in order to distil an intractable RBM into a tractable NADE.

### 4.2.1 The Restricted Boltzmann Machine

The Restricted Boltzmann Machine (RBM for short) is a widely-used generative model that is known to be a good distribution estimator (Salakhutdinov and Murray, 2008). It was introduced by Smolensky (1986) and in the last decade it has enjoyed considerable success as a building block of deep generative models (Bengio, 2009; Hinton et al., 2006). It owes its success to its flexibility in capturing high order dependencies between variables and to the existence of effective algorithms for training it (Hinton, 2002).

The RBM is a bipartite undirected graphical model (as shown in Figure 4.1), consisting of a layer of visible variables  $\mathbf{x}$  and a layer of hidden variables  $\mathbf{h}$ . It is “restricted” in the sense that there are no connections within layers, only between them. In this work, we focus on the binary version of the RBM, where each variable can either be 0 or 1, however real-valued versions exist (Nair and Hinton, 2010). From here on, whenever we say “RBM” we mean “binary RBM”.

The probability distribution defined by the RBM over the visible variables  $\mathbf{x}$  takes the following form

$$p(\mathbf{x}) = \frac{1}{Z} \sum_{\mathbf{h}} \exp (\mathbf{x}^T \mathbf{W} \mathbf{h} + \mathbf{a}^T \mathbf{x} + \mathbf{b}^T \mathbf{h}), \quad (4.30)$$

where  $\{\mathbf{W}, \mathbf{a}, \mathbf{b}\}$  are the parameters of the RBMs. Evaluating the unnormalized probability  $\bar{p}(\mathbf{x}) = Zp(\mathbf{x})$  seems to be intractable, since it is a sum of  $2^J$  terms, where  $J$  is the number of hidden variables. However, due to the special structure of the RBM, it is actually tractable. To see why, define the following quantity

$$\mathbf{z} = \mathbf{W}^T \mathbf{x} + \mathbf{b} \quad (4.31)$$

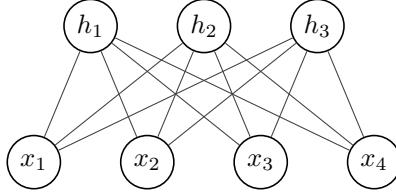


Figure 4.1: Graphical model of an RBM with 4 visible and 3 hidden variables. Its visible variables are  $x_1, x_2, x_3$  and  $x_4$  (bottom layer) and its hidden variables are  $h_1, h_2$  and  $h_3$  (top layer).

and rewrite  $\bar{p}(\mathbf{x})$  as follows

$$\begin{aligned}
 \bar{p}(\mathbf{x}) &= \exp(\mathbf{a}^T \mathbf{x}) \sum_{\mathbf{h}} \exp(\mathbf{z}^T \mathbf{h}) \\
 &= \exp(\mathbf{a}^T \mathbf{x}) \sum_{\mathbf{h}} \prod_j \exp(z_j h_j) \\
 &= \exp(\mathbf{a}^T \mathbf{x}) \prod_j \sum_{h_j} \exp(z_j h_j) \\
 &= \exp(\mathbf{a}^T \mathbf{x}) \prod_j (1 + \exp z_j).
 \end{aligned} \tag{4.32}$$

Hence,  $\bar{p}(\mathbf{x})$  can be computed in time which is linear in  $J$ .

Even though  $\bar{p}(\mathbf{x})$  is tractable, several quantities of interest associated with the RBM are intractable. The normalizing constant  $Z$  is intractable, as it is a sum of  $2^I$  terms where  $I$  is the number of visible variables. As a consequence,  $p(\mathbf{x})$  is intractable as well, and so are marginals  $p(x_i)$  and conditionals  $p(x_i | \mathbf{x}_{<i})$ . Finally, drawing exact samples from the RBM is also hard.

Even though exact sampling is hard, obtaining approximate samples from the RBM can be done using MCMC. A practical way to do it is using block Gibbs sampling. This technique is based on the fact that both  $p(\mathbf{h} | \mathbf{x})$  and  $p(\mathbf{x} | \mathbf{h})$  are easy to evaluate. Indeed, it is easy to show that

$$p(\mathbf{h} | \mathbf{x}) = \sigma(\mathbf{W}^T \mathbf{x} + \mathbf{b}) \tag{4.33}$$

and by symmetry

$$p(\mathbf{x} | \mathbf{h}) = \sigma(\mathbf{W}\mathbf{h} + \mathbf{a}), \tag{4.34}$$

where  $\sigma(\cdot)$  is the logistic sigmoid function applied elementwise. Given a sample  $\mathbf{x}_n$ , block Gibbs sampling draws a sample  $\mathbf{h}_n$  from  $p(\mathbf{h} | \mathbf{x}_n)$  and then draws a sample  $\mathbf{x}_{n+1}$  from  $p(\mathbf{x} | \mathbf{h}_n)$ . Starting from some seed value  $\mathbf{x}_0$  and iterating the above procedure  $N$  times, block Gibbs sampling generates a sequence  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$  of approximate correlated samples from  $p(\mathbf{x})$ . Of course, as with any MCMC method, letting the chain burn in, thinning samples or using parallel chains can all be employed to improve the quality of samples.

#### 4.2.2 The Neural Autoregressive Distribution Estimator

The Neural Autoregressive Distribution Estimator (NADE for short) is a generative model that is fully tractable, but at the same time flexible enough to model complex distributions. It was introduced by Larochelle and Murray (2011), where it was used for distribution estimation of binary data.

NADE is a feedforward neural network with a special autoregressive structure. The basic version of NADE has a single hidden layer and models binary data, however there are extensions that can handle real-valued data (Uria et al., 2013) and have multiple hidden layers (Uria et al., 2014). In this work we will make use of the binary, single hidden layer version, and from here on whenever we say “NADE” we shall refer to this.

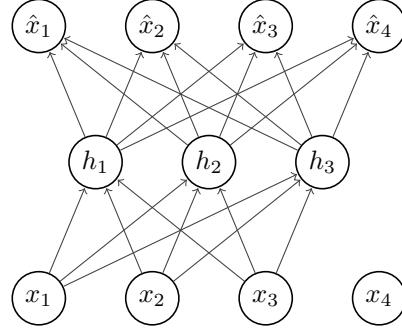


Figure 4.2: Architecture of a NADE with 4 input variables and 3 hidden units. Its input variables are  $x_1, x_2, x_3$  and  $x_4$  (bottom layer), its hidden units are  $h_1, h_2$  and  $h_3$  (middle layer) and its outputs are  $\hat{x}_1, \hat{x}_2, \hat{x}_3$  and  $\hat{x}_4$  (top layer). Note that input  $x_4$  is not actually forward-propagated, thus there are no connections from it to the hidden units.

NADE takes as input  $I$  binary variables  $\{x_i\}$  that can be either 0 or 1, and returns as output  $I$  real-valued variables  $\{\hat{x}_i\}$  that take on values between 0 and 1. The  $i^{\text{th}}$  output  $\hat{x}_i$  is the conditional probability of the  $i^{\text{th}}$  variable being equal to 1 given all variables thus far, that is

$$\hat{x}_i = p(x_i = 1 | \mathbf{x}_{<i}, \boldsymbol{\theta}). \quad (4.35)$$

Hence, given an input vector  $\mathbf{x}$  and using the chain rule, its probability according to NADE can be calculated as

$$p(\mathbf{x} | \boldsymbol{\theta}) = \prod_i \hat{x}_i^{x_i} (1 - \hat{x}_i)^{1-x_i}. \quad (4.36)$$

The outputs of NADE are computed by forward-propagating  $\mathbf{x}$  through the network, one variable at a time. The first output  $\hat{x}_1$  is computed as follows

$$\mathbf{a}_1 = \mathbf{c} \quad (4.37)$$

$$\mathbf{h}_1 = \sigma(\mathbf{a}_1) \quad (4.38)$$

$$\hat{x}_1 = \sigma(\mathbf{u}_1^T \mathbf{h}_1 + b_1). \quad (4.39)$$

where  $\sigma(\cdot)$  is the logistic sigmoid function applied elementwise. From there on, each output  $\hat{x}_i$  (for  $i > 1$ ) is computed in turn by forward-propagating the  $(i-1)^{\text{th}}$  input  $x_{i-1}$  as follows

$$\mathbf{a}_i = \mathbf{a}_{i-1} + \mathbf{w}_{i-1} x_{i-1} \quad (4.40)$$

$$\mathbf{h}_i = \sigma(\mathbf{a}_i) \quad (4.41)$$

$$\hat{x}_i = \sigma(\mathbf{u}_i^T \mathbf{h}_i + b_i). \quad (4.42)$$

In the above equations,  $\{\mathbf{u}_i\}$ ,  $\{b_i\}$ ,  $\{\mathbf{w}_i\}$  and  $\mathbf{c}$  are the adjustable parameters of NADE, which are collectively denoted as  $\boldsymbol{\theta}$ . Also,  $\mathbf{a}_i$  and  $\mathbf{h}_i$  are the hidden layer activations and hidden layer states in the  $i^{\text{th}}$  step.

Notice the special autoregressive structure of NADE; the activation  $\mathbf{a}_i$  of the hidden units in the  $i^{\text{th}}$  step depends on the activation  $\mathbf{a}_{i-1}$  in the step before. Also notice that the last input  $x_I$  is never forward-propagated through NADE, but is only used in computing  $p(\mathbf{x} | \boldsymbol{\theta})$ . Thanks to this autoregressive structure, NADE can compute  $p(\mathbf{x} | \boldsymbol{\theta})$  in time  $\mathcal{O}(IJ)$ ; without it, this computation would require time  $\mathcal{O}(IJ^2)$ . The downside of it is that NADE is sensitive to the ordering of the input variables; that is, NADEs that are trained on the same dataset with different orderings will not necessarily be consistent with each other.

As can be seen from the above description, NADE is fully tractable. That is,  $p(\mathbf{x} | \boldsymbol{\theta})$  can be calculated exactly, in time linear in the size of the input (compare this to the exponential time needed by an RBM). Also, conditioning in NADE becomes trivial, since it directly outputs

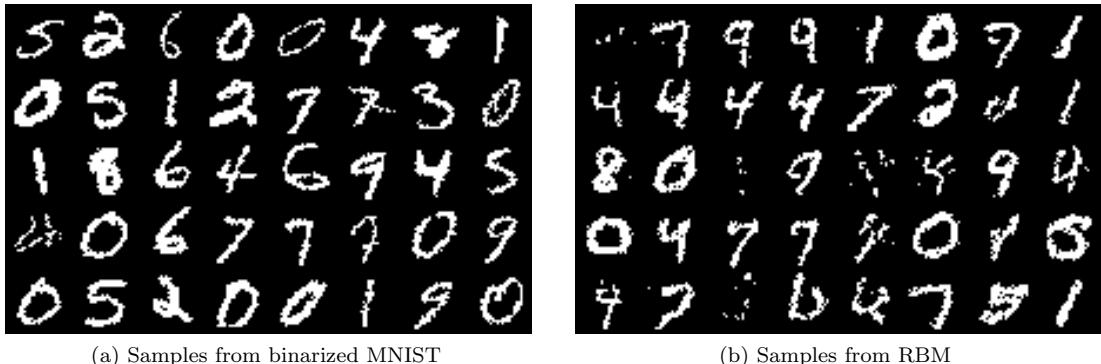


Figure 4.3: Random samples from (a) the binarized version of MNIST and (b) the RBM. The RBM samples were generated by Gibbs sampling; each sample was generated by a separate Markov chain, randomly initialized and burned in until convergence.

conditionals.<sup>2</sup> Generating exact samples from NADE is also straightforward. This can be done using ancestral sampling; generate  $x_1$  from  $\hat{x}_1$ , then forward-propagate it to get  $\hat{x}_2$ , generate  $x_2$  from  $\hat{x}_2$ , then forward-propagate it to get  $\hat{x}_3$ , and so on until the whole vector is generated. Last but not least, due to its neural network structure and the universal approximation theorem for neural networks, NADE can approximate with arbitrary accuracy any desired distribution, as long as it is given enough capacity.<sup>3</sup>

Finally, as discussed in section 4.1.2, in order to train NADE to mimic other generative models, we need to be in a position to calculate

$$\frac{\partial}{\partial \theta} \log p(\mathbf{x} | \theta) \quad \text{and} \quad \mathcal{R} \left\{ \frac{\partial}{\partial \theta} \log p(\mathbf{x} | \theta) \right\}. \quad (4.43)$$

This can be done efficiently (in time linear in the size of the input) by backward propagation and  $\mathcal{R}\{\text{backprop}\}$  respectively. More details on how to do this are given in appendix C.

### 4.2.3 Distilling an RBM into NADE

In this section, we present the experimental results of distilling an intractable RBM into a tractable NADE. Our goal is to assess how successful the distillation is, and to what extent NADE manages to capture the knowledge of the RBM.

#### The RBM

In all our experiments, we used an RBM that was trained on the MNIST dataset of handwritten digits (Le Cun et al.). The RBM was provided by Salakhutdinov and Murray (2008), who referred to it as CD25(500).<sup>4</sup> This particular RBM has 500 hidden variables and it was trained using Contrastive Divergence (Hinton, 2002) with 25 Gibbs steps for each gradient evaluation.

The RBM was trained to be a generative model of MNIST digits. The MNIST dataset consists of greyscale images of handwritten digits (0 to 9), of size  $28 \times 28$ . The dataset is partitioned into a train set of 60,000 images and a test set of 10,000 images. The images were binarized before training; each pixel was independently set to 1 with probability equal to the original pixel intensity (which varies from 0 to 1). For consistency, in all our experiments below, we use the same binarized version of MNIST.

<sup>2</sup>Note however that if a different conditional from those output by NADE is needed, then the only obvious way to obtain it would be to retrain NADE from scratch with a convenient ordering.

<sup>3</sup>Note however that this might mean that the number of hidden units has to become prohibitively large.

<sup>4</sup>It was downloaded as is from [http://www.cs.toronto.edu/~rsalakhu/rbm\\_ais.html](http://www.cs.toronto.edu/~rsalakhu/rbm_ais.html) (accessed on 20 July 2015).

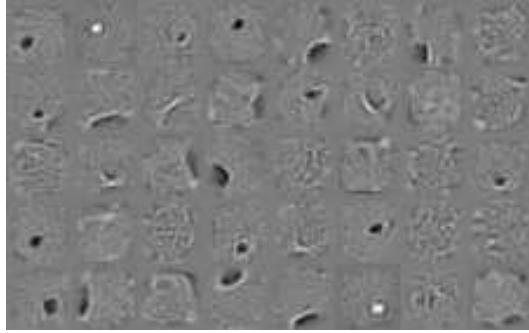


Figure 4.4: Features learnt by the RBM trained on MNIST. The features are the columns of the  $\mathbf{W}$  matrix of the RBM, reshaped as  $28 \times 28$  images.

In their paper, Salakhutdinov and Murray (2008) argue that this particular RBM is indeed a good distribution estimator for MNIST. They also use annealed importance sampling (Neal, 2001) to estimate the partition function of the RBM. Figure 4.3 shows random samples from both the binarized MNIST dataset and the RBM; their similarity indicates that the RBM is indeed a good distribution estimator of MNIST. Figure 4.4 shows some of the features the RBM has learnt.

### Distillation details

In order to assess how much capacity NADE should have, we used NADEs with 250, 500, 750 and 1000 hidden units. All NADEs used a raster ordering of variables; that is, the visible variables (image pixels) were ordered columnwise. Empirically, we found that using a raster ordering as opposed to a random ordering is more suitable for MNIST.

The loss functions we used are *KL divergence* and *square error* (see section 4.1.1). We did not use score matching, since—as we argued in the relevant section—score matching is only suitable for continuous distributions. The square error loss function has a free parameter  $c$  that needs to be set. In our relevant analysis, we argued that any value of  $c$  that is less or equal to  $\max_{\mathbf{x}} \log \bar{p}(\mathbf{x})$  should do. In the experiments below, we found a large value of  $\log \bar{p}(\mathbf{x})$  and set  $c$  to it. We used random sampling followed by hill climbing to determine a large value for  $\log \bar{p}(\mathbf{x})$ . The largest value we found was 436.49 which, incidentally, corresponds to the all-black image.<sup>5</sup> In all our experiments, we used this value for  $c$ .

During stochastic gradient training, we used minibatches of 20 samples from the RBM. The samples were generated using block Gibbs sampling with parallel chains. In particular, we maintained 2000 parallel Markov chains and, in each iteration, we simulated each chain once and selected 20 of them (in sequence) to form the minibatch. This way, all samples within a minibatch are independent (since they come from independently simulated chains) and each chain is thinned  $2000/20 = 100$  times before contributing a sample. Empirically, we found that both having independent samples within the minibatch and thinning across minibatches improved learning. Note that this sampling scheme is also efficient to run, since all chains can be updated in parallel. Also, all chains were randomly initialized and burned in for 2000 iterations before they actually started contributing samples.

In order to determine the learning rate in each stochastic update, we used ADADELTA (Zeiler, 2012), which is an adaptive learning rate method that uses a different learning rate for each parameter. ADADELTA is controlled by two hyperparameters, for which we used the default values suggested in the original paper. It comes with no convergence guarantees, but we have found that it works well in practice, while it requires minimal tuning. Finally, we chose to run

---

<sup>5</sup>Note that this is not guaranteed to be the maximum of  $\log \bar{p}(\mathbf{x})$ , as the maximization method we used is only approximate. For a general RBM, finding the exact maximum is intractable. However, any value less than the maximum will do for our purposes.

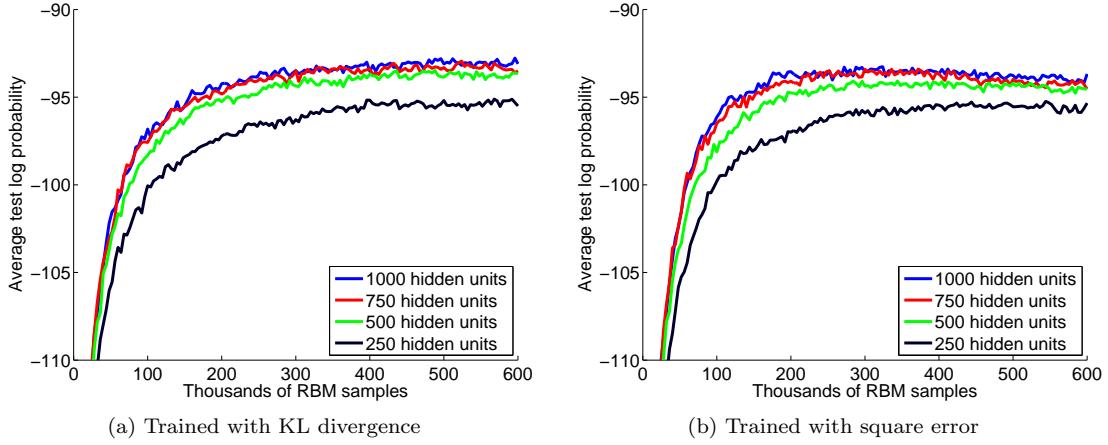


Figure 4.5: Training progress for each NADE, as measured by average log probability of the first 500 images of the MNIST test set. The training progress was measured every 200 iterations, that is, every  $200 \times 20 = 4000$  RBM samples.

| Hidden units | KL divergence     | Square error      |
|--------------|-------------------|-------------------|
| 250          | $-94.75 \pm 0.78$ | $-94.62 \pm 0.78$ |
| 500          | $-93.19 \pm 0.76$ | $-93.83 \pm 0.77$ |
| 750          | $-92.93 \pm 0.76$ | $-93.61 \pm 0.77$ |
| 1000         | $-92.49 \pm 0.76$ | $-93.01 \pm 0.76$ |

Table 4.1: Average log probability each NADE assigns to the full MNIST test set, at the end of training. Error bars correspond to 3 standard deviations.

stochastic gradient training for 30,000 iterations, producing in total  $30,000 \times 20 = 600,000$  RBM samples.

## Results and discussion

We trained 4 NADEs with 250, 500, 750 and 1000 hidden units, using KL divergence and square error; this resulted in 8 models in total. Figure 4.5 shows the progress of each NADE during training. The plots track the average log probability each NADE assigns to the first 500 MNIST test images, as this evolves during training. Table 4.1 shows the average log probability each NADE assigns to the full MNIST test set at the end of training. We can see that, in general, more hidden units lead to a higher log probability, with the difference being significant between the NADEs with 250 and 500 hidden units. This suggests that at least 500 hidden units are needed for NADE to have enough flexibility to capture the knowledge in the RBM (it is no coincidence that the RBM has 500 hidden variables as well).

Figure 4.6 visualizes the agreement between each NADE and the RBM. To get the plots, the log probability of each of the first 500 images of the MNIST test set was evaluated under each NADE and the RBM. Then, the images were sorted in descending order, according to the log probability the RBM assigns to them. The plots show the log probabilities of each NADE superimposed on the log probabilities of the RBM (the RBM curve is the same for all plots). Note that the true log probability of the RBM is actually intractable, as the partition function is needed to evaluate it. For the purposes of these plots, we set  $\log Z = 451.28$ , which is the value estimated by Salakhutdinov and Murray (2008). We can see that in general the shape of the NADE curves match that of the RBM curve, which suggests that a fair amount of the RBM

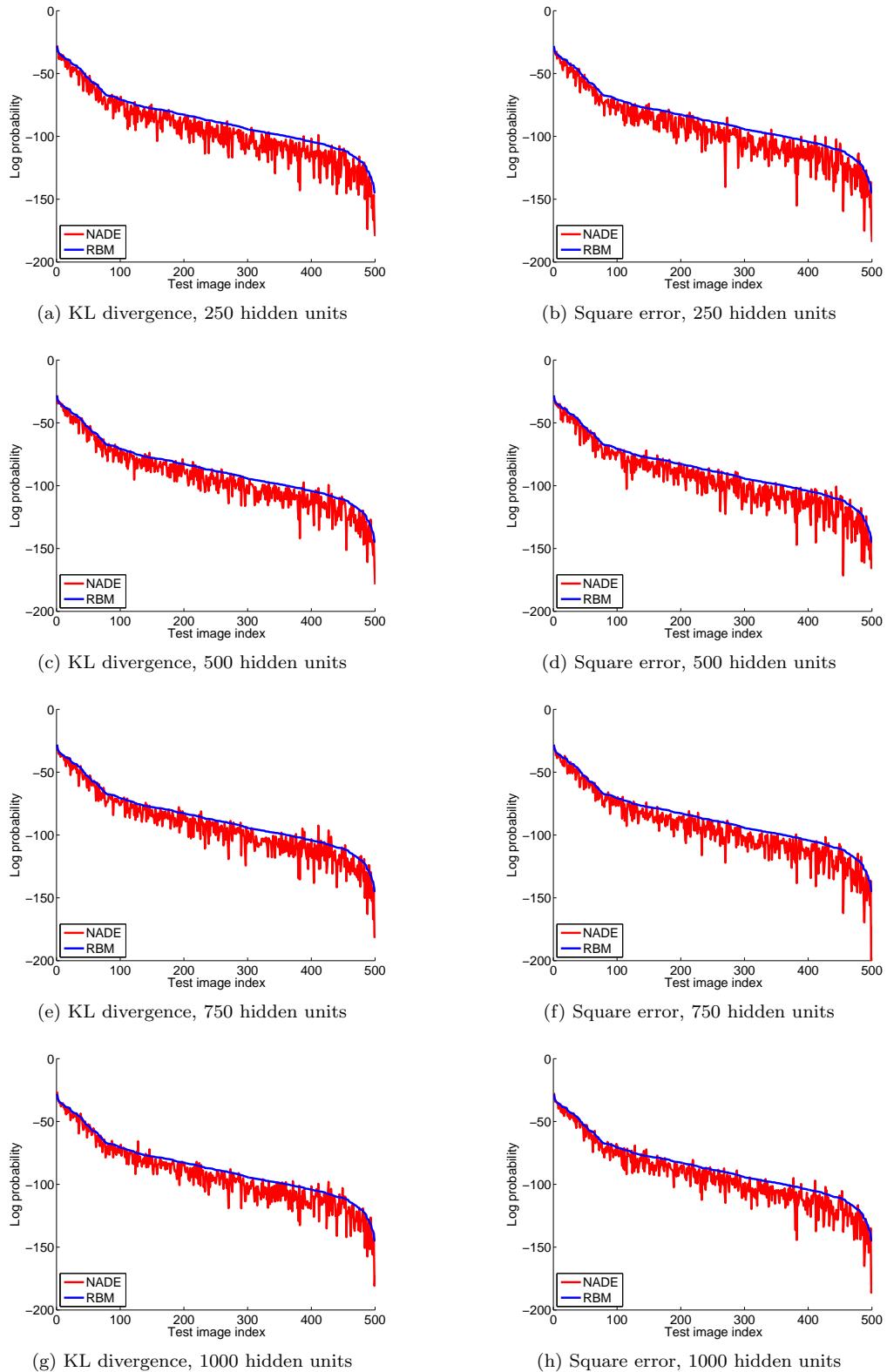
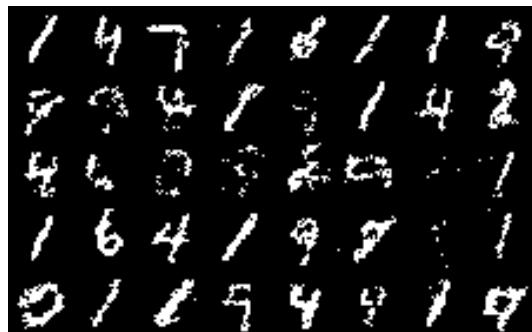
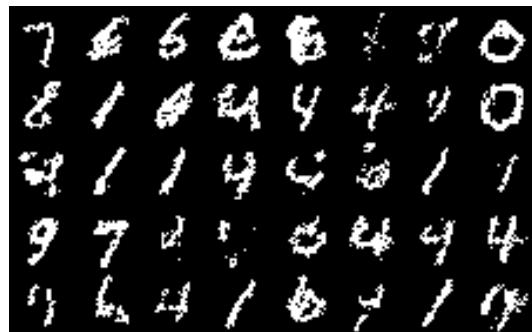


Figure 4.6: Log probability each trained NADE assigns to the first 500 images of the MNIST test set, compared to the RBM. Note that the log probability of the RBM was calculated by setting  $\log Z = 451.28$ , as estimated by Salakhutdinov and Murray (2008).



(a) KL divergence, 250 hidden units



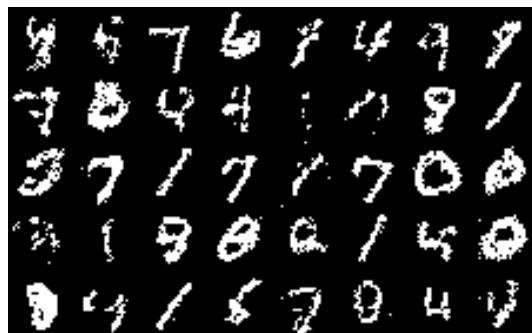
(b) Square error, 250 hidden units



(c) KL divergence, 500 hidden units



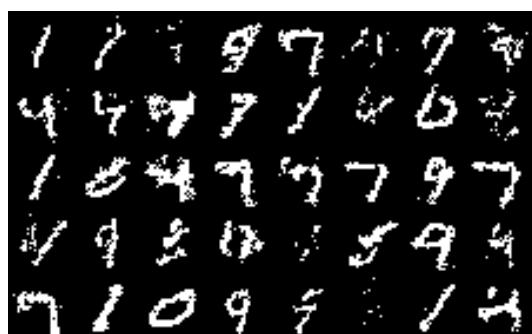
(d) Square error, 500 hidden units



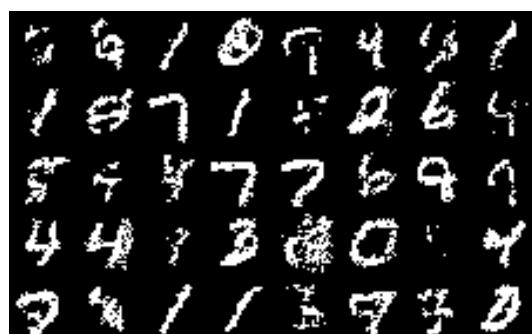
(e) KL divergence, 750 hidden units



(f) Square error, 750 hidden units



(g) KL divergence, 1000 hidden units



(h) Square error, 1000 hidden units

Figure 4.7: Random samples from each trained NADE. All samples are exact and were generated using ancestral sampling.

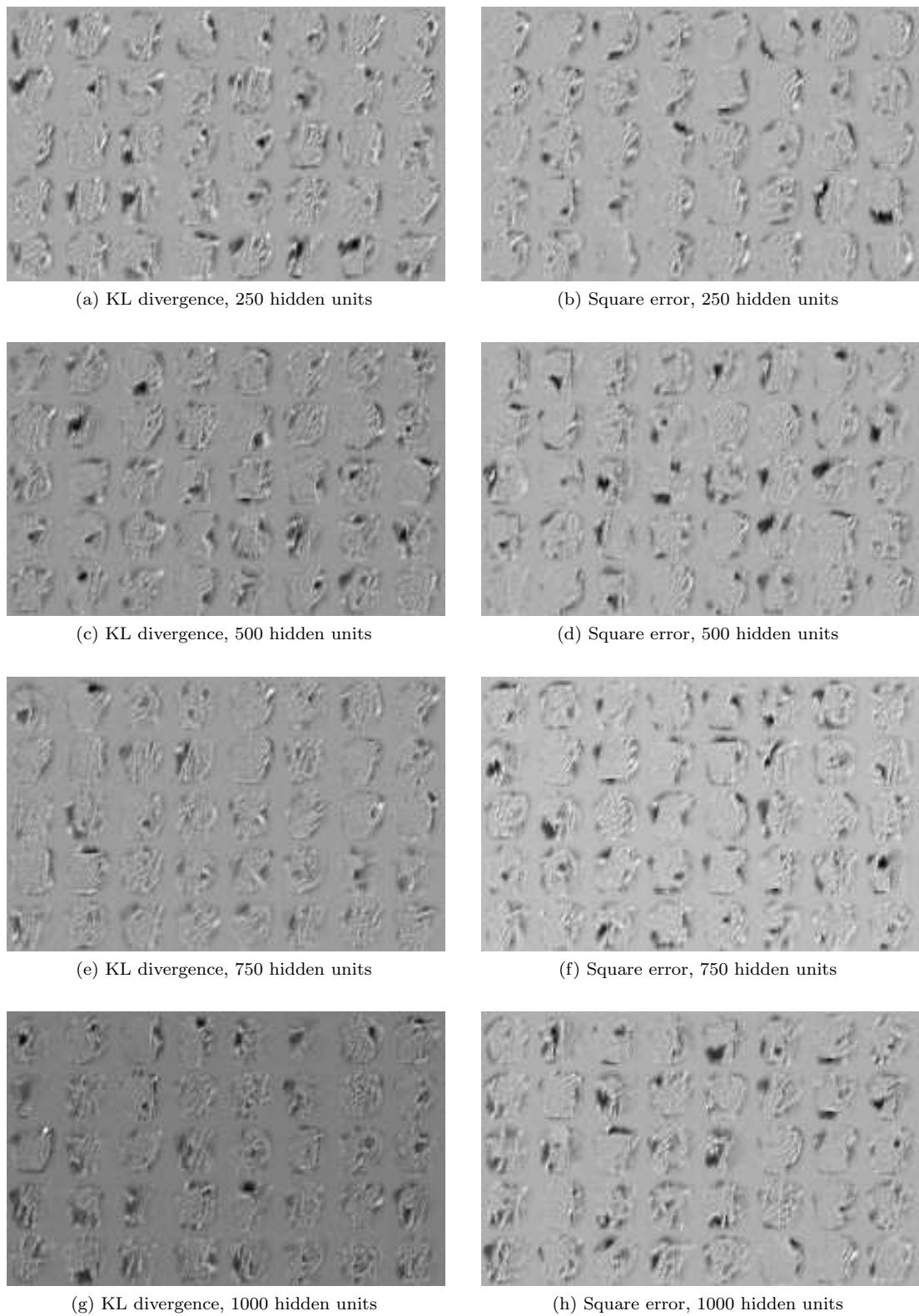


Figure 4.8: Features learnt by the trained NADEs. The features are the columns of the matrix that has  $\{\mathbf{w}_i\}$  as rows, reshaped as  $28 \times 28$  images.

knowledge was distilled into the NADEs. However, the NADEs seem to underestimate the log probability assigned by the RBM. This is actually expected, since minimizing  $D_{\text{KL}}(p(\mathbf{x}) \parallel p(\mathbf{x} | \boldsymbol{\theta}))$  encourages  $p(\mathbf{x} | \boldsymbol{\theta})$  to be broader than  $p(\mathbf{x})$ , so as to prevent NADE from not having mass in regions where the RBM has mass. Also, for certain images, the log probabilities from the RBM and the NADEs differ by several nats, which shows that the distillation was not perfect. However, in interpreting these results, one should bear in mind that the log probability of the RBM in these plots is only an estimate (albeit a good one), so the conclusions drawn from them are only as good as this estimate.

Figure 4.7 shows samples generated from each trained NADE. These samples are exact, and they were generated using ancestral sampling. It is evident that there is a strong resemblance between samples from NADE and actual images of digits, which shows that the NADEs have managed to learn a lot from the RBM. To see if the knowledge distillation was successful, this figure should be contrasted with Figure 4.3, which shows samples from the binarized MNIST dataset and the RBM. Obviously, it is easy to discriminate between real digits and samples from the RBM and the NADEs, which indicates that not even the RBM has learnt a perfect distribution estimator for MNIST. However, it is not so easy (at least visually) to discriminate between samples from the RBM and samples from the NADEs, suggesting that knowledge distillation has done a fairly good job. Finally, Figure 4.8 shows some features learnt by each NADE.

From the above results, we can see that both KL divergence and square error were successful in distilling the RBM into NADE. However, it appears that KL divergence performs slightly better than square error. In Figure 4.5, we can see that the performance of KL divergence continues to increase with the number of samples, whereas square error seems to reach a plateau fairly quickly, after which the performance shows signs of declining. This is also reflected in the results of Table 4.1, where KL divergence is shown to achieve a better performance at the end of training—although not significantly better. On the other hand, in the beginning of training, NADE appears to learn faster with square error. This is due to the fact that square error uses the unnormalized probabilities of the RBM in addition to sampling from it, therefore it transfers more information about the RBM to NADE. In contrast, KL divergence relies only on samples to transfer information from the RBM to NADE. This suggests that a potentially good strategy is to start training with square error to benefit from fast training, and after a few iterations switch to KL divergence to take advantage of the continuing improvement in performance.

Note that we also experimented with score matching. However, we found that score matching fails to train NADE satisfactorily. This is to be expected, since score matching does not work with discrete probability distributions, and it is only meant to be used with continuous probability distributions, as we have discussed in section 4.1.1.

#### 4.2.4 Estimating the partition function of the RBM

Having access to a tractable generative model that closely mimics an intractable generative model provides a significant opportunity for estimating intractable quantities of the latter. To demonstrate this, in this section we will show how to use the trained NADEs of the previous section in order to obtain good estimates of the partition function of the RBM.

Estimating the partition function of an intractable RBM was also done by Salakhutdinov and Murray (2008), who used annealed importance sampling for this purpose. Their method involved setting up a chain of 10,000 intermediate distributions between the RBM and a tractable base distribution. Here, we will show that, having access to a mimicking NADE, we can achieve estimates which are as good as theirs with simpler sampling methods.

##### Bounds based on KL divergence

Training a NADE to mimic the RBM was done by minimizing either the KL divergence from RBM to NADE, or the square error between their log probabilities, which, as we argued, achieves a similar effect. Directly estimating the KL divergence between the trained NADE and the RBM

| Hidden units | KL divergence     | Square error      |
|--------------|-------------------|-------------------|
| <b>250</b>   | $459.01 \pm 0.20$ | $459.71 \pm 0.21$ |
| <b>500</b>   | $458.20 \pm 0.18$ | $458.87 \pm 0.19$ |
| <b>750</b>   | $457.66 \pm 0.17$ | $458.72 \pm 0.19$ |
| <b>1000</b>  | $457.70 \pm 0.17$ | $458.52 \pm 0.19$ |

Table 4.2: Approximate upper bounds of  $\log Z$  given by estimating  $D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x} | \boldsymbol{\theta}))$  for each NADE. The lower this value is, the better NADE has fitted the RBM. Error bars correspond to 3 standard deviations.

| Hidden units | KL divergence     | Square error      |
|--------------|-------------------|-------------------|
| <b>250</b>   | $442.80 \pm 0.26$ | $439.21 \pm 0.39$ |
| <b>500</b>   | $442.98 \pm 0.27$ | $440.85 \pm 0.34$ |
| <b>750</b>   | $443.41 \pm 0.25$ | $439.19 \pm 0.45$ |
| <b>1000</b>  | $443.74 \pm 0.24$ | $438.11 \pm 0.51$ |

Table 4.3: Approximate lower bounds of  $\log Z$  given by estimating  $D_{\text{KL}}(p(\mathbf{x} | \boldsymbol{\theta}) \| p(\mathbf{x}))$  for each NADE. The higher this value is, the better NADE has fitted the RBM. Error bars correspond to 3 standard deviations.

provides a good upper bound to the log partition function. To see why, write the KL as follows

$$\begin{aligned} D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x} | \boldsymbol{\theta})) &= \langle \log p(\mathbf{x}) - \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} \\ &= \langle \log \bar{p}(\mathbf{x}) - \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} - \log Z. \end{aligned} \quad (4.44)$$

Due to the non-negativity of the KL divergence, we can upper bound  $\log Z$  as follows

$$\log Z \leq \langle \log \bar{p}(\mathbf{x}) - \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}. \quad (4.45)$$

The upper bound on the right hand side can be easily estimated by Monte Carlo as follows

$$\langle \log \bar{p}(\mathbf{x}) - \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x})} \approx \frac{1}{S} \sum_s (\log \bar{p}(\mathbf{x}_s) - \log p(\mathbf{x}_s | \boldsymbol{\theta})) \quad \text{where } \mathbf{x}_s \sim p(\mathbf{x}). \quad (4.46)$$

Since the objective during training was to minimize  $D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x} | \boldsymbol{\theta}))$ , this upper bound is optimized to be as tight as possible. Furthermore, its tightness indicates how faithfully NADE mimics the RBM.

We can use the same technique to calculate approximate lower bounds for  $\log Z$  by estimating the KL divergence the other way round, from NADE to RBM. We rewrite this KL divergence as follows

$$\begin{aligned} D_{\text{KL}}(p(\mathbf{x} | \boldsymbol{\theta}) \| p(\mathbf{x})) &= \langle \log p(\mathbf{x} | \boldsymbol{\theta}) - \log p(\mathbf{x}) \rangle_{p(\mathbf{x} | \boldsymbol{\theta})} \\ &= \langle \log p(\mathbf{x} | \boldsymbol{\theta}) - \log \bar{p}(\mathbf{x}) \rangle_{p(\mathbf{x} | \boldsymbol{\theta})} + \log Z, \end{aligned} \quad (4.47)$$

and by non-negativity we get the following lower bound

$$\langle \log \bar{p}(\mathbf{x}) - \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | \boldsymbol{\theta})} \leq \log Z, \quad (4.48)$$

which can be easily estimated by Monte Carlo as follows

$$\langle \log \bar{p}(\mathbf{x}) - \log p(\mathbf{x} | \boldsymbol{\theta}) \rangle_{p(\mathbf{x} | \boldsymbol{\theta})} \approx \frac{1}{S} \sum_s (\log \bar{p}(\mathbf{x}_s) - \log p(\mathbf{x}_s | \boldsymbol{\theta})) \quad \text{where } \mathbf{x}_s \sim p(\mathbf{x} | \boldsymbol{\theta}). \quad (4.49)$$

The higher this lower bound is, the better NADE has fitted the RBM. However, unlike the upper bound we calculated earlier, we expect this lower bound to be somewhat loose, since it was not optimized directly during training.

Tables 4.2 and 4.3 show the estimated upper and lower bounds respectively given by each NADE. We used 10,000 samples for each estimate. Samples from the RBM were obtained by running 10,000 parallel Gibbs chains, randomly initialized and burned in for 2000 iterations. Judging from the tightness of the bounds, we can see that NADE fits the RBM better with KL divergence, a result we also observed in the experiments of section 4.2.3. Also, the fit improves with the number of hidden units, which is to be expected since more hidden units add extra flexibility. We can thus conclude that the best models are the NADEs with 750 and 1000 hidden units, trained with KL divergence.

For comparison,  $\log Z$  was estimated by Salakhutdinov and Murray (2008) to be 451.28 nats, which, as expected, is closer to the estimated upper bounds than the estimated lower bounds. Note that we can always trivially lower bound  $\log Z$ , since

$$\max_{\mathbf{x}} \log \bar{p}(\mathbf{x}) \leq \log Z. \quad (4.50)$$

By approximately maximizing  $\log \bar{p}(\mathbf{x})$  using random sampling followed by hill climbing, we obtained a lower bound equal to 436.49 nats. The lower bounds in Table 4.3 are all better than this.

### Importance sampling

Importance sampling (MacKay, 2002, section 29.2) can be used in order to estimate the partition function. Given a tractable proposal distribution  $q(\mathbf{x})$  that is non-zero wherever  $p(\mathbf{x})$  is non-zero, importance sampling is based on rewriting the partition function as an expectation over  $q(\mathbf{x})$  as follows

$$Z = \sum_{\mathbf{x}} \bar{p}(\mathbf{x}) = \sum_{\mathbf{x}} \frac{\bar{p}(\mathbf{x})}{q(\mathbf{x})} q(\mathbf{x}) = \left\langle \frac{\bar{p}(\mathbf{x})}{q(\mathbf{x})} \right\rangle_{q(\mathbf{x})}. \quad (4.51)$$

Hence,  $Z$  can be estimated by Monte Carlo as follows

$$Z \approx \frac{1}{S} \sum_s \frac{\bar{p}(\mathbf{x}_s)}{q(\mathbf{x}_s)} \quad \text{where } \mathbf{x}_s \sim q(\mathbf{x}_s). \quad (4.52)$$

The above estimate is only as good as the proposal distribution. To work well,  $q(\mathbf{x})$  must be as similar to  $p(\mathbf{x})$  as possible. If there is significant discrepancy between the two distributions, the above estimator has high variance.

In our framework, NADEs that were trained by minimizing  $D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x} | \boldsymbol{\theta}))$  will tend to mimic closely the RBM and at the same time avoid at all costs not putting mass where the RBM has mass. Thanks to these properties, such NADEs can serve as effective proposal distributions for importance sampling. In this section, we demonstrate that proposing from a trained NADE can make a method as simple as importance sampling work decently.

Table 4.4 shows the estimates of  $\log Z$  obtained by using each trained NADE as proposal distribution. We used 10,000 exact samples from NADE for each estimate. In general, the estimates agree across NADEs. Salakhutdinov and Murray (2008) report an estimate of 451.28, with a 3 standard deviation confidence interval of (450.97, 451.52). Compared to this, it appears that importance sampling with NADE slightly underestimates  $\log Z$ . Still, it is interesting that a not too dissimilar result can be obtained just by using simple importance sampling.

### Bridge sampling

Previously, we argued that the trained NADEs are excellent proposal distributions for estimating the partition function of the RBM. However, importance sampling is still a poor method. A better method, with the same proposal distribution, should give better results.

| Hidden units | KL divergence           | Square error            |
|--------------|-------------------------|-------------------------|
| <b>250</b>   | 450.86 (450.06, 451.30) | 450.81 (449.77, 451.31) |
| <b>500</b>   | 450.96 (449.94, 451.46) | 450.96 (450.33, 451.34) |
| <b>750</b>   | 450.79 (450.47, 451.04) | 450.95 (450.40, 451.30) |
| <b>1000</b>  | 450.79 (450.40, 451.07) | 451.51 (450.09, 452.08) |

Table 4.4: Estimates of RBM’s log partition function based on importance sampling, using each trained NADE as proposal distribution. Brackets show confidence intervals that correspond to 3 standard deviations.

Importance sampling tends to fail when the proposal distribution has little mass in some regions where the target distribution has significant mass. Bridge sampling (Neal, 2005, section 1) is an improvement over importance sampling that tries to alleviate this problem. Bridge sampling uses a “bridge distribution”  $p_b(\mathbf{x}) = \frac{1}{Z_b} \bar{p}_b(\mathbf{x})$ , which is constructed to be the overlap between  $p(\mathbf{x})$  and  $q(\mathbf{x})$ . Then, using standard importance sampling, the following two quantities are estimated

$$Z_b = \left\langle \frac{\bar{p}_b(\mathbf{x})}{q(\mathbf{x})} \right\rangle_{q(\mathbf{x})} \quad \text{and} \quad \frac{Z_b}{Z} = \left\langle \frac{\bar{p}_b(\mathbf{x})}{\bar{p}(\mathbf{x})} \right\rangle_{p(\mathbf{x})}. \quad (4.53)$$

Notice that, by construction,  $p_b(\mathbf{x})$  does not have regions with significant mass that do not also have enough mass under  $p(\mathbf{x})$  and  $q(\mathbf{x})$ , hence importance sampling works well in this case. Having estimated the above quantities,  $Z$  can be trivially estimated by their ratio.

When the same number of importance samples has been used from both  $p(\mathbf{x})$  and  $q(\mathbf{x})$ , the asymptotically optimal choice of bridge distribution (Meng and Wong, 1996) is the following

$$\bar{p}_b(\mathbf{x}) = \frac{q(\mathbf{x})\bar{p}(\mathbf{x})}{Zq(\mathbf{x}) + \bar{p}(\mathbf{x})}. \quad (4.54)$$

In the above, the value of  $Z$  is needed, which is what the method is trying to estimate. Thus, the method can be set up as a fixed point system; initialize  $Z$  (e.g. to 1), calculate  $\bar{p}_b(\mathbf{x})$ , use it to estimate  $Z$ , and repeat until convergence. In our experiments, we found that in practice only few iterations are necessary.

Following the above description of bridge sampling, we used it together with the trained NADE as proposal distribution. Our algorithm is described below.

- (i) Generate  $S$  exact samples from NADE.
- (ii) Run  $S$  parallel Gibbs chains, initialized with the NADE samples, to generate  $S$  independent RBM samples.
- (iii) Initialize  $Z = 1$ .
- (iv) Iterate between calculating  $\bar{p}_b(\mathbf{x})$  and calculating  $Z$ . Reuse the samples collected in step (i) and step (ii) throughout.

Note that in the above algorithm we take advantage of the fact that NADE samples are similar to RBM samples, and we initialize the RBM Gibbs chains with them. This way we ensure we obtain good quality samples from the RBM. In our experiments, we used  $S = 10,000$  samples and we burned in the RBM Gibbs chains for 2000 iterations. Also, we iterated step (iv) 10 times.

Table 4.5 shows the estimates of  $\log Z$  using the above bridge sampling procedure, for each trained NADE. We can see that all estimates are consistent with each other and have small

| Hidden units | KL divergence           | Square error            |
|--------------|-------------------------|-------------------------|
| <b>250</b>   | 451.27 (451.14, 451.39) | 451.24 (451.10, 451.38) |
| <b>500</b>   | 451.28 (451.16, 451.39) | 451.30 (451.17, 451.43) |
| <b>750</b>   | 451.22 (451.11, 451.33) | 451.21 (451.08, 451.33) |
| <b>1000</b>  | 451.24 (451.13, 451.35) | 451.36 (451.23, 451.49) |

Table 4.5: Estimates of RBM’s log partition function based on bridge sampling, using each trained NADE as proposal distribution. Brackets show confidence intervals that correspond to 3 standard deviations.

variance. This suggests that bridge sampling provides more accurate and more reliable estimates of  $\log Z$  compared to importance sampling. For comparison, Salakhutdinov and Murray (2008), using annealed importance sampling, report an estimate of 451.28, with a 3 standard deviation confidence interval of (450.97, 451.52). The fact that our results are in close agreement with theirs, even though they were obtained by different means, is strong evidence for their correctness.<sup>6</sup>

### 4.3 Related work and extensions

In this section, we review two general techniques that are related to knowledge distillation in generative models. These are *variational inference* and *generative adversarial training*. Incorporating these techniques would be an interesting extension to the knowledge distillation framework presented in this chapter.

#### 4.3.1 Variational inference

Variational inference originated as a method for approximate Bayesian inference and has its roots in statistical physics. In Bayesian inference, predictions are made by taking expectations over posterior distributions that are often intractable. The idea behind variational inference is to replace the intractable posterior distribution with a simpler one, over which expectations become tractable.

Let  $p(\mathbf{x}) = \frac{1}{Z} \bar{p}(\mathbf{x})$  be the intractable distribution of interest and  $p(\mathbf{x}|\boldsymbol{\theta})$  be a tractable distribution, parameterized by  $\boldsymbol{\theta}$ . Variational inference approximates  $p(\mathbf{x})$  with the particular  $p(\mathbf{x}|\boldsymbol{\theta})$  that minimizes  $D_{\text{KL}}(p(\mathbf{x}|\boldsymbol{\theta}) \| p(\mathbf{x}))$ . If  $p(\mathbf{x}|\boldsymbol{\theta})$  is chosen such that expectations over it are analytically tractable, then minimizing the above KL divergence is particularly convenient. To see why, rewrite the KL divergence as follows

$$D_{\text{KL}}(p(\mathbf{x}|\boldsymbol{\theta}) \| p(\mathbf{x})) = -F(\boldsymbol{\theta}) + \log Z, \quad (4.55)$$

where

$$F(\boldsymbol{\theta}) = \langle \log \bar{p}(\mathbf{x}) \rangle_{p(\mathbf{x}|\boldsymbol{\theta})} - \langle \log p(\mathbf{x}|\boldsymbol{\theta}) \rangle_{p(\mathbf{x}|\boldsymbol{\theta})}. \quad (4.56)$$

The above quantity  $F(\boldsymbol{\theta})$  is tractable, because it involves expectations of tractable quantities over  $p(\mathbf{x}|\boldsymbol{\theta})$ , and is known as *variational free energy*. Minimizing the KL divergence is equivalent to maximizing the variational free energy. Due to the non-negativity of the KL divergence, it is easy to see that  $F(\boldsymbol{\theta})$  is a lower bound on  $\log Z$ .

Variational inference is closely related to knowledge distillation with KL divergence (as described in section 4.1.1). Their difference is that the former minimizes  $D_{\text{KL}}(p(\mathbf{x}|\boldsymbol{\theta}) \| p(\mathbf{x}))$  whereas the latter minimizes  $D_{\text{KL}}(p(\mathbf{x}) \| p(\mathbf{x}|\boldsymbol{\theta}))$ . Nevertheless, despite their conceptual similarity, these two techniques are computationally different, and have different requirements regarding the choice of  $p(\mathbf{x}|\boldsymbol{\theta})$ . Variational inference typically requires the expectations over  $p(\mathbf{x}|\boldsymbol{\theta})$  to

<sup>6</sup>Note however that consistency does not necessarily imply correctness. It could still be the case that both our estimates and those of Salakhutdinov and Murray (2008) are not correct.

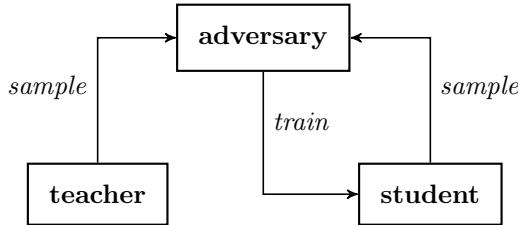


Figure 4.9: Schematic representation of generative adversarial training. The student is trained to fool the adversary, whereas the adversary is trained to discriminate between the teacher and the student.

be analytically tractable, which limits the choice of  $p(\mathbf{x} | \boldsymbol{\theta})$  severely. For instance, expectations under NADE are not analytically tractable, and therefore the variational framework is not directly applicable to distilling RBMs into NADEs. On the other hand, knowledge distillation only requires  $p(\mathbf{x} | \boldsymbol{\theta})$  to be differentiable, which is a much weaker requirement. Making variational inference work well with models like NADE is a future research direction.

For more information and details, variational inference and its applications are discussed in length by Bishop (2006, chapter 10) and Murphy (2012, chapter 21).

### 4.3.2 Generative adversarial training

If two generative models are equivalent, then, given a sample from one of them, it would be hard to tell which model it came from. Generative adversarial training (Goodfellow et al., 2014) takes this idea and turns it into a training objective, in order to train a student generative model to mimic a teacher generative model.

Figure 4.9 provides a schematic representation of generative adversarial training. The setup includes a teacher generative model  $p(\mathbf{x})$ , a student generative model  $p(\mathbf{x} | \boldsymbol{\theta})$ , parameterized by  $\boldsymbol{\theta}$ , and an adversary  $f(\mathbf{x} | \phi)$ , parameterized by  $\phi$ . Given a sample  $\mathbf{x}$ , the job of the adversary is to output how likely it thinks it is that  $\mathbf{x}$  came from the student or the teacher. During training, both the teacher and the student generate samples, and the adversary tries to predict their model of origin. The student and the adversary are trained simultaneously; the training objective for the student is to fool the adversary, and the training objective for the adversary is to improve so as not to be fooled. As this min-max game between the student and the adversary progresses, the student adapts in order to generate samples that look like those of the teacher, following the training signal from the adversary.

The main advantage of generative adversarial training is that it requires very little from  $p(\mathbf{x})$  and  $p(\mathbf{x} | \boldsymbol{\theta})$  and thus exhibits broad applicability. There are only two main requirements; (a) it must be possible to sample from both  $p(\mathbf{x})$  and  $p(\mathbf{x} | \boldsymbol{\theta})$ , and (b)  $p(\mathbf{x} | \boldsymbol{\theta})$  must be differentiable. The downside is that it can be tricky to get it to work well; there are critical decisions to be made regarding how much capacity to allow to the adversary and how long the adversary should be trained for before an update to the student is made. Also, in case we can evaluate  $p(\mathbf{x})$  and  $p(\mathbf{x} | \boldsymbol{\theta})$ , it is not clear whether the adversarial framework is any better than directly trying to minimize a discrepancy measure between  $p(\mathbf{x})$  and  $p(\mathbf{x} | \boldsymbol{\theta})$ . In any case, incorporating generative adversarial training is an interesting future direction, as it would add flexibility and generality to the knowledge distillation framework.

## 4.4 Summary and conclusions

In this chapter we presented a framework for distilling intractable generative models to tractable ones. We managed to achieve this by using loss functions that circumvent the intractable partition function, while at the same time capture as much information about the intractable model as possible.

We validated our framework by successfully distilling an intractable RBM to a tractable NADE. As an application of the distillation, we showed that the distilled NADE can be successfully used to estimate the intractable partition function of the RBM, using simple sampling methods that would not work well otherwise. This constitutes a novel method of estimating the partition function, and, to the best of our knowledge, this thesis is the first in the literature to describe it.

We concluded the chapter by examining two approaches relevant to knowledge distillation, namely variational inference and generative adversarial training. In future work, these two approaches can be incorporated into our framework, extending its flexibility and applicability.

Even though in principle the framework can be used as much for continuous as for discrete probability distributions, our evaluation focused on the latter kind. Due to this, score matching, even though it was included and implemented in our framework, was left out of the evaluation, since it only works with continuous probability distributions. Future work is needed in order to validate the framework and score matching in particular in the case of continuous distributions as well.

# Chapter 5

## Conclusions

Our goal in this thesis was to show how we could take a cumbersome but accurate model, and replace it with a convenient model that does the same job just as well. Imagining we could extract the knowledge hidden inside the cumbersome model, distil it, and inject it into the convenient model, we referred to the process of constructing the convenient model from the cumbersome one as knowledge distillation.

Our approach to knowledge distillation was to train the convenient model to mimic the cumbersome one, by making it observe the behaviour of the latter and penalizing it whenever it failed to reproduce it accurately. We applied this approach to three distinct and fairly general fields of machine learning: discriminative models, Bayesian inference and generative models. For each field, we presented a knowledge distillation framework, and successfully applied it to an example problem within the field.

Our work brought together ideas often referred to in the literature as knowledge distillation, model compression, compact approximations, mimicking models and teacher-student models. Our view in this thesis was that all these different terms refer to similar concepts and ideas. By viewing them as different manifestations of knowledge distillation, we were able to treat them in a unifying manner.

We contributed to the literature of knowledge distillation with three main novel ideas. Firstly, we introduced the concept of online distillation in the context of compact predictive distributions (chapter 3). With online distillation, we were able to significantly reduce the memory requirements of existing methods. Secondly, we proposed matching derivatives as part of the distillation process. We showed that matching derivatives can be done for a constant extra cost, and provided an efficient implementation of it. We argued that derivatives can convey more information about the model we wish to distil, justifying the extra cost when data is scarce. Thirdly, in the context of generative models (chapter 4), we demonstrated that the distilled model can be used for estimating intractable quantities of the original model, and we successfully applied this idea in estimating the partition function of an intractable RBM.

### 5.1 Knowledge distillation: is it worth it?

Knowledge distillation relies on the premise that the convenient model is capable of doing the job of the cumbersome model just as well. However, in the introduction (chapter 1) we argued that the cumbersome model often owes its success to those characteristics that make it cumbersome, such as its size and its complexity. Then, how can we expect a simple model to be able to perform just as well? On the other hand, if a simple model does indeed perform just as well, then why take the trouble to train the cumbersome model at all? Why not simply train and use the convenient model in the first place?

It is true indeed that a simple model will typically have less capacity than a larger, more complicated model. For instance, in our case study in chapter 2, where we distilled an ensemble of neural networks into a single small neural network, we observed that the single network performed

less well than the ensemble. Our goal there was to perform the distillation in such a way as to make this decrease in performance as small as possible. Moreover, in our case study in chapter 4, where an RBM was distilled into a NADE, we found that a NADE with at least the same capacity as the RBM was needed for accurately mimicking it. In cases such as the above, distillation makes sense as long as the decrease in performance is counterbalanced by the benefits offered by the convenient model.

On the other hand, the fact that a larger model typically has more capacity does not necessarily mean that the particular function it represents could not in principle be learnt by a simpler model. It could be the case that the larger model wastes capacity. For instance, in our two case studies in chapter 3, we observed that the prediction surfaces represented by large bags of MCMC samples were actually rather simple and smooth, and therefore could be easily represented by more compact models. After all, this is exactly what knowledge distillation as a metaphor is all about: the fact that the knowledge of the large model can be contained in a condensed, distilled form in the compact model, not wasting capacity.

But if the knowledge can fit in a compact model, why then not just train and use the compact model in the first place? Certainly, in several cases this is the right thing to do. However, there are situations where training a larger model and then distilling it into a smaller one might actually be preferable. In the following paragraphs, we describe four such situations.

Firstly, it might not be at all obvious how the compact model could be trained directly. In chapter 3, where a compact model was trained to approximate the true predictive distribution in the context of Bayesian inference, we needed to explore the posterior with MCMC in order to extract information from it. It is not obvious how we could have fitted a compact model to the predictive distribution directly, since the true predictive distribution is typically intractable.<sup>1</sup>

Secondly, the fact that a compact model can in principle represent the knowledge of a large model does not necessarily mean that it can also extract it directly from data. Perhaps our learning algorithms are simply not good enough yet. Existing learning algorithms might favour larger models, and more capacity might be needed for the successful extraction of knowledge from data. Hence, it might be more effective to first extract the knowledge from the data by a large capacity-wasting model, and then successfully distil it to a more compact model.

Thirdly, we might not have enough labelled data for training the compact model to begin with. A large model, together with proper regularization, might be better at learning from a small dataset. On the other hand, as we have seen, knowledge distillation does not suffer from shortage of labelled data, as it does not need labels at all. For instance, in the context of model compression (chapter 2) all labelling is done by the teacher model. As long as we have access to a suitable data generator (and in this work we demonstrated ways for building one) then we can have as much input data as we need.

Fourthly, it could be the case that we actually want the convenient model to be consistent with the cumbersome model. For example, in our case study in chapter 4, we saw that we could obtain better estimates of the partition function of the RBM by having a NADE that matched it as closely as possible. Moreover, it could be the case that the cumbersome model we try to emulate is actually the one and only true model. For instance, in chapter 3, the predictive distribution the compact model was trying to match was the one and only true distribution (according the rules of probability). The same would be true if we were trying to distil an intractable true posterior to a tractable model, as discussed in chapter 4.

## 5.2 What is there next?

Model compression, Bayesian inference and generative models do not exhaust the applications of knowledge distillation. Another potential application is in learning efficient models of otherwise expensive to evaluate functions. We can imagine having a function we could evaluate, but doing so routinely would be too expensive. Such a function, for instance, could be the result of a long physics simulation. Using knowledge distillation techniques, we could construct an efficient model,

---

<sup>1</sup>Snelson and Ghahramani (2005b) tried fitting a compact model directly to the predictive distribution using variational inference, but with little success.

such as a neural network, that would compute the same function in only a fraction of the time. Matching derivatives could prove useful in a situation like this, as it could reduce the number of times the original function would have to be called.

Distilling an expensive to evaluate function into a neural network was also done by Heess et al. (2013). Their work focuses on speeding up the calculation of messages that need to be passed within a graphical model, when using Expectation Propagation for doing inference. In EP, outgoing messages are deterministic functions of incoming messages that are often expensive to evaluate. By distilling the functions that compute messages into neural networks, they were able to make EP run in only a fraction of the time.

Another potential extension of knowledge distillation could be in the framework itself. As we have seen in previous chapters, knowledge distillation often requires the solution of an optimization problem of the form

$$\min_{\boldsymbol{\theta}} \langle E(\mathbf{x}, \boldsymbol{\theta}) \rangle_{p(\mathbf{x})}, \quad (5.1)$$

where  $E(\mathbf{x}, \boldsymbol{\theta})$  is some loss and  $p(\mathbf{x})$  is some input distribution. In our framework, we proposed optimizing objectives of the above form using a stochastic approximation, by first generating samples from  $p(\mathbf{x})$  and then minimizing the empirical average over the samples. This approach prioritizes the minimization of  $E(\mathbf{x}, \boldsymbol{\theta})$  at locations  $\mathbf{x}$  with high probability. However, we could imagine prioritizing locations  $\mathbf{x}$  for which the loss is high as well. In other words, the minimization procedure could actively seek input locations for which minimizing the loss would have the strongest impact. Such a scheme may need fewer loss evaluations, and would be particularly beneficial when such evaluations are expensive.

In this thesis we validated our knowledge distillation framework on example problems and standard machine learning benchmarks, such as the MNIST dataset of handwritten digits. It remains to be seen how useful knowledge distillation will prove to be in large-scale industrial applications. Our hope is that the techniques and ideas presented in this thesis will contribute to extending the applicability of machine learning to yet more interesting and useful practical problems.

## Appendix A

# The R Technique

Let  $f$  be a scalar-valued function of a multidimensional vector  $\mathbf{x}$ . Assume we are interested in calculating  $\mathbf{H}\mathbf{v}$ , where  $\mathbf{H}$  is the Hessian of  $f$ , i.e. the matrix of all second derivatives of  $f$ , and  $\mathbf{v}$  is an arbitrary vector of the same size as  $\mathbf{x}$ . Calculating and storing  $\mathbf{H}$  takes quadratic time and space in the size of  $\mathbf{x}$ , making it inefficient for a large number of inputs. Nevertheless, storing  $\mathbf{H}\mathbf{v}$  takes linear space in the size of  $\mathbf{x}$  and, as we will describe, computing it can be done in linear time as well.

Calculating  $\mathbf{H}\mathbf{v}$  in linear time is made possible by the *R technique*, first described by Pearlmutter (1994). Given some vector-valued function  $\mathbf{g}$  of  $\mathbf{x}$  and a vector  $\mathbf{v}$  of the same size as  $\mathbf{x}$ , we define the operator  $\mathcal{R}\{\cdot\}$  as

$$\mathcal{R}\{\mathbf{g}\} = \left( \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \right)^T \mathbf{v}. \quad (\text{A.1})$$

Since  $\mathcal{R}\{\cdot\}$  is a differential operator, its properties are similar to the familiar properties of derivatives. The following properties can be easily proved using the definition.

- (i)  $\mathcal{R}\{\mathbf{c}\} = \mathbf{0}$  for any constant  $\mathbf{c}$
- (ii)  $\mathcal{R}\{\mathbf{x}\} = \mathbf{v}$
- (iii)  $\mathcal{R}\{\mathbf{g}_1 + \mathbf{g}_2\} = \mathcal{R}\{\mathbf{g}_1\} + \mathcal{R}\{\mathbf{g}_2\}$
- (iv)  $\mathcal{R}\{\mathbf{g}_1 \mathbf{g}_2\} = \mathcal{R}\{\mathbf{g}_1\} \mathbf{g}_2 + \mathbf{g}_1 \mathcal{R}\{\mathbf{g}_2\}$
- (v)  $\mathcal{R}\{\mathbf{g}_1(\mathbf{g}_2)\} = \left( \frac{\partial \mathbf{g}_1}{\partial \mathbf{g}_2} \right)^T \mathcal{R}\{\mathbf{g}_2\}$

Using the  $\mathcal{R}\{\cdot\}$  operator, it is easy to see that  $\mathbf{H}\mathbf{v}$  can be written as  $\mathcal{R}\left\{ \frac{\partial f}{\partial \mathbf{x}} \right\}$ . The latter can then be calculated in linear time by the same equations that are used to calculate  $\frac{\partial f}{\partial \mathbf{x}}$ , after transforming them by  $\mathcal{R}\{\cdot\}$  making use of the above list of properties.

## Appendix B

# Derivatives in Neural Networks

A feedforward neural network is a function  $f$ , parameterized by a set of parameters  $\theta$ , that maps some input  $\mathbf{x}$  to some output  $\mathbf{y}$ . Mathematically, we write

$$\mathbf{y} = f(\mathbf{x} \mid \theta). \quad (\text{B.1})$$

In this appendix, we discuss forward and backward propagation in neural networks, as well as common nonlinearities and output functions, with a focus on how to efficiently calculate first and second derivatives.

### B.1 Computing the output: forward propagation

The network  $f$  is made of  $L$  cascaded layers, each consisting of a linear and a nonlinear phase, together with an input layer. Let  $\mathbf{x}^{(\ell)}$  be the output of the  $\ell^{\text{th}}$  layer, with  $\ell = 0$  corresponding to the input layer. Then,  $f$  is defined by the following recursive procedure, known as *forward propagation*.

$$\mathbf{x}^{(0)} = \mathbf{x} \quad (\text{B.2})$$

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{x}^{(\ell-1)} + \mathbf{b}^{(\ell)} \quad (\text{B.3})$$

$$\mathbf{x}^{(\ell)} = \phi^{(\ell)}(\mathbf{z}^{(\ell)}) \quad (\text{B.4})$$

$$\mathbf{y} = \mathbf{x}^{(L)} \quad (\text{B.5})$$

The network's adjustable parameters  $\theta$  are the collection of parameters from all linear phases, that is,  $\theta = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}$ . The nonlinear phase of the  $\ell^{\text{th}}$  layer is specified by  $\phi^{(\ell)}$ , which in practice can be any convenient nonlinear function. The choice of  $\phi^{(\ell)}$  and the number of layers  $L$  are part of the network's design and do not belong to the set of adjustable parameters.

In order to reduce unnecessary clutter and make the calculations more explicit, we will drop the dependence on  $\ell$  and expand the matrix equations, and thus we will express forward propagation as

$$z_i = \sum_j w_{ij} x_j + b_i \quad (\text{B.6})$$

$$x_i = \phi_i(z_1, z_2, \dots, z_I). \quad (\text{B.7})$$

In a slight abuse of notation, our convention is that variables indexed by  $i$  belong to layer  $\ell$  and variables indexed by  $j$  belong to layer  $\ell - 1$ .

### B.2 Computing first derivatives: backward propagation

Assume that the network's output  $\mathbf{y}$  is mapped to a real value  $E(\mathbf{y})$  by some differentiable function  $E$ . We would like to compute the derivatives of  $E$  with respect to the network's parameters  $\theta$

and inputs  $\mathbf{x}$ , that is  $\frac{\partial E}{\partial \boldsymbol{\theta}}$  and  $\frac{\partial E}{\partial \mathbf{x}}$ . This is easily done by applying the chain rule from calculus to the equations describing forward propagation, yielding the following recursive procedure known as *backward propagation* or *backprop* for short.

$$\frac{\partial E}{\partial b_i} = \sum_{i'} \frac{\partial E}{\partial x_{i'}} \frac{\partial \phi_{i'}}{\partial z_i} \quad (\text{B.8})$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial b_i} x_j \quad (\text{B.9})$$

$$\frac{\partial E}{\partial x_j} = \sum_{i'} \frac{\partial E}{\partial b_{i'}} w_{i'j} \quad (\text{B.10})$$

The recursion starts by setting  $\frac{\partial E}{\partial \mathbf{x}^{(L)}} = \frac{\partial E}{\partial \mathbf{y}}$ , which we assume we can calculate, and works from the output layer all the way down to the input layer. In each iteration, it yields  $\frac{\partial E}{\partial \mathbf{W}^{(\ell)}}$  and  $\frac{\partial E}{\partial \mathbf{b}^{(\ell)}}$ , and at the end of the recursion, it outputs  $\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{x}^{(0)}}$ . Forward propagation is assumed to have been run in advance, so that quantities  $\mathbf{x}^{(\ell)}$  and  $\frac{\partial \phi^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}$  are available. Note that if only  $\frac{\partial E}{\partial \mathbf{x}}$  is needed, then Equation (B.9)—which in practice is the most time-consuming—can be skipped.

In the common case where the nonlinearities  $\phi_i$  act elementwise, we have  $\frac{\partial \phi_{i'}}{\partial z_i} = 0$  when  $i' \neq i$ , so Equation (B.8) can be simplified to

$$\frac{\partial E}{\partial b_i} = \frac{\partial E}{\partial x_i} \frac{\partial \phi_i}{\partial z_i}. \quad (\text{B.11})$$

### B.3 Computing second derivatives: R{backprop}

Computing the matrix of second derivatives (i.e. the Hessian) of  $E$  with respect to the network's parameters  $\boldsymbol{\theta}$  and inputs  $\mathbf{x}$  is in principle possible, by applying the chain rule to the procedure jointly defined by forward and backward propagation. However, calculating or even storing the Hessian takes quadratic time and space in the size of  $\boldsymbol{\theta}$  and  $\mathbf{x}$ , making it inefficient for large networks. Nevertheless, in practice what is usually needed is not the full Hessian  $\mathbf{H}$  but its product with some vector  $\mathbf{v}$ , i.e.  $\mathbf{H}\mathbf{v}$ . Storing  $\mathbf{H}\mathbf{v}$  takes linear space in the size of  $\boldsymbol{\theta}$  and  $\mathbf{x}$  and it can be computed in linear time using the R technique (described in appendix A).

It is easy to see that  $\mathbf{H}\mathbf{v}$  can be written in terms of the  $\mathcal{R}\{\cdot\}$  operator as  $\mathcal{R}\left\{\frac{\partial E}{\partial (\boldsymbol{\theta}, \mathbf{x})}\right\}$ . The latter can be calculated in linear time by a procedure similar to joint forward and backward propagation, where all the equations have been transformed by  $\mathcal{R}\{\cdot\}$ . From here on, we will call this procedure *R{backprop}*. Using the properties of  $\mathcal{R}\{\cdot\}$  and applying it to both sides of the equations defining forward propagation, we get the forward phase of R{backprop}, which is as follows.

$$\mathcal{R}\{z_i\} = \sum_j (\mathcal{R}\{w_{ij}\} x_j + w_{ij} \mathcal{R}\{x_j\}) + \mathcal{R}\{b_i\} \quad (\text{B.12})$$

$$\mathcal{R}\{x_i\} = \sum_{i'} \frac{\partial \phi_i}{\partial z_{i'}} \mathcal{R}\{z_{i'}\}. \quad (\text{B.13})$$

In the above,  $\mathcal{R}\{\mathbf{W}^{(\ell)}\}$  and  $\mathcal{R}\{\mathbf{b}^{(\ell)}\}$  are simply the elements of  $\mathbf{v}$  that correspond to  $\mathbf{W}^{(\ell)}$  and  $\mathbf{b}^{(\ell)}$  respectively. Similarly,  $\mathcal{R}\{\mathbf{x}^{(0)}\}$  is initialized with the elements of  $\mathbf{v}$  that correspond to  $\mathbf{x}$ . Forward propagation is assumed to have been run in advance, so that quantities  $\mathbf{x}^{(\ell)}$  and  $\frac{\partial \phi^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}$  are available. Note also that in the common case where the nonlinearities  $\phi_i$  act elementwise, Equation (B.13) simplifies to

$$\mathcal{R}\{x_i\} = \frac{\partial \phi_i}{\partial z_i} \mathcal{R}\{z_i\}. \quad (\text{B.14})$$

Next, we apply  $\mathcal{R}\{\cdot\}$  to both sides of the backprop equations to get the backward phase of R{backprop}, which is as follows.

$$\mathcal{R}\left\{\frac{\partial E}{\partial b_i}\right\} = \sum_{i'} \left( \mathcal{R}\left\{\frac{\partial E}{\partial x_{i'}}\right\} \frac{\partial \phi_{i'}}{\partial z_i} + \frac{\partial E}{\partial x_{i'}} \mathcal{R}\left\{\frac{\partial \phi_{i'}}{\partial z_i}\right\} \right) \quad (\text{B.15})$$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{ij}}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial b_i}\right\} x_j + \frac{\partial E}{\partial b_i} \mathcal{R}\{x_j\} \quad (\text{B.16})$$

$$\mathcal{R}\left\{\frac{\partial E}{\partial x_j}\right\} = \sum_{i'} \left( \mathcal{R}\left\{\frac{\partial E}{\partial b_{i'}}\right\} w_{i'j} + \frac{\partial E}{\partial b_{i'}} \mathcal{R}\{w_{i'j}\} \right) \quad (\text{B.17})$$

In the above, we assume that  $\mathbf{x}^{(\ell)}$ ,  $\frac{\partial \phi^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}$  and  $\mathcal{R}\left\{\frac{\partial \phi^{(\ell)}}{\partial \mathbf{z}^{(\ell)}}\right\}$  are available from a previous run of forward propagation,  $\frac{\partial E}{\partial \mathbf{x}^{(\ell)}}$  and  $\frac{\partial E}{\partial \mathbf{b}^{(\ell)}}$  are available from a previous run of backprop, and  $\mathcal{R}\{\mathbf{x}^{(\ell)}\}$  is available from the forward phase of R{backprop}. The procedure needs to be initialized with  $\mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{x}^{(L)}}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{y}}\right\}$ , whose calculation depends on the choice of function  $E$ . Finally, in each iteration the procedure yields  $\mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{W}^{(\ell)}}\right\}$  and  $\mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{b}^{(\ell)}}\right\}$ , and at the end it outputs  $\mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{x}}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{x}^{(0)}}\right\}$ , which collectively form  $\mathbf{Hv}$ . Note that in the common case where the nonlinearities  $\phi_i$  act elementwise, Equation (B.15) simplifies to

$$\mathcal{R}\left\{\frac{\partial E}{\partial b_i}\right\} = \mathcal{R}\left\{\frac{\partial E}{\partial x_i}\right\} \frac{\partial \phi_i}{\partial z_i} + \frac{\partial E}{\partial x_i} \mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\}. \quad (\text{B.18})$$

A couple of simplifications in the R{backprop} procedure are possible if only part of the Hessian needs to be considered. Let  $\mathbf{v} = [\mathbf{v}_\theta \quad \mathbf{v}_x]^T$ , where  $\mathbf{v}_\theta$  is the part of  $\mathbf{v}$  corresponding to  $\theta$  and  $\mathbf{v}_x$  is the part of  $\mathbf{v}$  corresponding to  $x$ . Then, the Hessian-vector product can be written as

$$\mathbf{Hv} = \begin{bmatrix} \frac{\partial^2 E}{\partial \theta^2} \mathbf{v}_\theta + \frac{\partial^2 E}{\partial \theta \partial x} \mathbf{v}_x \\ \frac{\partial^2 E}{\partial x \partial \theta} \mathbf{v}_\theta + \frac{\partial^2 E}{\partial x^2} \mathbf{v}_x \end{bmatrix}. \quad (\text{B.19})$$

If only  $\frac{\partial^2 E}{\partial \theta \partial x} \mathbf{v}_x$  and  $\frac{\partial^2 E}{\partial x^2} \mathbf{v}_x$  are needed, we can equivalently set  $\mathbf{v}_\theta = \mathbf{0}$ . In this case, Equation (B.12) simplifies to

$$\mathcal{R}\{z_i\} = \sum_j w_{ij} \mathcal{R}\{x_j\} \quad (\text{B.20})$$

and Equation (B.17) simplifies to

$$\mathcal{R}\left\{\frac{\partial E}{\partial x_j}\right\} = \sum_{i'} \mathcal{R}\left\{\frac{\partial E}{\partial b_{i'}}\right\} w_{i'j}. \quad (\text{B.21})$$

## B.4 Common nonlinearities

In forward propagation, the output  $x_i$  of a layer is given by applying a nonlinearity  $\phi_i$ , that is

$$x_i = \phi_i(z_1, z_2, \dots, z_I). \quad (\text{B.22})$$

As part of the network's design, for each layer a nonlinearity  $\phi_i$  has to be specified, together with its derivatives  $\frac{\partial \phi_i}{\partial z_j}$  and  $\mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_j}\right\}$ , which are needed for backprop and R{backprop} respectively. In this section, we describe some common choices for  $\phi_i$ , namely linear, rectified linear, logistic, probit and softmax.

### B.4.1 Linear

The simplest choice for  $\phi_i$  is the identity function, that is

$$\phi_i(z_i) = z_i. \quad (\text{B.23})$$

This function acts elementwise and its derivatives are

$$\frac{\partial \phi_i}{\partial z_i} = 1 \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\} = 0. \quad (\text{B.24})$$

It is easy to see that the expressions in backprop and  $\mathcal{R}\{\text{backprop}\}$  that involve  $\frac{\partial \phi_i}{\partial z_i}$  or  $\mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\}$  can be significantly simplified in this case.

A linear layer is a good choice for the final layer of a network used for regression. Note however that a network having only linear layers throughout can only represent linear functions.

### B.4.2 Rectified linear

The rectified linear layer (or rectified linear unit—ReLU for short) acts elementwise on its inputs and is defined as

$$\phi_i(z_i) = \max(0, z_i) \quad (\text{B.25})$$

that is, it behaves as linear for  $z_i > 0$  and as constant (zero) otherwise. Its derivatives are

$$\frac{\partial \phi_i}{\partial z_i} = \begin{cases} 0 & z_i < 0 \\ 1 & z_i > 0 \end{cases} \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\} = 0. \quad (\text{B.26})$$

Notice that  $\phi_i$  is not differentiable at one point, namely at  $z_i = 0$ . However, since it is differentiable arbitrarily close to it, in practice we can take the derivative at  $z_i = 0$  to be either 0 or 1. Of course, expressions in  $\mathcal{R}\{\text{backprop}\}$  that involve  $\mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\}$  can be simplified due to it being always equal to 0.

### B.4.3 Logistic

The logistic nonlinearity acts elementwise and is defined as

$$\phi_i(z_i) = \frac{1}{1 + \exp(-z_i)}. \quad (\text{B.27})$$

Given that  $x_i = \phi_i(z_i)$ , it can easily be shown that its first derivative can be written as

$$\frac{\partial \phi_i}{\partial z_i} = x_i(1 - x_i), \quad (\text{B.28})$$

and therefore

$$\mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\} = \mathcal{R}\{x_i\}(1 - 2x_i). \quad (\text{B.29})$$

Mathematically it is always the case that  $0 < \phi_i(z_i) < 1$ . However, due to numerical reasons, it is easy for  $\phi_i(z_i)$  to underflow to 0 for large negative  $z_i$  or to 1 for large positive  $z_i$ , which is undesirable. Therefore care needs to be taken; one possible solution is to limit  $z_i$  to be within a safe range.

### B.4.4 Probit

The probit function is similar in shape to the logistic and also acts elementwise. It is defined as the cumulative density of the standard normal, that is

$$\phi_i(z_i) = \int_{-\infty}^{z_i} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{t^2}{2}\right) dt. \quad (\text{B.30})$$

Its derivatives can be easily calculated,

$$\frac{\partial \phi_i}{\partial z_i} = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z_i^2}{2}\right), \quad (\text{B.31})$$

and, given that  $\mathcal{R}\{x_i\} = \frac{\partial \phi_i}{\partial z_i} \mathcal{R}\{z_i\}$ ,

$$\mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_i}\right\} = -z_i \mathcal{R}\{x_i\}. \quad (\text{B.32})$$

Similarly to the logistic function, it has to be the case that  $0 < \phi_i(z_i) < 1$ , therefore care has to be taken to avoid underflow to 0 or 1 for large negative or positive values of  $z_i$  respectively.

### B.4.5 Softmax and log-softmax

Unlike the nonlinearities discussed so far, the softmax function does not act elementwise and it is defined as

$$\phi_i(z_1, z_2, \dots, z_I) = \frac{\exp z_i}{\sum_{i'} \exp z_{i'}}. \quad (\text{B.33})$$

Notice that the outputs  $x_i$  of a softmax layer form a categorical probability distribution, since  $0 < x_i < 1$  and  $\sum_i x_i = 1$ , therefore the softmax function is commonly used in the final layer of networks that perform multi-label classification. Given that  $x_i = \phi_i(z_1, z_2, \dots, z_I)$ , its derivatives can be easily calculated to be

$$\frac{\partial \phi_i}{\partial z_j} = x_i(\delta_{ij} - x_j), \quad (\text{B.34})$$

where  $\delta_{ij}$  is the Kronecker delta, and

$$\mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_j}\right\} = \mathcal{R}\{x_i\}(\delta_{ij} - x_j) - x_i \mathcal{R}\{x_j\}. \quad (\text{B.35})$$

When using the softmax, care has to be taken so as to prevent the exponential from overflowing or underflowing. One approach is based on the fact that for any scalar  $c$  we have

$$\frac{\exp(z_i - c)}{\sum_{i'} \exp(z_{i'} - c)} = \frac{\exp z_i}{\sum_{i'} \exp z_{i'}}. \quad (\text{B.36})$$

Taking  $c = \max(z_1, z_2, \dots, z_I)$  prevents the exponential from overflowing. However, underflowing is still possible. A better approach is to work entirely with the logarithm of the softmax (or log-softmax for short). In this case, we define

$$\phi_i(z_1, z_2, \dots, z_I) = z_i - \log \sum_{i'} \exp z_{i'} \quad (\text{B.37})$$

whose derivatives are

$$\frac{\partial \phi_i}{\partial z_j} = \delta_{ij} - \exp x_j \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial \phi_i}{\partial z_j}\right\} = -(\exp x_j) \mathcal{R}\{x_j\}. \quad (\text{B.38})$$

Note that for any scalars  $c$  we also have

$$(z_i - c) + \log \sum_{i'} \exp(z_{i'} - c) = z_i + \log \sum_{i'} \exp z_{i'} \quad (\text{B.39})$$

therefore we can prevent both overflow and underflow by taking  $c = \max(z_1, z_2, \dots, z_I)$ .

## B.5 Useful output functions

In backprop and  $\mathcal{R}\{\text{backprop}\}$ , we assumed that the network's output  $\mathbf{y}$  is mapped to  $E(\mathbf{y})$  by some differentiable function  $E$ . Typically,  $E$  measures how much the network fits or deviates from a target, and the objective is to maximize or minimize it accordingly with respect to the network's adjustable parameters  $\boldsymbol{\theta}$ . As we have seen, backprop and  $\mathcal{R}\{\text{backprop}\}$  need to be initialized with  $\frac{\partial E}{\partial \mathbf{y}}$  and  $\mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{y}}\right\}$  respectively. Below, we discuss some common choices for  $E$  and provide the corresponding derivatives.

### B.5.1 Square error

The square error is typically used for regression problems, and it is defined as

$$E(\mathbf{y}) = \frac{1}{2} \|\mathbf{y} - \mathbf{t}\|^2, \quad (\text{B.40})$$

for some given target vector  $\mathbf{t}$ . Its derivatives are

$$\frac{\partial E}{\partial \mathbf{y}} = \mathbf{y} - \mathbf{t} \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{y}}\right\} = \mathcal{R}\{\mathbf{y}\}. \quad (\text{B.41})$$

### B.5.2 Cross entropy

Cross entropy is typically used for classification problems. Given a target vector  $\mathbf{t}$ , it is defined as

$$E(\mathbf{y}) = \sum_i t_i \log y_i, \quad (\text{B.42})$$

with derivatives

$$\frac{\partial E}{\partial y_i} = \frac{t_i}{y_i} \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial E}{\partial y_i}\right\} = -\frac{t_i}{y_i^2} \mathcal{R}\{y_i\}. \quad (\text{B.43})$$

Cross entropy is suitable when both  $\mathbf{t}$  and  $\mathbf{y}$  are discrete probability distributions (e.g.  $\mathbf{y}$  could be the output of a softmax layer), in which case maximizing  $E$  is the same as minimizing  $D_{\text{KL}}(\mathbf{t} \parallel \mathbf{y})$ .

For binary classification problems,  $t$  and  $y$  are typically scalars, and represent the probability of one of the two classes. In this case, the following version of cross entropy can be used

$$E(y) = t \log y + (1 - t) \log (1 - y) \quad (\text{B.44})$$

whose derivatives are

$$\frac{\partial E}{\partial y} = \frac{t}{y} - \frac{1-t}{1-y} \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial E}{\partial y}\right\} = -\left(\frac{t}{y^2} + \frac{1-t}{(1-y)^2}\right) \mathcal{R}\{y\}. \quad (\text{B.45})$$

### B.5.3 Dot product

Given a target vector  $\mathbf{t}$ , the dot product is simply

$$E(\mathbf{y}) = \mathbf{t}^T \mathbf{y} \quad (\text{B.46})$$

with derivatives

$$\frac{\partial E}{\partial \mathbf{y}} = \mathbf{t} \quad \text{and} \quad \mathcal{R}\left\{\frac{\partial E}{\partial \mathbf{y}}\right\} = \mathbf{0}. \quad (\text{B.47})$$

Note that using the softmax layer in combination with cross entropy is equivalent to using the log-softmax layer in combination with the dot product.

# Appendix C

## Derivatives in NADE

NADE is a generative model, parameterized by a set of parameters  $\theta$ , which takes a binary input vector  $\mathbf{x}$  and outputs its log probability

$$L = \log p(\mathbf{x} | \theta). \quad (\text{C.1})$$

An overview of NADE has been given in section 4.2.2. In this appendix, we discuss in more detail forward and backward propagation in NADE, with a focus on how to efficiently calculate the first and second derivatives of  $L$ .

### C.1 Computing the output: forward propagation

NADE is essentially a feedforward neural network with a single hidden layer and a special autoregressive structure. As such, its output  $L$  is given by a procedure similar to forward propagation in neural networks. Assuming NADE has  $I$  input units,  $I$  output units and  $J$  hidden units, forward propagation in NADE is described below.

$$a_{1j} = c_j \quad (\text{C.2})$$

$$a_{ij} = a_{i-1,j} + w_{i-1,j}x_{i-1} \quad (\text{for } i > 1) \quad (\text{C.3})$$

$$h_{ij} = \sigma(a_{ij}) \quad (\text{C.4})$$

$$z_i = \sum_j u_{ij}h_{ij} + b_i \quad (\text{C.5})$$

$$\hat{x}_i = \sigma(z_i) \quad (\text{C.6})$$

$$L = \sum_i (x_i \log \hat{x}_i + (1 - x_i) \log (1 - \hat{x}_i)) \quad (\text{C.7})$$

In the above equations,  $u_{ij}$ ,  $b_i$ ,  $w_{ij}$  and  $c_j$  are the adjustable parameters of NADE, which are collectively denoted as  $\theta$ . Also,  $a_{ij}$  and  $h_{ij}$  are the hidden layer activations and hidden layer states respectively, and  $z_i$  are the output layer activations. Our notation is that  $i$  indexes input and output units, and  $j$  indexes hidden units. Note that the number of input/output units  $I$  and the number of hidden units  $J$  do not belong to the set of adjustable parameters, but are part of NADE's architecture.

### C.2 Computing first derivatives: backward propagation

Computing the derivatives of NADE with respect to the parameters  $\frac{\partial L}{\partial \theta}$  and with respect to the input  $\frac{\partial L}{\partial \mathbf{x}}$  can easily be done by applying the chain rule of calculus to the equations describing forward propagation. This yields a procedure similar to backward propagation in neural networks,

which for NADE is as follows.

$$\frac{\partial L}{\partial b_i} = x_i - \hat{x}_i \quad (\text{C.8})$$

$$\frac{\partial L}{\partial u_{ij}} = \frac{\partial L}{\partial b_i} h_{ij} \quad (\text{C.9})$$

$$\frac{\partial L}{\partial a_{ij}} = \frac{\partial L}{\partial u_{ij}} u_{ij} (1 - h_{ij}) \quad (\text{C.10})$$

$$\frac{\partial L}{\partial c_j} = \sum_i \frac{\partial L}{\partial a_{ij}} \quad (\text{C.11})$$

$$\frac{\partial L}{\partial w_{ij}} = \left( \sum_{i' > i} \frac{\partial L}{\partial a_{i'j}} \right) x_i \quad (\text{C.12})$$

$$\frac{\partial L}{\partial x_i} = z_i + \sum_j \left( \sum_{i' > i} \frac{\partial L}{\partial a_{i'j}} \right) w_{ij} \quad (\text{for } i < I) \quad (\text{C.13})$$

$$\frac{\partial L}{\partial x_I} = z_I \quad (\text{C.14})$$

Before running backward propagation, we assume that forward propagation has already been performed, so that quantities  $h_{ij}$ ,  $z_i$  and  $\hat{x}_i$  are available. Notice that the cumulative sum  $\sum_{i' > i} \frac{\partial L}{\partial a_{i'j}}$  can be computed efficiently, by using the recursion

$$\sum_{i' > i} \frac{\partial L}{\partial a_{i'j}} = \frac{\partial L}{\partial a_{i+1,j}} + \sum_{i' > i+1} \frac{\partial L}{\partial a_{i'j}}. \quad (\text{C.15})$$

Note also that, despite  $\mathbf{x}$  being actually binary, there is nothing in the maths stopping us from pretending it is continuous and computing derivatives with respect to it. Finally, if only the derivatives with respect to  $\mathbf{x}$  are needed, then Equations (C.11) and (C.12) can be skipped.

### C.3 Computing second derivatives: R{backprop}

Let  $\mathbf{H}$  be the Hessian of  $L$  with respect to the parameters and the inputs and  $\mathbf{v}$  be some vector of appropriate size. Even though computing  $\mathbf{H}$  explicitly is in principle possible by applying the chain rule of calculus to the procedure jointly defined by forward and backward propagation, in practice what is usually needed is only the Hessian-vector product  $\mathbf{H}\mathbf{v}$ . The latter can be computed in linear time using the R technique (see appendix A). Compare this to calculating  $\mathbf{H}$  explicitly, which takes quadratic time and space.

The R technique is based on applying the  $\mathcal{R}\{\cdot\}$  operator to the equations describing forward and backward propagation, yielding a procedure that we shall call R{backprop}. The forward part of R{backprop} is as follows.

$$\mathcal{R}\{a_{1j}\} = \mathcal{R}\{c_j\} \quad (\text{C.16})$$

$$\mathcal{R}\{a_{ij}\} = \mathcal{R}\{a_{i-1,j}\} + \mathcal{R}\{w_{i-1,j}\} x_{i-1} + w_{i-1,j} \mathcal{R}\{x_{i-1}\} \quad (\text{for } i > 1) \quad (\text{C.17})$$

$$\mathcal{R}\{h_{ij}\} = h_{ij}(1 - h_{ij}) \mathcal{R}\{a_{ij}\} \quad (\text{C.18})$$

$$\mathcal{R}\{z_i\} = \sum_j (\mathcal{R}\{u_{ij}\} h_{ij} + u_{ij} \mathcal{R}\{h_{ij}\}) + \mathcal{R}\{b_i\} \quad (\text{C.19})$$

$$\mathcal{R}\{\hat{x}_i\} = \hat{x}_i(1 - \hat{x}_i) \mathcal{R}\{z_i\} \quad (\text{C.20})$$

In the above,  $\mathcal{R}\{u_{ij}\}$ ,  $\mathcal{R}\{b_i\}$ ,  $\mathcal{R}\{w_{ij}\}$ ,  $\mathcal{R}\{c_j\}$  and  $\mathcal{R}\{x_i\}$  are simply the corresponding elements of  $\mathbf{v}$ . We assume that forward propagation has been performed in advance, so that quantities  $h_{ij}$  and  $\hat{x}_i$  are available.

Applying the  $\mathcal{R}\{\cdot\}$  operator to the equations describing backward propagation yields the backward phase of  $R\{backprop\}$ , which is as follows.

$$\mathcal{R}\left\{\frac{\partial L}{\partial b_i}\right\} = \mathcal{R}\{x_i\} - \mathcal{R}\{\hat{x}_i\} \quad (C.21)$$

$$\mathcal{R}\left\{\frac{\partial L}{\partial u_{ij}}\right\} = \mathcal{R}\left\{\frac{\partial L}{\partial b_i}\right\} h_{ij} + \frac{\partial L}{\partial b_i} \mathcal{R}\{h_{ij}\} \quad (C.22)$$

$$\mathcal{R}\left\{\frac{\partial L}{\partial a_{ij}}\right\} = \mathcal{R}\left\{\frac{\partial L}{\partial u_{ij}}\right\} u_{ij}(1 - h_{ij}) + \frac{\partial L}{\partial u_{ij}} \mathcal{R}\{u_{ij}\}(1 - h_{ij}) - \frac{\partial L}{\partial u_{ij}} u_{ij} \mathcal{R}\{h_{ij}\} \quad (C.23)$$

$$\mathcal{R}\left\{\frac{\partial L}{\partial c_j}\right\} = \sum_i \mathcal{R}\left\{\frac{\partial L}{\partial a_{ij}}\right\} \quad (C.24)$$

$$\mathcal{R}\left\{\frac{\partial L}{\partial w_{ij}}\right\} = \left( \sum_{i' > i} \mathcal{R}\left\{\frac{\partial L}{\partial a_{i'j}}\right\} \right) x_i + \left( \sum_{i' > i} \frac{\partial L}{\partial a_{i'j}} \right) \mathcal{R}\{x_i\} \quad (C.25)$$

$$\mathcal{R}\left\{\frac{\partial L}{\partial x_i}\right\} = \mathcal{R}\{z_i\} + \sum_j \left( \sum_{i' > i} \mathcal{R}\left\{\frac{\partial L}{\partial a_{i'j}}\right\} \right) w_{ij} + \sum_j \left( \sum_{i' > i} \frac{\partial L}{\partial a_{i'j}} \right) \mathcal{R}\{w_{ij}\} \quad (\text{for } i < I) \quad (C.26)$$

$$\mathcal{R}\left\{\frac{\partial L}{\partial x_I}\right\} = \mathcal{R}\{z_I\} \quad (C.27)$$

In the above,  $h_{ij}$  is available from a previous run of forward propagation,  $\frac{\partial L}{\partial b_i}$ ,  $\frac{\partial L}{\partial u_{ij}}$  and  $\frac{\partial L}{\partial a_{ij}}$  are available from a previous run of backward propagation, and  $\mathcal{R}\{\hat{x}_i\}$ ,  $\mathcal{R}\{h_{ij}\}$  and  $\mathcal{R}\{z_i\}$  are available from the forward phase of  $R\{backprop\}$ .

$R\{backprop\}$  can be significantly simplified if only part of the Hessian needs to be calculated. To see that, let  $\mathbf{v} = [\mathbf{v}_{\theta} \quad \mathbf{v}_{\mathbf{x}}]^T$ , where  $\mathbf{v}_{\theta}$  is the part of  $\mathbf{v}$  corresponding to  $\theta$  and  $\mathbf{v}_{\mathbf{x}}$  is the part of  $\mathbf{v}$  corresponding to  $\mathbf{x}$ . The Hessian-vector product can then be written as

$$\mathbf{H}\mathbf{v} = \begin{bmatrix} \frac{\partial^2 L}{\partial \theta^2} \mathbf{v}_{\theta} + \frac{\partial^2 L}{\partial \theta \partial \mathbf{x}} \mathbf{v}_{\mathbf{x}} \\ \frac{\partial^2 L}{\partial \mathbf{x} \partial \theta} \mathbf{v}_{\theta} + \frac{\partial^2 L}{\partial \mathbf{x}^2} \mathbf{v}_{\mathbf{x}} \end{bmatrix}. \quad (C.28)$$

If, for instance, only  $\frac{\partial^2 L}{\partial \theta \partial \mathbf{x}} \mathbf{v}_{\mathbf{x}}$  needs to be calculated, we can simply set  $\mathbf{v}_{\theta} = \mathbf{0}$ . Hence, all  $\mathcal{R}\{u_{ij}\}$ ,  $\mathcal{R}\{b_i\}$ ,  $\mathcal{R}\{w_{ij}\}$  and  $\mathcal{R}\{c_j\}$  will become 0, which simplifies the calculations significantly.

## Appendix D

# Batch and Online EM for MoG

Suppose we have a dataset  $\{\mathbf{x}_n\}$ , to which we want to fit a MoG using maximum likelihood. The MoG is given by

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \sum_i \pi_i \mathcal{N}(\mathbf{x} \mid \mathbf{m}_i, \mathbf{S}_i). \quad (\text{D.1})$$

The MoG's adjustable parameters are  $\boldsymbol{\theta} = \{\pi_i, \mathbf{m}_i, \mathbf{S}_i\}$ . Fitting  $p(\mathbf{x} \mid \boldsymbol{\theta})$  to the dataset using maximum likelihood amounts to solving the following optimization problem

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_n \log p(\mathbf{x}_n \mid \boldsymbol{\theta}). \quad (\text{D.2})$$

One way to solve the above problem is via the *EM algorithm*. In this appendix, we describe two versions of the EM algorithm for MoG, the classic batch version, and an online version.

### D.1 Batch EM

Let  $z_{in}$  be a binary random variable with the meaning “datapoint  $\mathbf{x}_n$  was generated by the  $i^{\text{th}}$  component”. The batch EM algorithm is an iterative procedure which alternates between two phases: the *expectation* phase, where it computes the expectation of each  $z_{in}$  using the current value of  $\boldsymbol{\theta}$ , and the *maximization* phase, where it updates  $\boldsymbol{\theta}$  using the previously computed expectations of  $z_{in}$ .

More precisely, in the expectation phase, the algorithm computes  $\langle z_{in} \rangle$  for all  $i$  and  $n$  as follows

$$\langle z_{in} \rangle = \frac{\pi_i \mathcal{N}(\mathbf{x}_n \mid \mathbf{m}_i, \mathbf{S}_i)}{\sum_{i'} \pi_{i'} \mathcal{N}(\mathbf{x}_n \mid \mathbf{m}_{i'}, \mathbf{S}_{i'})}. \quad (\text{D.3})$$

Note that the above expectation is taken with respect to the posterior of  $z_{in}$  given  $\mathbf{x}_n$  and  $\boldsymbol{\theta}$  (not indicated to simplify notation). In the maximization phase, using the previously computed expectations of  $z_{in}$ , the algorithm computes for each  $i$  the following three statistics

$$\Phi_{i1} = \sum_n \langle z_{in} \rangle \quad (\text{D.4})$$

$$\Phi_{i2} = \sum_n \langle z_{in} \rangle \mathbf{x}_n \quad (\text{D.5})$$

$$\Phi_{i3} = \sum_n \langle z_{in} \rangle \mathbf{x}_n \mathbf{x}_n^T \quad (\text{D.6})$$

and then it updates  $\pi_i$ ,  $\mathbf{m}_i$  and  $\mathbf{S}_i$  as follows

$$\pi_i = \frac{1}{N} \Phi_{i1} \quad (\text{D.7})$$

$$\mathbf{m}_i = \frac{\Phi_{i2}}{\Phi_{i1}} \quad (\text{D.8})$$

$$\mathbf{S}_i = \frac{\Phi_{i3}}{\Phi_{i1}} - \mathbf{m}_i \mathbf{m}_i^T. \quad (\text{D.9})$$

It can be shown (Bishop, 2006, section 9.4) that each iteration of the above procedure can only increase  $\frac{1}{N} \sum_n \log p(\mathbf{x}_n | \boldsymbol{\theta})$  or leave it the same. Hence, by repeating the above until convergence, a local maximum of the average log likelihood can be found.

## D.2 Online EM

The downside of batch EM is that it needs to scan the whole dataset in every iteration. For large or possibly infinite datasets, this becomes problematic. Online EM, on the other hand, uses only a minibatch of points from the dataset in each iteration, which can be made as small as a single point. There are more than one flavours of online EM (Liang and Klein, 2009). In this thesis, we make use of the version by Cappé and Moulines (2009), which is exemplified below for the MoG case.

Suppose that  $\{\mathbf{x}_m\}$  is a minibatch of  $M$  points from the dataset selected at random. Note that  $M$  can be as small as 1, however larger values are typically used for better stability. For each  $m$  and  $i$ , online EM computes the expectation of the associated variable  $z_{im}$  as follows

$$\langle z_{im} \rangle = \frac{\pi_i \mathcal{N}(\mathbf{x}_m | \mathbf{m}_i, \mathbf{S}_i)}{\sum_{i'} \pi_{i'} \mathcal{N}(\mathbf{x}_m | \mathbf{m}_{i'}, \mathbf{S}_{i'})} \quad (\text{D.10})$$

and then it proceeds to calculate the values of the sufficient statistics as follows

$$\Phi'_{i1} = \sum_m \langle z_{im} \rangle \quad (\text{D.11})$$

$$\Phi'_{i2} = \sum_m \langle z_{im} \rangle \mathbf{x}_m \quad (\text{D.12})$$

$$\Phi'_{i3} = \sum_m \langle z_{im} \rangle \mathbf{x}_m \mathbf{x}_m^T. \quad (\text{D.13})$$

The above two steps are identical to the corresponding steps in batch EM, except that instead of using the whole dataset, only a minibatch is used.

The main difference between batch and online EM is in the update of the parameters. The previously calculated minibatch statistics cannot be used like this, since they are only a stochastic approximation of the true statistics. Instead, a running average of statistics is maintained, which is updated with the minibatch statistics as follows

$$\begin{pmatrix} \Phi_{i1} \\ \Phi_{i2} \\ \Phi_{i3} \end{pmatrix} \leftarrow (1 - \alpha) \begin{pmatrix} \Phi_{i1} \\ \Phi_{i2} \\ \Phi_{i3} \end{pmatrix} + \alpha \begin{pmatrix} \Phi'_{i1} \\ \Phi'_{i2} \\ \Phi'_{i3} \end{pmatrix}. \quad (\text{D.14})$$

Here,  $0 < \alpha < 1$  is the learning rate. The lower the value of  $\alpha$  is, the more biased towards the previous value of the average statistics the algorithm is. Using these average statistics, the parameters  $\pi_i$ ,  $\mathbf{m}_i$  and  $\mathbf{S}_i$  can be updated as in Equations (D.7)–(D.9).

The update schedule of the learning rate is highly important for the stability and convergence of the algorithm. In general, the learning rate should be high in the beginning, and it should decay towards zero during training. Let  $\alpha_t$  be the learning rate in the  $t^{\text{th}}$  iteration. Following

standard results in the stochastic approximation literature (Robbins and Monro, 1951), a learning rate schedule satisfying

$$\sum_t \alpha_t = \infty \quad \text{and} \quad \sum_t \alpha_t^2 < \infty \quad (\text{D.15})$$

is sufficient to guarantee convergence. For instance, a schedule that satisfies the above is the following

$$\alpha_t = \frac{1}{(t_0 + t)^\beta} \quad \text{where} \quad t_0 \geq 0 \quad \text{and} \quad \frac{1}{2} < \beta \leq 1. \quad (\text{D.16})$$

In practice however, other schedules are sometimes used, such as linear or logarithmic decay, that do not necessarily guarantee convergence but work well in practice.

# Appendix E

## Notations and Symbols

|  |  |
|--|--|
| $x, y, z, X, Y, Z$                                 | Scalars (lowercase or uppercase letters)   |
| $\mathbf{x}, \mathbf{y}, \mathbf{z}$               | Vectors (lowercase bold letters)   |
| $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$               | Matrices (uppercase bold letters)  |
| $\mathbf{x}^T, \mathbf{X}^T$                       | Transpose of vector $\mathbf{x}$ and matrix $\mathbf{X}$                                   |
| $\mathbf{I}$                                       | The identity matrix of appropriate size  |
| $\{\dots\}$  | A set, depending on context  |
| $\log(\cdot)$                                      | The natural logarithm (base $e$ )  |
| $\ \cdot\ $  | $\ell_2$ vector norm   |
| $\frac{\partial f}{\partial \mathbf{x}}$           | Derivative of function $f$ with respect to $\mathbf{x}$                                    |
| $\sigma(\cdot)$                                    | Logistic sigmoid function  |
| $\delta(\cdot)$                                    | Delta function   |
| $\mathcal{R}\{\cdot\}$                             | R operator for Hessian-vector products   |
| $\langle \cdot \rangle_{p(\mathbf{x})}$            | Expectation with respect to distribution $p(\mathbf{x})$                                   |
| $\mathcal{N}(\mathbf{x}   \mathbf{m}, \mathbf{S})$ | Gaussian distribution over $\mathbf{x}$ with mean $\mathbf{m}$ and covariance $\mathbf{S}$ |
| $D_{\text{KL}}(p \  q)$                            | KL divergence from distribution $p$ to distribution $q$                                    |
| $\mathcal{O}(\cdot)$                               | Asymptotic big O notation  |

## Appendix F

### List of Abbreviations

|       |  |
|-------|--|
| EM    | Expectation-Maximization                             |
| EP    | Expectation Propagation                              |
| GPU   | Graphics Processing Unit                             |
| KL    | Kullback-Leibler                                     |
| MCMC  | Markov Chain Monte Carlo                             |
| MNIST | Mixed National Institute of Standards and Technology |
| MoG   | Mixture of Gaussians                                 |
| NADE  | Neural Autoregressive Distribution Estimator         |
| pdf   | probability density function                         |
| RBM   | Restricted Boltzmann Machine                         |
| ReLU  | Rectified Linear Unit                                |

# Bibliography

- J. Ba and R. Caruana. Do deep nets really need to be deep? In *Advances in Neural Information Processing Systems 27*, pages 2654–2662. 2014.
- D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015.
- D. Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.
- S. Becker and Y. Le Cun. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 Connectionist Models Summer School*, pages 29–37, 1989.
- Y. Bengio. Learning deep architectures for AI. *Foundation and Trends in Machine Learning*, 2(1):1–127, Jan. 2009.
- C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., 2006.
- O. Bojar, C. Buck, C. Federmann, B. Haddow, P. Koehn, J. Leveling, C. Monz, P. Pecina, M. Post, H. Saint-Amand, R. Soricut, L. Specia, and A. Tamchyna. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the 9th Workshop on Statistical Machine Translation*, pages 12–58, June 2014.
- L. Bottou. On-line learning in neural networks. chapter On-line Learning and Stochastic Approximations, pages 9–42. Cambridge University Press, 1998.
- L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- C. Bucilă, R. Caruana, and A. Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 535–541, Aug. 2006.
- O. Cappé and E. Moulines. On-line expectation-maximization algorithm for latent data models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 71(3):593–613, 2009.
- R. Caruana, A. Niculescu-Mizil, G. Crew, and A. Ksikes. Ensemble selection from libraries of models. In *Proceedings of the Twenty-first International Conference on Machine Learning*, July 2004.
- D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, June 2012.
- A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B*, 39:1–38, 1977.
- I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, pages 2672–2680. 2014.

- A. Graves, A.-R. Mohamed, and G. E. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.
- A. Griewank. Who invented the reverse mode of differentiation? *Optimization Stories*, Documenta Mathematica:389–400, 2012.
- N. Heess, D. Tarlow, and J. Winn. Learning to pass expectation propagation messages. In *Advances in Neural Information Processing Systems 26*, pages 3219–3227. 2013.
- G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, Aug. 2002.
- G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, July 2006.
- G. E. Hinton, L. Deng, D. Yu, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov. 2012.
- G. E. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv:1503.02531v1*, Mar. 2015.
- A. Hyvärinen. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6:695–709, Dec. 2005.
- A. Hyvärinen. Some extensions of score matching. *Computational Statistics and Data Analysis*, 51(5):2499–2512, Feb. 2007.
- M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233, 1999.
- D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.
- A. Korattikara, V. Rathod, K. P. Murphy, and M. Welling. Bayesian dark knowledge. *arXiv:1506.04416v2*, July 2015.
- H. Larochelle and I. Murray. The Neural Autoregressive Distribution Estimator. *JMLR: W&CP*, 15:29–37, 2011.
- Y. Le Cun, C. Cortes, and C. J. C. Burges. MNIST handwritten digit database. <http://yann.lecun.com/exdb/mnist/>. Accessed on 20 July 2015.
- Y. Le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems 2*, pages 598–605. 1990.
- J. Li, R. Zhao, J.-T. Huang, and Y. Gong. Learning small-size DNN with output-distribution-based criteria. In *Proceedings of Interspeech*, Sept. 2014.
- P. Liang and D. Klein. Online EM for unsupervised models. In *Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics*, pages 611–619, 2009.
- D. J. C. MacKay. The evidence framework applied to classification networks. *Neural Computation*, 4(5):720–736, Sept. 1992.
- D. J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, 2002.
- X.-L. Meng and W. H. Wong. Simulating ratios of normalizing constants via a simple identity: A theoretical exploration. *Statistica Sinica*, 6:831–860, 1996.

- T. P. Minka. Expectation propagation for approximate Bayesian inference. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, UAI '01, pages 362–369, 2001.
- T. P. Minka. EP: A quick reference. Feb. 2008.
- K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.
- I. Murray. *Advances in Markov chain Monte Carlo methods*. PhD thesis, Gatsby computational neuroscience unit, University College London, 2007.
- V. Nair and G. E. Hinton. Rectified linear units improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning*, pages 807–814, June 2010.
- R. M. Neal. Probabilistic inference using Markov chain Monte Carlo methods, Sept. 1993.
- R. M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag New York, Inc., 1996.
- R. M. Neal. Slice sampling. *Annals of Statistics*, 31:705–767, 2000.
- R. M. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2):125–139, 2001.
- R. M. Neal. Estimating ratios of normalizing constants using linked importance sampling. *arXiv:math/0511216v1*, Nov. 2005.
- R. M. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 54: 113–162, 2010.
- B. A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160, Jan. 1994.
- S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning*, pages 833–840, June 2011.
- H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, Sept. 1951.
- A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. FitNets: Hints for thin deep nets. *arXiv:1412.6550v4*, Mar. 2015.
- O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, pages 1–42, Apr. 2015.
- R. Salakhutdinov and I. Murray. On the quantitative analysis of Deep Belief Networks. In *Proceedings of the 25th Annual International Conference on Machine Learning*, pages 872–879, 2008.
- L. F. Shampine. Vectorized adaptive quadrature in MATLAB. *Journal of Computational and Applied Mathematics*, 211(2):131–140, Jan. 2008.
- P. Simard, B. Victorri, Y. Le Cun, and J. Denker. Tangent prop—a formalism for specifying selected invariances in an adaptive network. In *Advances in Neural Information Processing Systems 4*, pages 895–903. 1992.
- P. Smolensky. Parallel distributed processing: Explorations in the microstructure of cognition. volume 1, chapter Information Processing in Dynamical Systems: Foundations of Harmony Theory, pages 194–281. MIT Press, 1986.
- E. Snelson and Z. Ghahramani. Compact approximations to Bayesian predictive distributions. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 840–847, 2005a.

- E. Snelson and Z. Ghahramani. Variational Bayes for predictive distributions, 2005b.
- N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, Jan. 2014.
- I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112. 2014.
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- B. Uria, I. Murray, and H. Larochelle. RNADE: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems 26*, pages 2175–2183. 2013.
- B. Uria, I. Murray, and H. Larochelle. A deep and tractable density estimator. In *Proceedings of the 31th International Conference on Machine Learning*, June 2014.
- M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1–2):1–305, Jan. 2008.
- M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the International Conference on Machine Learning*, 2011.
- J. Winn and C. M. Bishop. Variational message passing. *Journal of Machine Learning Research*, 6:661–694, 2005.
- L. Younes. On the convergence of Markovian stochastic algorithms with rapidly decreasing ergodicity rates. In *Stochastics and Stochastics Models*, pages 177–228, 1999.
- M. D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv:1212.5701v1*, Dec. 2012.

# Index

ADADELTA, 14, 17, 18, 57

Auto-encoder, 26

Contractive auto-encoder, 26

Backward propagation, 11, 56, 73, 79

Bayes rule, 48

Bayesian dark knowledge, 46

Bayesian inference, 28, 29, 48

Cross entropy, 10, 14, 15, 37, 78

Derivative square error, 15, 37

Deterministic approximate inference, 46

Discriminative model, 9

Dot product, 78

Dropout, 9

EM, 82

Batch EM, 82

Online EM, 83

Ensemble, 9, 14

EP, 45, 47

FitNets, 25

Forward propagation, 73

Generative adversarial training, 67

Generative model, 17, 48

Graphical model, 48

KL divergence, 15, 30, 49, 62

Knowledge distillation, 6, 25, 69

Layer

Linear, 75

Logistic, 76

Probit, 76

Rectified linear, 76

ReLU, 14

Softmax, 14, 77

Likelihood, 29, 48

Logit, 24, 25

Machine learning, 6

Maximum likelihood, 49

MCMC, 29

MNIST dataset, 14

Model comparison, 48

Model compression, 7, 9, 24

MoG, 31, 82

Monte Carlo, 28

NADE, 17, 54, 79

Neural network, 9, 14, 42, 73

Bayesian neural network, 46

Optimal brain damage, 23

Partition function, 48, 62

Posterior distribution, 29, 48

Predictive distribution, 29

Prior distribution, 29, 48

R technique, 13, 53, 72, 74

R{backprop}, 56, 74, 80

Random forest, 9

Ratio matching, 51

RBM, 53

Sampling

Ancestral sampling, 56, 62

Annealed importance sampling, 57, 62

Bridge sampling, 64

Gibbs sampling, 54

Importance sampling, 64

Slice sampling, 32

Score matching, 26, 51

Square error, 10, 12, 50, 78

Stochastic gradient, 10, 52

Stochastic gradient descent, 14, 17

Tangent prop, 26

Variational free energy, 50, 66

Variational inference, 46, 66

Variational message passing, 47