Cyprus University of Technology

Faculty of Engineering and Technology

Doctoral Dissertation

Bayesian Inference Techniques for Deep Learning

Charalampos Partaourides

Limassol, January 2018

CYPRUS UNIVERSITY OF TECHNOLOGY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING, COMPUTER ENGINEERING AND INFORMATICS

Doctoral Dissertation

Bayesian Inference Techniques for Deep Learning

Charalampos Partaourides

Limassol, January 2018

Approval Form

Doctoral Dissertation

Bayesian Inference Techniques for Deep Learning

Presented by

Charalampos Partaourides

Supervisor: Dr. Sotirios Chatzis, Assistant Professor

Signature ____

Member of the committee: Dr. Elpida Keravnou-Papailiou, Professor

Signature ____

Member of the committee: Dr. Theodora Varvarigou, Professor

Signature ____

Cyprus University of Technology

Limassol, January 2018

Copyrights

Copyright© 2018 Charalampos Partaourides

All rights reserved.

The approval of the dissertation by the Department of Electrical Engineering, Computer Engineering and Informatics does not imply necessarily the approval by the Department of the views of the writer.

I would like to thank ....

my supervisor Dr. Sotirios Chatzis without whom nothing would have been possible, my friends and family that still don't know what i do, and also the people that constitute the Cyprus University of Technology; the teachers and the students, the academic and the administrative staff. All have provided me with the constructive friction needed to cherish the PhD experience.

A few words ...

Obtaining a PhD is a daunting task. It is a treacherous journey that, more often that it should, feels like it will never end. It may be bleak to reconcile that what is produced with so much effort may be obsolete in 20 years time but that is not what the PhD has essentially offered. An undergraduate degree's ultimate goal is to provide you with a coherent thinking process in order to evaluate and judge the things we all take for granted. In contrast, a PhD's goal is to provide you with the imagination tools in order to derive new things. These may be or not be the future status quo.

As researchers that is the risk we take. Nobody knows what the future will bring and in my opinion it is better not to completely know. A partially observable environment may be surrounded with uncertainty but that uncertainty is an excellent motivator for exploration.

ABSTRACT

Deep learning has achieved state of the art performance in various challenging machine learning tasks pushing the Artificial Intelligence frontier into new heights. Tasks like object recognition, speech perception, language understanding and robotics are improving year by year. This is mainly due to the recent breakthroughs in Bayesian inference, the increased volume of datasets and the increased computational power. These make it feasible to tractably train these challenging hierarchical structured models that contain millions of parameters.

Deep Learning is an umbrella term which entails numerous deep architecture models that are able to capture even the most complex dynamics of the environment. Typically, they are trained under the maximum likelihood estimation paradigm. Unfortunately, in many real world tasks the high dimensionality of the observations results in even the largest datasets to being sparse. As such, there is an immense need for the training algorithm to compensate the uncertainty introduced by the data sparsity, overcome the model's overfitting tendencies and in result generalize well.

The statistical method of Bayesian inference provides a mathematically coherent way of dealing with data sparsity and overfitting. It essentially uses the Bayes theorem to accumulate evidence-based knowledge. This is achieved by postulating probability distributions over the parameters instead of trying to derive point estimates of them. Under the Bayesian view, we impose a prior distribution that encapsulates our initial belief about the model's dynamics and we correct that belief as we are presented with more data; this consists in inferring the posterior distribution. It is conspicuous that the choice of the distribution heavily controls the expressiveness of the model.

In this thesis, we present innovative approaches to train deep networks by considering sparsity, skewness and heavy tails on the form of the parameters distribution. Specifically, among our contributions, we impose a sparsity inducing distribution over the network synaptic weights to improve generalization. On a different vein, we consider the imposition of a skew normal distribution over the latent variables to increase the deep networks capacity. In parallel, we examine the efficacy of inferring the feature functions by devising a novel random sampling rational combined by an optimizable sample weighting scheme. The models derived by the aforementioned approaches are trained by means of approximate Bayesian inference scheme to allow for scalability in large datasets. We exhibit the advantages of these methods over existing approaches by conducting an extensive experimental evaluation using benchmark

datasets.

Keywords: Deep Learning, Machine Learning, Bayesian Inference, Variational Bayes, Regularization

TABLE OF CONTENTS

ABSTRACT . . . . **ix**

TABLE OF CONTENTS . . . . **xi**

LIST OF TABLES . . . . **xiv**

LIST OF FIGURES . . . . **xv**

LIST OF ABBREVIATIONS . . . . **xvi**

LIST OF PUBLICATIONS . . . . **xvii**

1 Introduction **1**

1.1 Deep Neural Networks 5

1.2 Deep Learning 9

1.3 Generative Models 11

1.4 Deep Network Regularization 12

2 Bayesian Inference **15**

2.1 Variational Bayes 17

2.2 Sampling Methods 19

2.3 Model Selection 22

3 Deep Network Regularization via Bayesian Inference of Synaptic Connectivity **23**

3.1 Introduction 24

3.2 Theoretical Background 25

3.2.1 DropConnect 25

3.2.2 Black-Box Variational Inference 27

3.3 Proposed Approach 28

3.3.1 Training DNNs with DropConnect++ layers 29

3.3.2 Feedforward computation in DNNs with DropConnect++ layers . . . 31
3.4 Experimental Evaluation 32
3.4.1 Computational complexity 35
3.4.2 Further investigation 36
3.5 Conclusions 37

4 Asymmetric Deep Generative Models 39
4.1 Introduction 40
4.2 Theoretical Foundation 42
4.2.1 Variational Auto-Encoder 42
4.2.2 The rMSN distribution 43
4.2.3 Skip Deep Generative Models 45
4.3 Proposed Approach 48
4.4 Expiremental Evaluation 51
4.4.1 Workflow recognition dataset 52
4.4.2 Honeybee dance dataset 53
4.4.3 Yearly song classifciation using audio features 54
4.4.4 Image classifcation benchmarks 55
4.4.5 A note on computational complexity 56
4.5 Conclusions 57

5 Deep Learning with t-Exponential Bayesian Kitchen Sinks 59
5.1 Introduction 59
5.2 Methodological Background 62
5.2.1 Weighted Sums of Random Kitchen Sinks 62
5.2.2 The Student's-*t* Distribution 63
5.2.3 The *t*-Divergence 64
5.3 Proposed Approach 65
5.3.1 Model Formulation 65
5.3.2 Model Training 66
5.3.3 Inference Algorithm 70
5.4 Experimental Evaluation 70
5.4.1 Comparative Results 72
5.4.2 Further Investigation 74

5.4.3 Are t-Exponential Bayesian Kitchen Sinks More Potent Than Random Kitchen Sinks? 74
5.4.4 Computational Complexity 75
5.5 Conclusions 76

6 Future Endeavors 77
6.1 Generative Adversarial Networks 78
6.2 Neural Attention 79
6.3 Deep Q-networks 79

REFERENCES 81

APPENDIX I 97

LIST OF TABLES

3.1 Predictive accuracy (%) of the evaluated methods. 33
3.2 Computational complexity (sec) per iteration at training time ($L = 1$). 33
3.3 Variation of the predictive accuracy (%) of the MC-driven approach (Eq:3.21) with the number of MC samples. 35

4.1 Activity recognition experiments: Test error (%) of the evaluated methods (means and standard deviations over multiple repetitions). 54
4.2 Song classification experiments: Test error (%) of the evaluated methods (means and standard deviations over multiple repetitions). 55
4.3 Image classification benchmarks: Test error (%) of the evaluated methods (means and standard deviations over multiple repetitions). 56

5.1 Obtained performance for best model configuration (the lower the better) . . . 71
5.2 D*t*BKS performance when replacing *t*-Exponential Bayesian Kitchen Sinks with Random Kitchen Sinks. 75

LIST OF FIGURES

1.1 Hierarchical representations form low to high level abstractions 3
1.2 A simple neuron 6
1.3 Single layer neural network 6
1.4 Activation functions 8

3.1 Accuracy convergence 36
3.2 Inferred posterior probabilities, $\tilde{\pi}$. 37

5.1 Univariate Student's-t distribution $t(y_t;\mu,\Sigma,\nu)$, with μ, Σ fixed, for various values of ν [176]. 64
5.2 Graphical illustration of the configuration of one DtBKS model layer. 67
5.3 DtBKS performance fluctuation with the number of layers, L, and the output size of each hidden layer, η (as a fraction of input dimensionality, δ) 73

LIST OF ABBREVIATIONS

AsyDGM:	Asymmetric Deep Generative Models
AI:	Artificial Intelligence
AVI:	Amortized Variational Inference
BBVI:	Black Box Variational Inference
BP:	BackPropagation
cdf:	cumulative distribution function
DBMs:	Deep Boltzmann Machines
DBNs:	Deep Belief Networks
DGMs:	Deep Generative Models
DGP:	Deep Gaussian Process
DNNs:	Deep Neural Networks
DOP:	Degrees Of Freedom
DtBKS:	Deep t-exponential Bayesian Kitchen Sinks
ELBO:	Evidence Lower Bound
EM:	Expectation Maximization
FA:	Factor Analysis
GD:	Gradient Descent
i.i.d.:	independent and identically distributed
KL:	Kullback Leibler (divergence)
MC:	Monte Carlo
ML:	Maximum Likelihood
pdf:	probability density function
RBF:	Radial Basis Function
RBMs:	Restricted Boltzmann Machines
ReLU:	Rectified Linear Unit
RKS:	Random Kitchen Sinks
rMSN:	restricted Multivariate Skew-Normal (distribution)
SBG:	Shannon Boltzmann Gibbs
SDGM:	Skip Deep Generative Model
SGD:	Stochastic Gradient Descent
VAEs:	Variational AutoEncoders

LIST OF PUBLICATIONS

Harris Partaourides and Sotirios P Chatzis. “Deep Network Regularization via Bayesian Inference of Synaptic Connectivity”. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining. Springer. 2017, pp. 30–41.

Harris Partaourides and Sotirios P Chatzis. “Asymmetric deep generative models”. In: Neurocomputing 241 (2017), pp. 90–96.

Harris Partaourides and Sotirios P. Chatzis. “Deep Learning with t-Exponential Bayesian Kitchen Sinks”. In:Expert Systems with Applications. Vol. 98. May 2018.

Charalambos Chrysostomou, Harris Partaourides, and Huseyin Seker. “Prediction of Influenza A virus infections in humans using an Artificial Neural Network learning approach”. In: Engineering in Medicine and Biology Society (EMBC), 2017 39th Annual International Conference of the IEEE. IEEE. 2017, pp. 1186–1189.

Chapter 1

Introduction

We are experiencing a period in human history where information is in abundance. The digital revolution has led to a monumental increase in information access, transmission, storage and computation. To provide a sense of the sheer volume of digital information, IBM estimated in 2013 to be 2.5 exabytes per day. Trying to model, analyze and extract information manually from this volume is an impossible task; thus we resort to systems and algorithms to distill knowledge from the raw data.

The first systems designed for modelling data were rule based systems, where an input was passed through a manual designed program to produce the desired output. These systems are exceptional at tasks where the problem can be efficiently described in rigid mathematical rules implemented under the ”if-then-else” paradigm. Unfortunately, as tasks become more complicated, the model dynamics become ambiguous making it infeasible to find rules and axioms analytically. Therefore, we resort to data driven approaches that can derive the dynamics in an automate matter.

Machine learning was introduced to tackle this type of problems [1]. Similarly, as humans learn from experience, machine learning aims to mimic that process. It uses models with adaptive parameters and learns to infer their optimal values based on observed data and relevant performance metrics. We typically require the algorithm to learn the data structure and regularities, encode them inside the parameters of the algorithmically developed mathematical model and finally use them to generate predictive outcomes of high accuracy and quality. Generalization capacity which is the quality of generating good outcomes on unseen data, is the utmost goal of any machine learning model.

To ensure generalization we must capture the data dynamics. In this context, extracting

effective representations that are distinctive and characteristic information of the raw data is significant; this is commonly known as feature extraction. The first machine learning algorithms relied on the manual extraction of features e.g single-layer perceptron [2], logistic regression [3], naive Bayes [4], kernel regression [5], support vector machines [6, 7, 8]. Selecting the correct distinctive attributes from the raw data to produce the features is a daunting process, requiring expert knowledge of the task at hand [9].

Feature learning (representation learning) [10] alleviates the need of ad hoc manual extraction by replacing it with an automated process. We effectively introduce a trainable processing layer between the model's input and output that process the observations and produces the features. This is beneficial not only for extracting special knowledge from the data but also for deriving the latent (hidden) dynamics of the task. The aforementioned dynamics are necessary for an efficient and expressive model. In effect, the output of the latent layer produces an abstract representation of the input.

Models with one layer of abstraction are considered shallow and were initially preferred [11]. The prominent choice for shallow models is the single layer neural network [12] which in theory under the universal approximation theorem [13, 14] can represent any desirable function given the appropriate number of units at the hidden layer (model's width). In practice however, as the task's complexity increases so does the required model width; this renders the optimization of the resulting increased number of parameters inefficient.

The theoretical and practical limitations of shallow techniques combined with the biological findings on how the human brain learns feature extraction, led to an important conclusion: Building equivalent systems requires models with deep architectures that involve several hidden layers of nonlinear processing structured hierarchically. For instance, the visual cortex solves the object recognition problem with a complex multilayer processing of the visual information in order to capture the spatio-temporal dynamics [15, 16, 17].

Deep Networks [18] is an umbrella term which entails numerous deep architecture models. As such, they comprise a huge number of trainable parameters integrated in many layers of neural networks with nonlinear dependence. These probabilistic directed models are able to capture even the most complex latent dynamics that exist beneath the observations. In each subsequent layer the model learns to extract a more abstract representation of the observations; thus learns features that range from coarse to task specific. For clarity in figure 1.1, we present an example of the learned hierarchical representation of a raw image in a trained deep convolutional network (The figure was taken from [19]).

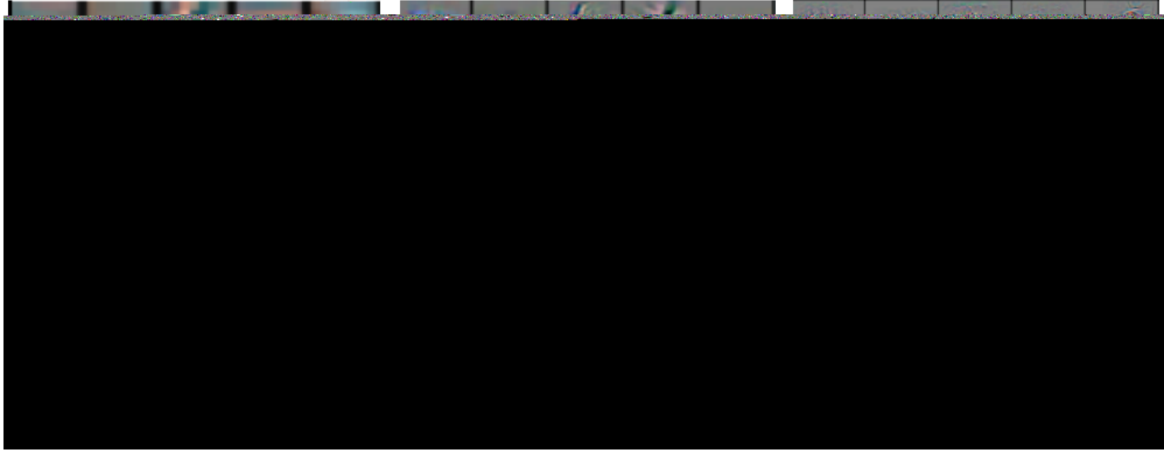


Figure 1.1: Hierarchical representations form low to high level abstractions

The automatic and abstract feature extraction process that exists in deep networks is typically required to have the transferability property. This effectively means that we want to reuse the relevant layers in similar tasks thus minimizing the model training procedure [20]. This property is extremely helpful in tasks where the labeled data are limited.

In general, hierarchical models do not have a unique estimator since they give rise to non convex objective functions of the training algorithm. As a result, deep networks have no unique solution thus, the same model trained under different initializations will converge to different estimators of each parameters. In addition, the non convexity of the model exhibits multiple local minima in the objective function, making maximum likelihood estimation [21] difficult. Furthermore, in many real world tasks the high dimensionality of the observations results even in the largest datasets to being sparse. As such, there is an immense need for the training algorithm to compensate the uncertainty introduced by the data sparsity, overcome the model's overfitting tendencies [22, 23] and in result generalize well.

The statistical method of Bayesian inference provides a mathematically coherent way of dealing with data sparsity and overfitting [24]. It essentially uses the Bayes theorem to accumulate evidence-based knowledge. This is achieved by postulating probability distributions over the parameters instead of trying to derive point estimates of them. Under the Bayesian view, we impose a prior distribution that encapsulates our initial belief about the model's dynamics and we correct that belief as we are presented with more data; this consists in inferring the posterior distribution. It is conspicuous that the choice of the distribution heavily controls the expressiveness of the model.

In the last years, deep learning has achieved state of the art performance in various challenging

machine learning tasks pushing the Artificial Intelligence frontier into new heights. Tasks like object recognition [25, 26, 27, 28, 29], speech perception [30, 31, 32, 33], language understanding [34, 35] and robotics [36, 37, 38] are improving year by year. In addition, they can make efficient use of the large supply of unlabeled sensory data [39, 40, 41, 42]. This is mainly due to the recent breakthroughs in Bayesian inference, the increased volume of datasets and the increased computational power. These make it feasible to tractably train these challenging hierarchical structured models that contain millions of parameters.

In addition, they have been assisted by software advancements reducing the development time for deep architectures. Software frameworks such as Tensorflow [43] and Caffe [44]. Including toolkits and wrappers on top of them such as Pylearn2 [45], Lasagne [46] and Keras [47]. The software packages and libraries allow for leveraging fast and extremely scalable parallelizations in modern GPU devices, in way transparent to the programming; thus making computationally feasible the training of even more complex mathematical models [48, 49, 50].

In this thesis, we present innovative deep learning approaches which tackle sparsity, skewness and heavy tails in the observed data and the underlying latent dynamics. One of our major contributions, consists in introducing a novel method for deep network regularization. Specifically, for the first time in the literature, we impose a sparsity inducing distribution over the network synaptic weights [51] for inferring a posterior distribution over the synaptic connectivity. This approach generalizes DropConnect [52], improves generalization capacity and is efficiently trained under the Bayesian inference paradigm by means of black-box variational inference algorithm [53]. On a different vein, we address the prevalence of asymmetric latent patterns in deep autoencoders by considering the imposition of a skew normal distribution over the latent variables [54]. This approach increases the deep networks capacity and is efficiently trained with the amortized variational inference algorithm [55]. In parallel, we examine the efficacy of inferring the feature functions of deep networks by devising a novel random sampling rational combined by an optimizable sample weighting scheme [56]; this approach was influenced by the Random Kitchen sinks [57] rationale. The models derived by the aforementioned approaches are trained by means of approximate Bayesian inference scheme to allow for scalability in large datasets. We exhibit the advantages of these methods over existing approaches by conducting an extensive experimental evaluation using benchmark datasets.

The remainder of the thesis is organized as follows: In the rest of Chapter 1, we present deep architectures in a more thorough context. In Chapter 2, we present the mathematical background of the thesis notably Bayesian inference fundamentals and Variational Bayes.

In Chapter 3, we present our innovative approach of regularization with sparsity inducing distributions. In Chapter 4, we expand the capacity of Deep Generative Models with our innovative approach that can capture asymmetric latent dynamics. In Chapter 5, we will present our alternative way of implementing the model's nonlinearities and extend it to capture latent dynamics with heavy tails. Finally in Chapter 6, we will provide the conclusions of the thesis accompanied with future endeavors and open research area topics.

1.1 Deep Neural Networks

Deep Neural Networks (DNNs) is an umbrella term which entails numerous deep architecture models, a variety of machine learning algorithms and even a set of divergent paradigms. They are primary designed to efficiently solve Artificial Intelligence (AI) tasks that entail learning from high dimensionality observations. In their simplest form, they consist of hierarchical neural networks structure with the ultimate goal to infer the function that accurately describes the data. In this context, discriminative models aim to learn the dependence form between the observed variables X and predictive variables Y.

The mathematical formulation of the discriminative model is:

$$Y = f(X; \boldsymbol{\theta}) \tag{1.1}$$

where $f()$ the function that describes the task and $\boldsymbol{\theta}$ the trainable parameters. In the case of a single layer neural network (1.1) becomes:

$$Y = f(X; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(X; \boldsymbol{\theta})^T \boldsymbol{w} \tag{1.2}$$

where $\phi(.)$ a non linear transformation, $\boldsymbol{\theta}$ the parameters of $\phi(.)$ and $\boldsymbol{w}$ the parameters that map the features to output. In case we are dealing with a multilayer neural network $\phi(.)$ becomes to:

$$\phi(X) = \phi^N(..(\phi^2(\phi^1(X)))) \tag{1.3}$$

where N the number of hidden layers i.e the depth of the model. In (1.3) we omitted the parameters θ that exist in each layer for clarity. A graphical representation of a simple neuron and a single layer neural network are shown at figure 1.2 and figure 1.3 respectively.

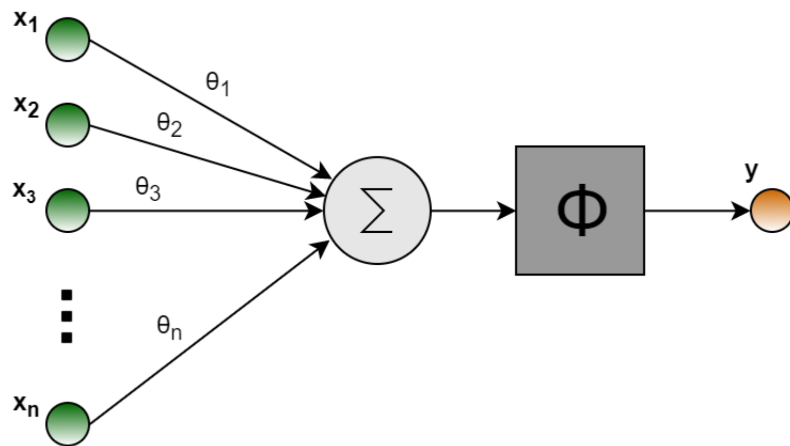


Figure 1.2: A simple neuron

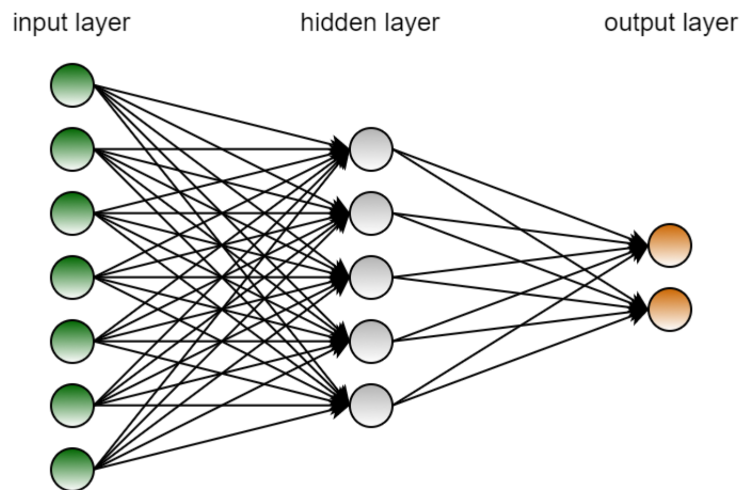


Figure 1.3: Single layer neural network

Constructing a simple DNN entails numerous configuration choices. The first we consider is the number of hidden layers. Adding layers into the architecture will result to a deeper model with an increased capacity; thus able to capture even more complex latent dynamics. Unfortunately, without a proper training algorithm and regularization tools, the model will fail to generalize. This is due to the increased number of trainable parameters which subsequently increases the model's overfitting tendencies. In addition, as we increase the number of layers we start to experience the vanishing/exploding gradient problem, where the gradients that are used to optimize the adaptive parameters get susceptible to numerical instability errors (overflow/underflow).

The next configuration choice one has to consider is the number of units that will comprise the networks layers. This choice in effect defines the width of the architecture. Considering that we are working with high dimensional data, a common approach is to gradually reduce the units in each subsequent layer, from the dimensionality of the input to the dimensionality of the output. This is reasonable if we recall that the features on each layer constitute an abstract representation of the layers input; thus the output dimensionality should be lower than the input dimensionality in each layer.

Another essential component of DNNs is the non linear feature functions. Constructing a deep network without feature functions (activation functions) is impossible, as will collapse the deep architecture to an equivalent shallow one. Typical choices are logistic sigmoid, hyperbolic tangent and rectified linear unit (ReLU) [58, 59]. The logistic sigmoid and hyperbolic tangent activation functions are closely related; both belong to the sigmoid family. A disadvantage of the sigmoid activation function is that it must be kept small due to their tendency to saturate with large positive or negative values. To alleviate this problem, practitioners have derived piecewise linear units like the popular ReLU [60] which are now the standard choice in deep learning research. In addition, ReLU is a viable option for the universal approximation theorem to hold [61] and even has task specific generalizations such as leaky ReLU [62], parametric ReLU [63] and Maxout units [64]. At figure 1.4 we illustrate some of the above functions. In this thesis, a novel construction choice with arbitrary nonlinearities will be presented in Chapter 5.

Another architecture choice is the connectivity between layers. The simplest choice is a dense layer where the units in each layer are fully connected. Alternative popular choices include but are not limited to convolutional layers for capturing spatial dependencies [25, 28, 65], recurrent neural networks for capturing temporal dependencies [66, 67, 68] and skip connections from layer to layer to facilitate gradient flow [69, 70].

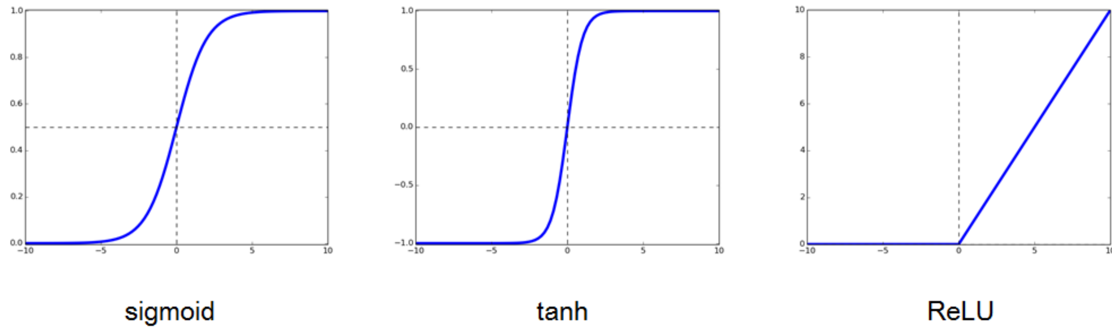


Figure 1.4: Activation functions

Finally, last but not least we must define the model output. This heavily depends on the task we are aiming to solve. Typically, when we are dealing with regression problems we resort to linear output layers:

$$y = W^T x + b \tag{1.4}$$

where W the weights and b the bias. In the case of binary classification we use the sigmoid function:

$$y = \sigma(W^T x + b) \tag{1.5}$$

where $\sigma()$ the sigmoid function. In the case of classification we use the softmax function:

$$y_j = \frac{e^{W_j^T x}}{\sum_{k=1}^{K} e^{W_k^T x}} \tag{1.6}$$

where the K the number of classes and j the specific class.

These choices conclude the model architecture and in effect the set of configuration choices. In addition, they give rise to the volume of trainable parameters and the associated objective function. We typically require the model to be large enough to be able to capture the hidden dependencies in the data. Once we have defined these architecture choices, we must estimate all the trainable parameters of the model. This is achieved, by optimizing some performance criterion with the use of the available training data. In the next section, we will present the backbone of Deep Learning.

1.2 Deep Learning

The trainable parameters of deep networks do not yield a unique estimator; this is due to the non convex nature of the employed objective functions. Indeed, training is performed by means of iterative optimization algorithms with proven convergence to a local optimum [71, 72, 73, 74]. One of the most popular choices to this end is gradient descent (GD) and its variations. GD is a first-order iterative optimization algorithm, for minimizing the objective function by means of the gradient of the objective function with respect to the parameters.

We have:

$$\theta = \theta - \eta \nabla_{\theta} J(\theta) \tag{1.7}$$

where θ the parameters, $J()$ the objective function and η the learning rate.

GD uses the gradient of all training examples in the dataset (batch) at each step to update the sought parameters. This is computationally demanding when we are dealing with huge datasets; an alternative that address this limitation is to update the parameters on each training example. The latter is known as stochastic gradient descent (SGD) [75]. A typical problem of SGD is that it produces high variance updates; hence, we typically resort to mini-batch GD where the parameters are updated by using a few tens of training examples. In deep networks, the gradients of the objective function with respect to the parameters are computed with the help of the backpropagation (BP) method [76]. GD does not guarantee to find the global optimum; this mainly due the saddle points of the objective function that make the optimizer susceptible to get stuck in suboptimal local minima.

A method that address this limitation is the proper initialization of the GD algorithm. A typical choice is the initialization with small random values. A prominent example is Glorot uniform [77] that samples from a uniform distribution where the sampling range depends on the number of input and output units of the layer. To facilitate convergence rate, a plethora of alternative schemes have been developed; these suggest different ways of selecting the GD's learning rate [78]. The most noteworthy examples are momentum [79] and Nesterov accelerated gradient [80] that use the gradient of the previous steps to accelerate the descent. Followed by Adagrad [81], Adadelta [82], RMSprop [83] and Adam [84] that use different learning rate for each parameter in addition to the gradient of the previous step. Furthermore, the latter techniques are less dependent to proper parameters initializations [85] due to their adaptive learning rates. In a different vein, it has been observed that adding random small

values on the gradient such as Gaussian noise helps the model's generalization capacity [86].

Achieving a strong generalization capacity is the ultimate goal in machine learning. This means that the trained model achieves accurate and comparable performance metrics in an independent dataset. To evaluate the generalization performance of a trained network, we typically split the complete dataset in training, testing and validation datasets. To alleviate the burden of a valid independent and identically distributed (i.i.d) split we resort to cross validation techniques like k-fold or Monte Carlo cross validation [87] where we train the network multiple times in different splits and compute the average performance. The training set is used for training the network parameters. The validation set is used for the selection of the hyperparameters of the model such as number of layers, units in each layer and initial learning rates. The test set is used for assessing the generalization performance and needs to be excluded from any parameter (or hyperparameter) value estimation.

A measure of performance in regression is:

$$MSE = \frac{1}{N}\sum_{n=1}^{N} ||\hat{y}^{(i)} - y^{(i)}||^2 \tag{1.8}$$

where y the target value and $\hat{y}$ the model's output. This is a common but rather simplistic metric since it assumes spherical distributed data. To relax this assumption one needs to define a predictive distribution that renders the deterministic model to a stochastic one. For instance, we may replace the linear output of Eq.(1.4) with:

$$p(y|x) = N(y|x^T W, \sigma^2 I) \tag{1.9}$$

where we postulate a Gaussian predictive density conditional on the input. Thus the model's output is the mean of a conditional multivariate Gaussian distribution with the variance σ^2 fixed to some constant. Under the maximum likelihood perspective, we derive the adaptive parameters values by maximizing the likelihood of observations given the parameters. The mathematical formulation of the maximum likelihood is:

$$\theta_{ML} = arg_{\theta}max[logP(X;\theta)] \tag{1.10}$$

where log is the natural logarithm; a monotonic transformation used to simplify the products to summations without affecting the parameter estimation process. The mathematical formulation of the conditional maximum log likelihood becomes:

$$\theta_{ML} = arg_{\theta} max[logP(Y|X;\theta)]$$
$$= arg_{\theta} max \sum_{n=1}^{N} logP(y^{(i)}|x^{(i)};\theta) \tag{1.11}$$

where the summation was introduced by considering i.i.d datapoints. The summation can be further analyzed as:

$$\sum_{i=1}^{N} \log P(y^{(i)}|x^{(i)};\theta) = -N\log\sigma - \frac{N}{2}\log(2\pi) - \sum_{i=1}^{N} \frac{||\hat{y}^{(i)} - y^{(i)}||^2}{2\sigma^2} \tag{1.12}$$

where N the number of examples. We observe that maximizing the log-likelihood with respect to the parameters yields the same result as does minimizing the mean square error when $\sigma = 1$.

Training DNNs under the maximum likelihood paradigm is effective but not sufficient. One of the reasons for searching better techniques for training DNNs is data sparsity, where we may have an abundance of data but those are located in a small part of the observable space. In addition, some machine learning tasks can only provide a limited amount of training data. This biases the model training procedure into capturing a suboptimal distribution of the data, hence failing to generalize well. To address this problem, we need to introduce statistical assumptions inside the hidden layers. There are three major directions where we can postulate statistical assumptions with probability density functions. The first is the layers outputs, the second is the trainable parameters and the last is the feature functions. These directions will be examined in the following Chapters.

1.3 Generative Models

In the previous sections, we talked about models that try to capture the discriminative dependency between the independent variable x and a predictable variable y. For instance, x could be a collection of animal images and y their correct class or x could be a recorded speech signal of phonemes and y the pronunciated phoneme. These are referred to as discriminative models and are typically trained with supervised learning that requires both variables x and y i.e. labeled training data. A different type of models with great importance are generative models [88, 89, 90].

Generative models goal is to model the phenomenon that generates the observations, thus capturing the phenomenon's hidden structure. Generative models have multiple practical

uses such as generating images [91] and modeling sequential data (e.g. human motion [92, 93]). In this context, the training algorithm attempts to learn the probability distribution by solely using the raw data. This is very convenient characteristic due to the unlimited supply of unlabeled data collected by a plethora of applications of the modern society. As we have already mentioned, deep networks entail a procedure of learning to extract hierarchical representations (features) of the input. Therefore, we can postulate a generative model which is parameterized via deep networks that will be trained to extract informative features with the use of unlabeled data.

In addition, semi-supervised learning [94] disentangles the representation learning and the classifier. Therefore, we can use a large dataset of observations to train a generative model and a small labeled dataset to train the simple classifier. This type of learning has shown great performance results [95, 96, 97] and alleviates the time consuming and expensive issue of an expert having to label huge datasets.

Generative models have a long history in machine learning, with the most promising approach being Deep Generative Models (DGMs) [98]. This fairly recent development started when Hinton et al. in 2006 introduced Deep Belief Networks (DBNs) [99] which were primarily based on Restricted Boltzmann Machines (RBMs) [100]. DBNs comprise of multiple layers of hidden variables trained efficiently with a greedy layer-by-layer learning algorithm. The algorithm is moderately fast and scales well to large sets of unlabeled data. DBNs were followed by Deep Boltzmann Machines (DBMs) [101] which is an undirected graphical model also based on RBM. Howerver, the most recent advance in the field with impressive results are the Variational Autoencoders (VAEs). A VAE model learns to extract a salient, lower representation of the input that is more useful and its training criterion reflects how well we can reconstruct the data.

1.4 Deep Network Regularization

As we have mentioned, the overfitting tendency of deep models brings to the fore the immerse need of regularization. Dealing with overfitting is the key ingredient in the effort of enabling good generalization capacity, and obtaining a model of comparable performance in training and test data. Overfitting rises from the model's incapability to account for uncertainty which is due to the epistemic nature of the available training data. In addition, not dealing with overfitting will lead to a model biased on the training set.

A simple approach to prevent overfitting would be to gradually increase the model parameters till we start to observe overfitting. This technique may be plausible in shallow networks but on deep networks is not a valid solution. This mainly due to the dependencies between layers, the activation functions and the optimizers that make impossible to derive the optimal number of parameters. In addition, this procedure will restrict the model's capacity in capturing dependencies.

Another approach is to increase the available training set. To augment the dataset requires human labor which is costly, lengthy and not always a possible procedure. Thus, a similar approach will be to generate synthetic data. The goal when creating synthetic data is to create variations that are as close as possible to the original data. This is refer to as data augmentation scheme and its process varies from problem to problem. For instance, the data augmentation schemes for tasks with image datasets consist in shifting the image, adding noise, or changing the background.

An alternative approach, is to halt the training process when starting to observe overfitting on a validation set. This is known as early stopping and even though the process minimizes the overfitting it often results in under trained networks. In addition, it does not address any of the reasons for overfitting and misuses precious data.

A more coherent way of dealing with the issue is weight penalty. It was constructed as a tool to force the network to favor simple explanations of the data, to more complex ones. It's inspired by the Occam's razor principle that states that the simplest explanation is usually the best. This is achieved by keeping the parameters value as small as possible by introducing a penalty term on the objective function that promotes small values for the parameters. These were the earliest form of regularizers. Common weight penalty techniques are: a) L1 regularization, that penalizes the absolute value of the parameter weights; thus resorts to sparse weights. b) L2 regularization, that penalizes the square values of the parameter weights. Unfortunately, weight penalty in deep networks exhibits a small impact on overfitting thus, we must resort to more advanced methods.

Dropout [102] is one of the most popular regularization techniques for deep networks. In essence, it consists in randomly dropping different units of the network on each iteration of the training algorithm. This way, only the parameters related to a subset of the network units are trained on each iteration; this effectively limits overfitting and increases the model's performance. It can be thought of as training numerous small dependent networks and also ensuring that all network parameters are effectively trained. In addition, by using the dropout

regularizer we also limit the co-adaptation of the parameters during the training procedure.

A typical network output is:

$$r = a(W \cdot u) \tag{1.13}$$

where r is the layer output vector, $a(\cdot)$ is the adopted activation function, W is the matrix of synaptic weights and u is the layer input vector. When dropout is applied, the network output units (1.13) become:

$$r = m \circ a(W \cdot u) \tag{1.14}$$

where $\circ$ is the element-wise product and m is a vector of binary variables (indicators) encoding the unit selection. They are drawn independently from $m_j \sim Bernoulli(p)$ where j the unit of the layer's output and p the probability of keeping the unit. Hence, Dropout effectively introduces dynamic sparsity within the model, specifically on the output vectors of the layers. Training of a Dropout layer begins by selecting an example u, and drawing a mask vector m from a Bernoulli(p) distribution to mask out elements of the Dropout layer. The parameters throughout the model can be updated via GD variants as usual. During testing, we do not draw samples of the Bernoulli distribution but scale the weights with the drop probability p. Furthermore, a generalization of Dropout, DropConnect [52], consists in dropping weights instead of units and will presented in detail in Chapter 3.

Finally, we would like to underline that training DNNs without considering overfitting will result to suboptimal performance. To alleviate this burden machine learning scientists have developed various techniques that have been consistently shown to be linked to Bayesian inference [103]. This insight have ignited an interest in developing regularizers that are more mathematically coherent [104, 105, 106]. To be precise, L1 regularization imposes a Laplace prior [107], L2 regularization a Gaussian prior [108] and the popular choice for DNNs, Dropout, enjoys links with Gaussian process [109]. In the next Chapter, we will present the mathematical background of Bayesian inference that will be used through out the thesis.

Chapter 2

Bayesian Inference

Statistical inference is a collection of algorithms and paradigms employed for estimating the statistical properties and the underlying probability distribution of the observed data. In this context, the observed data is assumed to be a subset of the observable space and constitutes a population from which we can infer salient characteristics. Thus, the hypothesis is that the characteristics of the entire population can be estimated from the available observations.

Bayesian inference is a paradigm of statistical inference under which we update a full probabilistic model as more data becomes available. Bayesian inference [24] has recently risen in popularity due to its successful use in deep networks among other machine learning models. It has overshadowed the frequentist inference paradigm whereby the maximum likelihood estimation is the prominent example. The key difference of the two paradigms lies in the nature of the sought parameters estimates. On the one hand, the frequentist approach treats them as fixed values that we seek to estimate i.e. point estimates. On the other hand, Bayesian inference treats them as random values, postulates a prior hypothesis (a probability distribution over them) and seeks to refine the assumption based on data; that is, obtain an appropriate posterior distribution over them.

Specifically, under the Bayesian perspective, we initially impose a fundamental assumption about the parameters distribution with the use of a prior distribution. This is essentially based on our assumption about the underlying data dynamics or some kind of expert knowledge. Then, we correct that belief as we are presented with more information by inferring the sought posterior distribution. From a different perspective, we can claim that the Bayesian paradigm uses the available data to update our initial belief of our distribution, in contrast to the frequentist paradigm that determines some sufficient statistics which characterize the

sought parameters. Thus, Bayesian inference does not yield point estimates, instead it yields a full (posterior) probability distribution which encodes different perspectives of the model parameterization accompanied with the corresponding probability of relevance. Hence, can account for the epistemic uncertainty of the available observable data; that is, the uncertainty of the choice of model parameters and structure that best explain the data.

Consider a set of observations x, and the unknown set of parameters θ. Under the Bayes theorem the sought posterior distribution is:

$$p(\theta|x) = \frac{p(x|\theta)p(\theta)}{p(x)} \tag{2.1}$$

where $p(\theta|x)$ the posterior distribution; this is the estimated probability of the parameters given the observations. In addition, $p(x|\theta)$ is the likelihood function which represents the probability of the observations given the parameters. Furthermore, $p(\theta)$ is the prior distribution that encapsulates our initial belief about the parameters distribution. Finally, $p(x)$ is the model evidence, also known as the marginal likelihood that can be expanded as:

$$p(x) = \int p(x|\theta)p(\theta)d\theta \tag{2.2}$$

Typically, in order to allow for modeling complex data dynamics we need to introduce additional data dependencies on some unobserved latent variable z. Under such an assumption the marginal likelihood is expressed conditional on the latent variable z. Furthermore, we need to impose a prior distribution over the latent variable, $p(z)$ and seek to derive $p(z|x)$. This renders the posterior distribution to:

$$p(z,\theta|x) = \frac{p(x|z,\theta)p(z,\theta)}{p(x)} = \frac{p(x|z,\theta)p(z)p(\theta)}{p(x)} \tag{2.3}$$

While for the marginal likelihood, we have:

$$p(x) = \int p(x,z,\theta)dzd\theta = \int p(x|z,\theta)p(z)p(\theta)dzd\theta \tag{2.4}$$

A typical issue with Bayesian inference is an intractable marginal likelihood. This issue rises even in cases of moderately complicated models. In essence, for the case of continuous variables, the required integrations may not have closed-form analytical solutions. While for discrete variables, the marginalizations which involve summing over all possible configurations are prohibitively demanding. In these cases, we resort to approximations. There

are primarily two types of approximations; stochastic and deterministic. Stochastic approximations use sampling methods to compute the intractable integrals and are proven to be cable of obtaining satisfactory approximations under mild conditions. In practice however, they are computationally demanding which limits their use to small scale problems. On the other hand, deterministic approximations e.g. Taylor expansions, use simplifying assumptions to ameliorate the intractabilities; this way we trade accuracy for linear computational time. Variational Bayes is the most popular approximate inference technique for deep networks.

2.1 Variational Bayes

Variational Bayes approximates the intractable posterior distribution with a simpler family of distributions that are analytically tractable; then, seek the distribution that minimizes a similarity measure. This consists in constructing a lower bound and maximizing it, instead of maximizing the marginal likelihood which is intractable. In essence, this renders the posterior estimation task, an optimization one.

Consider x the observed variables, z the unobserved variables and θ the parameters. We can rewrite the log marginal likelihood with the use of two variational distributions. We have:

$$\begin{aligned}\log p(x) &= \log \iint \frac{q(z)q(\theta)}{q(z)q(\theta)} p(x,z,\theta) dz d\theta \\ &= \log \iint \frac{q(z)q(\theta)}{q(z)q(\theta)} p(x|z,\theta)p(z)p(\theta) dz d\theta\end{aligned} \tag{2.5}$$

where $q(z)$ and $q(\theta)$ are the two new probability distributions that we will use to approximate the true distributions $p(z|x)$ and $p(\theta|x)$ respectively. Note that at this point, we have considered that the posterior distribution (2.3) factorizes over z and θ. This is called the mean field approximation, and it is widely used in the context of variational Bayesian inference in order to simplify the mathematical equations. On this basis, and making use of Jensen's inequality, specifically $\log E[x] \geqslant E[\log x]$ we have:

$$
\begin{aligned}
\log p(x) &\geqslant \iint q(z)q(\theta)\log\frac{p(x|z,\theta)p(z)p(\theta)}{q(z)q(\theta)}dzd\theta \\
&= \iint q(z)q(\theta)\log p(x|z,\theta)dzd\theta \\
&+ \iint q(z)q(\theta)\log\frac{p(z)}{q(z)}dzd\theta \\
&+ \iint q(z)q(\theta)\log\frac{p(\theta)}{q(\theta)}dzd\theta \\
&= E_{q(z)q(\theta)}[\log p(x|z,\theta)] \\
&+ \int q(z)\log\frac{p(z)}{q(z)}dz + \int q(\theta)\log\frac{p(\theta)}{q(\theta)}d\theta \\
&= E_{q(z)q(\theta)}[\log p(x|z,\theta)] - D_{KL}[q(z) \parallel p(z)] - D_{KL}[q(\theta) \parallel p(\theta)] \\
&= \mathscr{L}(q)
\end{aligned}
\tag{2.6}
$$

where $\mathscr{L}(q)$ the evidence lower bound (ELBO) that we seek to maximize. This bound is often referred to as the free energy of the model. Then, the marginal likelihood can be rewritten as:

$$
\log p(x) = \mathscr{L}(q) + D_{KL}[q(z) \parallel p(z)] + D_{KL}[q(\theta) \parallel p(\theta)] \tag{2.7}
$$

where D_{KL} the Kullback-Leibler divergence between two distributions. The divergence is always positive and equals to zero if and only if the two distributions match. In essence, minimizing the divergence maximizes the lower bound. Thus the goal is now finding a tighter bound to the real distribution that is scalable to larger applications.

In case the postulated model is conjugate, this maximization can be performed by means of simple moment matching. However, the need of imposing conjugate priors limits the expressiveness of the model and its ability to capture complex distributions that often govern real world scenarios.

On the other hand, non-conjugate models can be treated under the variational inference paradigm by means of deriving a further lower bound to the ELBO by means of using ad hoc approximation as suggested in [110, 111, 112]. Model treated liked this include Bayesian logistic regression [113], Bayesian generalized linear models, discrete choice models [114], Bayesian item response models [115] and non-conjugate topic models [116]. In addition, Wang and Blei [117] developed two extensions to Mean-Field Variational Inference that can be applied to a wider range of non-conjugate models, namely Laplace Variational Inference and Delta method variational inference. The former makes appropriate use of

Laplace approximations which is a second order Taylor expansion around the mode of the sought distribution, while the later a first order Taylor expansion.

The above solutions require model specific analysis that demand expert mathematical knowledge. Thus, the capability of automating the procedure of variational inference is of utmost importance. Black-Box Variational Inference [53] and Amortized Variational Inference [55] are efficient training algorithms aimed at that direction. They use differentiable unbiased estimators with the help of computationally inexpensive sampling methods. The latter two inference procedures will be explained and used in the following chapters of this thesis in order to efficiently train our innovative models.

2.2 Sampling Methods

Sampling methods are very important tools in machine learning. They are used in cases where we want to draw samples from a distribution, approximate an intractable integral, an expensive summation or a mathematical expectation. Monte Carlo (MC) sampling uses random samples and its allure is due to the law of large numbers. This law dictates that as we continue to sample from the probability distribution the average will converge to the theoretical mean (expected value). In this context, the goal of sampling is to obtain satisfactory approximations with as a limited number of samples as possible.

In this section, we will present common sampling methods used in deep networks. Recall that a common issue of deep networks is that they have intractable exact inference. Specifically, the expectation of the function $f(z)$ with respect to the posterior distribution $p(z|x)$, cannot be computed analytically due to its complexity. Consider a simple case of MC sampling where we require the evaluation of the following expectation:

$$E[f] = \int p(z)f(z) \tag{2.8}$$

To compute the expectation we draw independent samples from $p(z)$ and compute the empirical average. Thus, we have:

$$E[f] = \lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(z^{(i)}) \tag{2.9}$$

The above estimator is unbiased with a bounded variance that decreases as we increase the

samples. The problem however, lies in drawing samples that make significant contributions to the sum. In addition, the mass of probability distributions of interest often lie in small regions of z space, thus uniform sampling will under estimate the expectation. We ideally want to draw samples that fall in regions where $p(z)f(z)$ is large.

A common characteristic of deep probabilistic models is that it is impractical to draw independent and significant samples from the distribution $p(x)$ but, we can easily evaluate the distribution for any value of z. Influenced by this, rejection sampling uses a proposal distribution $q(z)$ that we can easily draw samples from. In addition, the samples are enlarged with a constant k so that are a better fit to the sought distribution, $kq(z) \geq p(z)$. To apply rejection sampling, we first draw sample z_0 from the proposal distribution. Then we draw a sample u_0 from the uniform distribution of range $[0, kq(z_0)]$ which we compare with the evaluation of the original distribution with the sample z_0. If $u_0 > p(z_0)$ we reject the z_0 sample, otherwise we retain it.

In essence, rejection sampling uses an envelope function to decide where to accept or reject the samples. The probability of rejected samples depends on the area between the desired distribution and the comparison function. Choosing an envelope function that minimizes the rejection rate is difficult under the rejection sampling scheme. Adaptive rejection sampling uses tangent lines to construct the envelope function. Specifically, when a sample is rejected a new tangent line is introduced that refines the envelope function. Adaptive rejection sampling is used on concave distributions, while a variant named adaptive rejection Metropolis sampling can be used for a more broad range of distributions.

A major problem with the above methods is that they are suited for low dimensional z. As the dimensionality increases the acceptance rate of samples diminishes leading to expensive sampling loop per datapoint. Importance sampling uses a different approach; it evaluates the expectation without drawing samples from the distribution $p(z)$. It is based on the fact that we can rewrite equation (2.8) with the assistance of a proposal distribution $q(z)$, as:

$$E[f] = \int \frac{q(z)}{q(z)} p(z) f(z) \tag{2.10}$$

Thus the importance sampling estimator is:

$$E[f] = \lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} \frac{p(z)^{(i)}}{q(z)^{(i)}} f(z^{(i)}) \tag{2.11}$$

with $\frac{p(z)^{(i)}}{q(z)^{(i)}}$ the importance weights. These quantities correct the bias introduced by the proposal distribution leading to an unbiased estimator with variance sensitive to the choice of the importance sampling distribution $q(z)$. Unfortunately, importance sampling also suffers from the curse of dimensionality. This is due to the fact that as the dimensionality increases finding a good proposal distribution is impossible; thus, it fails to generate significant samples.

In the case of multivariate distributions with conjugate nature, we resort to Gibbs sampling [118, 119]. The process involves choosing a new sample for each dimension separately from the others. Below we summarize the Gibbs sampling process:

1. Pick initial values $[z_i : i = 1, ..., N]$ at random
2. For i in range N:
   Sample z_i from $q(z_i | z_{\setminus i})$
3. Go to 2 until M steps generated

where $\setminus i$ implies all the variables except i.

Finally, Markov Chain Monte Carlo (MCMC) [120] is a more general approach that allows sampling from various distributions. As with importance sampling, we sample from a proposal distribution with the difference that the distribution depends on the current state $q(z^t | z^{t-1})$. In this context, consecutive samples are drawn in a way that tends to formulate a Markov chain whereby consecutive samples obey the first order transition dynamics. That is, the density of the sample drawn at the t iteration depends only on the value of the samples drawn at $t-1$. After a candidate sample is drawn, we accept or reject it according to an acceptance criterion. This is similar to rejection sampling with the difference that when a sample is rejected we stay on the same state.

Under this construction, as we take steps on the Markov chain, we will eventually reach a point where each update will leave the distribution invariant. This is known as the equilibrium distribution and the steps needed, mixing time. Taking samples from the equilibrium distribution is the same as drawing samples from the desired distribution with the difference that the samples are highly correlated. To alleviate this burden we often keep every m^{th} sample with m sufficiently large to alleviate the correlation. Metropolis-Hastings [121, 122] is one of the most popular and widely known MCMC algorithms. Below we summarize the process:

1. Pick an initial state z_0 at random $(t = 0)$

2. Randomly pick a new state z_t from $q(z_t|z_{t-1})$

3. Accept the state according to $A(z_t|z_{t-1})$

   For the symmetric proposal distribution, Metropolis choice is

$$A(z_{t+1}|z_t) = min(1, \frac{q(z_{t+1})}{q(z)}) \tag{2.12}$$

   For the nonsymmetric proposal distribution, Hastings choice is

$$A(z_{t+1}|z_t) = min(1, \frac{T(z_t|z_{t+1})q(z_{t+1})}{T(z_{t+1}|z_t)q(z)}) \tag{2.13}$$

4. Go to 2 until M states generated

2.3 Model Selection

Model selection is the process of selecting the best candidate from a collection of alternatives. Expressing model selection under Bayesian inference we have the posterior:

$$p(m_i|y) = \frac{p(y|m_i)p(m_i)}{p(y)} \tag{2.14}$$

where $p(m_i|y)$ is the posterior probability of the model given the available data, y. The first term in the equation's numerator $p(y|m_i)$ is the likelihood function, the second term $p(m_i)$ is the prior distribution of the model which expresses our intuition of how plausible the model is with respect to alternative ones, and the denominator $p(y)$ is the normalizing constant derived by marginalizing over the different possible models. Thus, the evidence function reads:

$$p(y) = \sum p(y|m_i)p(m_i) \tag{2.15}$$

Finding the best model can be simplified by considering the same prior for all alternative models. In such a case, selection consists in maximizing the likelihood function. However, this is a simplistic approach, hence using a full Bayesian treatment is preferable in most real world applications where epistemic uncertainty is an issue.

Chapter 3

Deep Network Regularization via Bayesian Inference of Synaptic Connectivity

Deep neural networks often require good regularizers to generalize well. Currently, state-of-the-art DNN regularization techniques consist in randomly dropping units and/or connections on each iteration of the training algorithm. Dropout and DropConnect are characteristic examples of such regularizers, that are widely popular among practitioners. However, a drawback of such approaches consists in the fact that their postulated probability of random unit/connection omission is a constant that must be heuristically selected based on the obtained performance in some validation set. To alleviate this burden, in this Chapter we regard the DNN regularization problem from a Bayesian inference perspective: We impose a sparsity-inducing prior over the network synaptic weights, where the sparsity is induced by a set of Bernoulli-distributed binary variables with Beta (hyper-)priors over their prior parameters. This way, we eventually allow for marginalizing over the DNN synaptic connectivity for output generation, thus giving rise to an effective, heuristics-free, network regularization scheme. We perform Bayesian inference for the resulting hierarchical model by means of an efficient Black-Box Variational inference scheme. We exhibit the advantages of our method over existing approaches by conducting an extensive experimental evaluation using benchmark datasets.

The aforementioned innovative approach dubbed as DropConnect++ has been published in the ”Pacific-Asia Conference on Knowledge Discovery and Data Mining” in 2017 under the title

”Deep Network Regularization via Bayesian Inference of Synaptic Connectivity”.

3.1 Introduction

In the last few years, the field of machine learning has experienced a new wave of innovation; this is due to the rise of a family of modeling techniques commonly referred to as deep neural networks (DNNs) [18]. DNNs constitute large-scale neural networks, that have successfully shown their great learning capacity in the context of diverse application areas. Since DNNs comprise a huge number of trainable parameters, it is key that appropriate techniques be employed to prevent them from overfitting. Indeed, it is now widely understood that one of the main reasons behind the explosive success and popularity of DNNs consists in the availability of simple, effective, and efficient regularization techniques, developed in the last few years [18].

Dropout has been the first, and, expectably enough, the most popular regularization technique for (dense-layer) DNNs [109]. In essence, it consists in randomly dropping different units of the network on each iteration of the training algorithm. This way, only the parameters related to a subset of the network units are trained on each iteration; this ameliorates the associated network overfitting tendency, and it does so in a way that ensures that all network parameters are effectively trained. In a different vein, [52] proposed randomly dropping DNN synaptic connections, instead of network units (and all the associated parameters); they dub this approach DropConnect. As showed therein, such a regularization scheme yields better results than Dropout in several benchmark datasets, while offering provable bounds of computational complexity.

Despite their merits, one drawback of these regularization schemes can be traced to their very foundation and rationale: The postulated probability of random unit/connection omission (e.g., dropout rate) is a constant that must be heuristically selected; this is effected by evaluating the network’s predictive performance under different selections of this probability, in some validation set, and retaining the best performing value. This drawback has recently motivated research on the theoretical properties of these techniques. Indeed, recent theoretical work at the intersection of deep learning and Bayesian statistics has shown that Dropout can be viewed as a simplified approximate Bayesian inference algorithm, and enjoys links with Gaussian process models under certain simplistic assumptions [123, 109].

These recent results formed our main motivation behind DropConnect++. Specifically, the

main question the method aims to address is the following: Can we devise an effective Deep network regularization scheme, that marginalizes over all possible configurations of network synaptic connectivity (i.e., active synaptic connections), with the posterior over them being inferred from the data? To address this problem, for the first time in the literature, we regard the DNN regularization problem from the following Bayesian inference perspective: We impose a sparsity-inducing prior over the network synaptic weights, where the sparsity is induced by a set of Bernoulli-distributed binary variables. Further, the parameters of the postulated Bernoulli-distributed binary variables are imposed appropriate Beta (hyper-)priors over their prior parameters.

Under this hierarchical Bayesian construction, we can derive appropriate posteriors over the postulated binary variables, which essentially function as indicators of whether some (possible) synaptic connection is retained or dropped from the network. Once these posteriors are obtained using some available training data, prediction can be performed by averaging (under a Bayesian inference sense) over multiple (posterior) samples of the network configuration. This inferential setup constitutes the main point of differentiation between our approach and DropConnect. We finaly derive an efficient inference algorithm for our model by resorting to the Black-Box Variational Inference (BBVI) [53].

The remainder of this Chapter is organized as follows: In the following section, we present the theoretical background; DropConnect and the inferential framework that will be used in the context of the proposed approach, namely BBVI. Next, we introduce our approach, and derive its inference and prediction generation algorithms. Then, we perform an extensive experimental evaluation of our approach, and compare to popular (dense-layer) DNN regularization approaches, including Dropout and DropConnect. To this end, we consider a number of well-established benchmarks in the related literature. Finally, in the concluding section, we summarize our contribution and discuss our results.

3.2 Theoretical Background

3.2.1 DropConnect

DropConnect [52] is another form of regularization for dealing with overfitting of deep networks that was inspired from Dropout. In essence, is a generalization of Dropout to the full connection structure of a layer. The core difference consists in the fact that Dropout imposes

sparsity on the output vectors of a layer, while DropConnect imposes sparsity on the synaptic weights W.

To make the generalization more clear lets observe the two regularizers more closely. When dropout is applied to the layer we consequently drop the bundle of synaptic weights that lead to the dropped unit. By contrast, DropConnect drops each synaptic weight independently.

As was presented in their paper, such a regularization scheme yields better results than Dropout in several benchmark datasets, while offering provable bounds of computational complexity.

When applied, the output of the network's layers is given by

$$\mathbf{r} = a((\mathbf{M} \circ \mathbf{W})\mathbf{u}) \tag{3.1}$$

where $\circ$ is the elementwise product, $a(\cdot)$ is the adopted activation function, **W** is the matrix of synaptic weights, **u** is the layer input vector, **r** is the layer output vector and **M** is a matrix of binary variables (indicators) encoding the connection information. They are drawn independently from $M_{ij} \sim Bernoulli(p)$ where i,j the weight correspondence and p the probability of keeping the weight.

Training of a DropConnect layer begins by selecting an example **u**, and drawing a mask matrix **M** from a Bernoulli(p) distribution to mask out elements of both the weight matrix and the biases in the DropConnect layer. The parameters throughout the model can be updated via stochastic gradient descent (SGD), or some modern variant of it, by backpropagating gradients of the postulated loss function with respect to the parameters. To update the weight matrix **W** in a DropConnect layer, the mask is applied to the gradient to update only those elements that were active in the forward pass. Additionally, when passing gradients down, the masked weight matrix $\mathbf{Z} \circ \mathbf{W}$ is used.

At section 3.3, we will describe our approach of dealing with the heuristic parameter p of the above regularization technique. In essence we provide a coherent Bayesian inference treatment of DropConnect alleviating the need to heuristically define the parameter's value by using a data driven approach. Our approach gives insights in the ongoing scientific research of full Bayesian approach of training deep networks.

3.2.2 Black-Box Variational Inference

In general, Bayesian inference for a statistical model can be performed either exactly, by means of Markov Chain Monte Carlo (MCMC), or via approximate techniques. Variational inference is the most widely used approximate technique; it approximates the posterior with a simpler distribution, and fits that distribution so as to have minimum Kullback-Leibler (KL) divergence from the exact posterior [124]. This way, variational inference effectively converts the problem of approximating the posterior into an optimization problem.

One of the significant drawbacks of traditional variational inference consists in the fact that its objective entails posterior expectations which are tractable only in the case of conjugate postulated models. Hence, recent innovations in variational inference have attempted to allow for rendering it feasible even in cases of more complex, non-conjugate model formulations. Indeed, recently proposed solutions to this problem consist in using stochastic optimization, by forming noisy gradients with Monte Carlo (MC) approximation. In this context, a number of different techniques have been proposed so as to successfully reduce the unacceptably high variance of conventional MC estimators. BBVI is one of these recently proposed alternatives, amenable to non-conjugate probabilistic models that entail both discrete and continuous latent variables.

Let us consider a probabilistic model $p(x,z)$ with observations x and latent variables z, as well as a sought variational family $q(z;\phi)$. BBVI optimizes an evidence lower bound (ELBO), with expression

$$\log p(x) \geq \mathscr{L}(\phi) = \mathbb{E}_{q(z;\phi)}[\log p(x,z) - \log q(z;\phi)] \tag{3.2}$$

This is performed by relying on the "log-derivative trick" [125, 126] to obtain MC estimates of the gradient. Specifically, by application of the identities

$$\nabla_\phi q(z;\phi) = q(z;\phi)\nabla_\phi \log q(z;\phi) \tag{3.3}$$

$$\mathbb{E}_{q(z;\phi)}[\nabla_\phi \log q(z;\phi)] = 0 \tag{3.4}$$

the gradient of the ELBO (3.2) reads

$$\nabla_\phi \mathscr{L}(\phi) = \mathbb{E}_{q(z;\phi)}[f(z)] \tag{3.5}$$

where

$$f(z) = \nabla_\phi \log q(z;\phi) \left[\log p(x,z) - \log q(z;\phi)\right] \tag{3.6}$$

The so-obtained MC estimator, based on computing the posterior expectations $\mathbb{E}_{q(z;\phi)}[\cdot]$ via sampling from $q(z;\phi)$, only requires evaluating the log-joint distribution $\log p(x,z)$, the log-variational distribution $\log q(z;\phi)$, and the score function $\nabla_\phi \log q(z;\phi)$, which is easy for a large class of models. However, the resulting estimator may have high variance, especially if the variational approximation $q(z;\phi)$ is a poor fit to the actual posterior. In order to reduce the variance of the estimator, one common strategy in BBVI consists in the use of control variates.

A control variate is a random variable that is included in the estimator, preserving its expectation but reducing its variance. The most usual choice for control variates, which we adopt in this work, is the so-called weighted score function: Under this selection, the ELBO gradient becomes

$$\nabla_\phi \mathscr{L}(\phi) = \mathbb{E}_{q(z;\phi)}[f(z) - \varpi h(z)] \tag{3.7}$$

where the score function reads

$$h(z) = \nabla_\phi \log q(z;\phi) \tag{3.8}$$

while the weights ϖ yield the (optimized) expression

$$\varpi = \frac{\mathrm{Cov}\left(f(z), h(z)\right)}{\mathrm{Var}\left(h(z)\right)} \tag{3.9}$$

On this basis, derivation of the sought variational posteriors is performed by utilizing the gradient expression (3.7) in the context of popular, off-the-shelf optimization algorithms, e.g. AdaM [84] and Adagrad [81].

3.3 Proposed Approach

The output expression of a DropConnect++ layer is fundamentally similar to conventional DropConnect, and is given by (3.1). However, DropConnect++ introduces an additional

hierarchical set of assumptions regarding the matrix of binary (mask) variables $\mathbf{Z} = [z_{ij}]_{i,j}$, which indicate whether a synaptic connection is inferred to be on or off.

Specifically, as usual in hierarchical graphical models, we assume that the random matrix $\mathbf{Z}$ is drawn from an appropriate prior; we postulate

$$p(\mathbf{Z}|\Pi) = \prod_{i,j} p(z_{ij}|\pi_{ij}) = \prod_{i,j} \text{Bernoulli}(z_{ij}|\pi_{ij}) \tag{3.10}$$

Subsequently, to facilitate further regularization for the layers under a Bayesian inferential perspective, the prior parameters $\pi_{ij} \triangleq p(z_{ij} = 1)$ are imposed their own (hyper-)prior. Specifically, we elect to impose a Beta hyper-prior, yielding

$$p(\pi_{ij}|\alpha,\beta) = \text{Beta}(\pi_{ij}|\alpha,\beta),\ \forall i,j \tag{3.11}$$

Under this definition, to train a postulated DNN incorporating DropConnect++ layers, we need to resort to some sort of Bayesian inference technique; for this model we will resort to BBVI which was explained in the previous subsection.

3.3.1 Training DNNs with DropConnect++ layers

Let us consider a DNN where the observed training data of which constitute the set $\mathscr{D} = \{\mathbf{d}_n\}_{n=1}^{N}$. In case of a generative modeling scheme, each example $\mathbf{d}_n$ is a single observation, say $\mathbf{x}_n$, from the distribution we wish to model. On the other hand, in case of a discriminative modeling task, each example $\mathbf{d}_n$ is an input/output pair, for instance $\mathbf{d}_n = (\mathbf{x}_n, \mathbf{y}_n)$. In both cases, conventional DNN training consists in optimizing a negative loss function, measuring the fit of the model to the training dataset $\mathscr{D}$. Such measures can be equivalently expressed in terms of a log-likelihood function $\log p(\mathscr{D})$; under this regard, DNN training effectively boils down to maximum-likelihood estimation [127, 77].

The deviation of a DNN comprising DropConnect++ layers from this simple training scheme stems from obtaining appropriate posterior distributions over the latent variables of DropConnect++, namely the binary indicator matrices of synaptic connectivity, $\mathbf{Z}$, as well as the associated parameters with hyper-priors imposed over them, namely the matrices of (prior) parameters Π. To this end, DropConnect++ postulates separate posteriors over each entry of the random matrices $\mathbf{Z}$, that correspond to each individual synapse, (i, j):

$$q(\mathbf{Z}) = \prod_{i,j} q(z_{ij}|\tilde{\pi}_{ij}), \text{ with :} \qquad q(z_{ij}|\tilde{\pi}_{ij}) = \text{Bernoulli}(z_{ij}|\tilde{\pi}_{ij}) \tag{3.12}$$

Further, we consider that the matrices of prior parameters, Π, yield a factorized (hyper-) posterior with Beta-distributed factors of the form

$$q(\pi_{ij}) = \text{Beta}(\pi_{ij}|\tilde{\alpha}_{ij}, \tilde{\beta}_{ij}) \tag{3.13}$$

Our construction entails a conditional log-likelihood term, $\log p(\mathscr{D}|\mathbf{Z})$. This is similar to a conventional DNN, with the weight matrices $\mathbf{W}$ at each layer multiplied with the corresponding latent indicator (mask) matrices, $\mathbf{Z}$ (in analogy to DropConnect). The corresponding posterior expectation term, $\mathbb{E}_{q(\mathbf{Z})}[\log p(\mathscr{D}|\mathbf{Z})]$, constitutes part of the ELBO expression of our model. Unfortunately, this term is analytically intractable due to the entailed nonlinear dependencies on the indicator matrix $\mathbf{Z}$, which stem from the nonlinear activation function $a(\cdot)$. Following the previous discussion, we ameliorate this issue by resorting to an efficient approximation obtained by drawing MC samples. The so-obtained ELBO functional expression eventually becomes:

$$\begin{aligned} \mathscr{L}(\mathscr{D}) \approx & -\sum_{i,j} \text{KL}\big[q(z_{ij}|\tilde{\pi}_{ij})||p(z_{ij}|\pi_{ij})\big] - \sum_{i,j} \text{KL}\big[q(\pi_{ij}|\tilde{\alpha}_{ij}, \tilde{\beta}_{ij})||p(\pi_{ij}|\alpha, \beta)\big] \\ & + \frac{1}{L} \sum_{l,n=1}^{L,N} \log p(d_n|Z^{(l)}) \end{aligned} \tag{3.14}$$

where L is the number of samples, $Z^{(l)} = [z_{ij}^{(l)}]_{i,j}$ and $z_{ij}^{(l)} \sim \text{Bernoulli}(z_{ij}|\tilde{\pi}_{ij})$.

This concludes the formulation of the proposed inferential setup for a DNN that contains DropConnect++ layers. On this basis, inference is performed by resorting to BBVI, which proceeds as described previously. Denoting $\tilde{\pi} = (\tilde{\pi}_{ij})_{i,j}, \tilde{\alpha} = (\tilde{\alpha}_{ij})_{i,j}, \tilde{\beta} = (\tilde{\beta}_{ij})_{i,j}$, the used ELBO gradient reads

$$\begin{aligned} \nabla_{\tilde{\pi},\tilde{\alpha},\tilde{\beta},W} \mathscr{L}(\mathscr{D}) \approx & \frac{1}{L} \sum_{l,n=1}^{L,N} \nabla_W \log p(d_n|Z^{(l)}) - \sum_{i,j} \nabla_{\tilde{\pi},\tilde{\alpha},\tilde{\beta}} \text{KL}\big[q(z_{ij}|\tilde{\pi}_{ij})||p(z_{ij}|\pi_{ij})\big] \\ & - \sum_{i,j} \nabla_{\tilde{\alpha},\tilde{\beta}} \text{KL}\big[q(\pi_{ij}|\tilde{\alpha}_{ij}, \tilde{\beta}_{ij})||p(\pi_{ij}|\alpha, \beta)\big] \\ & - \varpi \sum_{i,j} \nabla_{\tilde{\pi},\tilde{\alpha},\tilde{\beta}} [\log q(z_{ij}|\tilde{\pi}_{ij}) + q(\pi_{ij}|\tilde{\alpha}_{ij}, \tilde{\beta}_{ij})] \end{aligned} \tag{3.15}$$

where ϖ is defined in (3.9). As one can note, we do not perform Bayesian inference for the synaptic weight parameters W. Instead, we obtain point-estimates, similar to conventional DropConnect.

The KL divergence analysis that is used for the ELBO of Eq.(3.15) is:

$$\begin{aligned} \mathrm{KL}\big[q(z_{ij}|\tilde{\pi}_{ij})||p(z_{ij}|\pi_{ij})\big] &= \tilde{\pi}_{ij}\log\tilde{\pi}_{ij} + (1-\tilde{\pi}_{ij})\log(1-\tilde{\pi}_{ij}) \\ &\quad - \tilde{\pi}_{ij}\mathbb{E}_{q(\pi_{ij})}[\log\pi_{ij}] - (1-\tilde{\pi}_{ij})\mathbb{E}_{q(\pi_{ij})}[\log(1-\pi_{ij})] \end{aligned} \tag{3.16}$$

$$\begin{aligned} \mathrm{KL}\big[q(\pi_{ij}|\tilde{\alpha}_{ij},\tilde{\beta}_{ij})||p(\pi_{ij}|\alpha,\beta)\big] &= \log\Gamma(\tilde{\alpha}_{ij}+\tilde{\beta}_{ij}) - \log\Gamma(\tilde{\alpha}_{ij}) - \log\Gamma(\tilde{\beta}_{ij}) \\ &\quad + (\tilde{\alpha}_{ij}-\alpha)\mathbb{E}_{q(\pi_{ij})}[\log\pi_{ij}] + (\tilde{\beta}_{ij}-\beta)\mathbb{E}_{q(\pi_{ij})}[\log(1-\pi_{ij})] \end{aligned} \tag{3.17}$$

where:

$$\mathbb{E}_{q(\pi_{ij})}[\log\pi_{ij}] = \psi(\tilde{\alpha}_{ij}) - \psi(\tilde{\alpha}_{ij}+\tilde{\beta}_{ij}) \tag{3.18}$$

$$\mathbb{E}_{q(\pi_{ij})}[\log(1-\pi_{ij})] = \psi(\tilde{\beta}_{ij}) - \psi(\tilde{\alpha}_{ij}+\tilde{\beta}_{ij}) \tag{3.19}$$

$\Gamma(\cdot)$ is the Gamma function, and $\psi(\cdot)$ is the Digamma function.

3.3.2 Feedforward computation in DNNs with DropConnect++ layers

Computation of the output of a trained DNN with DropConnect++ layers, given some network input x_*, requires that we come up with an appropriate solution to the problem of computing the posterior expectation of the DropConnect++ layers output, say $\mathbf{r}_*$.

Let us consider a DropConnect++ layer with input u_* (corresponding to a DNN input observation x_*); we have

$$r_* = \mathbb{E}_{q(Z)}[a((Z \circ W)u_*)] \tag{3.20}$$

This computation essentially consists in marginalizing out the layer synaptic connectivity structure, by appropriately utilizing the variational posterior distribution $q(Z)$, learned by means of BBVI, as discussed in the previous subsection. Unfortunately, this posterior expectation cannot be computed analytically, due to the nonlinear activation function $a(\cdot)$.

This problem can be solved by approximating (17) via simple MC sampling:

$$r_* \approx \frac{1}{L} \sum_{l=1}^{L} a((Z^{(l)} \circ W)u_*) \tag{3.21}$$

where the $Z^{(l)}$ are drawn from $q(Z)$. However, an issue such an approach suffers from is the need to retain in memory large sample matrices $\{Z^{(l)}\}_{l=1}^{L}$, that may comprise millions of entries, in cases of large-scale DNNs. To completely alleviate such computational efficiency issues, in this work we opt for an alternative approximation that reads

$$r_* \approx a((\tilde{\Pi} \circ W)u_*) \tag{3.22}$$

where the matrix $\tilde{\Pi} = [\tilde{\pi}_{ij}]_{i,j}$ is obtained from the model training algorithm, described previously. Note that such an approximation is similar to the solution adopted by Dropout, which undoubtedly constitutes the most popular DNN regularization technique to date. We shall examine how this solution compares to MC sampling in the experimental subsection.

3.4 Experimental Evaluation

To empirically evaluate the performance of our approach, we consider a number of supervised learning experiments, using the CIFAR-10, CIFAR-100, SVHN, and NORB benchmarks. In all our experiments, the used datasets are normalized with local zero mean and unit variance; no other pre-processing is implemented in this work[1]. To obtain some comparative results, apart from our method we also evaluate in our experiments DNNs with similar architecture but: (i) application of no regularization technique; (ii) regularized via Dropout; and (iii) regularized via DropConnect.

In all cases, we use Adagrad with minibatch size equal to 128. Adagrad's global stepsize is chosen from the set $\{0.005, 0.01, 0.05\}$, based on the network performance on the training set in the first few iterations[2]. The units of all the postulated DNNs comprise ReLU nonlinearities

[1]Hence, our experimental setup is not completely identical to that of related works, e.g. dropconnect; these employ more complex pre-processing for some datasets.

[2]We have found that Adagrad allows for the best possible network regularization by drawing just one sample per minibatch; that is, we use $L = 1$ at training time. This alleviates the training costs of both DropConnect and DropConnect++. We train all networks for 100 epochs; we do not apply L2 weight decay.

Table 3.1: Predictive accuracy (%) of the evaluated methods.

Method	CIFAR-10	CIFAR-100	SVHN	NORB
No regularization	74.47	41.96	90.53	90.55
Dropout	75.70	46.65	92.14	92.07
DropConnect	76.06	46.12	91.41	91.88
DropConnect++	76.54	47.01	91.99	93.75

Table 3.2: Computational complexity (sec) per iteration at training time ($L = 1$).

#Method	CIFAR-10	CIFAR-100	SVHN	NORB
No regularization	9	10	15	5
Dropout	9	10	15	5
DropConnect	9	10	15	5
DropConnect++	10	13	19	6

[60]. Initialization of the network parameters is performed via Glorot-style uniform initialization [77]. To account for the effects of random initialization on the observed performances, we repeat our experiments 50 times; we report the resulting mean accuracies, and run the Student's-t statistical significance test to examine the statistical significance of the reported performance differences.

Prediction generation using our method is performed by employing the efficient approximation (3.22). The alternative approach of relying on MC sampling to perform feedforward computation (3.21) is evaluated in subsection 3.4.7. In all cases, we set the prior hyperparameters of DropConnect++ to $\alpha = \beta = 1$; this is a convenient selection which reflects that we have no preferred values for the priors π_{ij}. The Dropout and DropConnect rates are selected on the grounds of performance maximization, following the selection procedures reported in the related literature. Our source codes have been developed in Python, using the Theano[3] [128] and Lasagne[4] libraries. We run our experiments on an Intel Xeon 2.5GHz Quad-Core server with 64GB RAM and an NVIDIA Tesla K40 GPU.

[3] `http://deeplearning.net/software/theano/`

[4] `https://github.com/Lasagne/Lasagne`.

CIFAR-10

The CIFAR-10 dataset consists of color images of size 32×32, that belong to 10 categories (airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships, trucks). We perform our experiments using the available 50,000 training samples and 10,000 test samples. All the evaluated methods comprise a convolutional architecture with three layers, 32 feature maps in the first layer, 32 feature maps in the second layer, 64 feature maps in the third layer, a 5×5 filter size, and a max-pooling sublayer with a pool size of 3×3. These three layers are followed by a dense layer with 64 hidden units, regularized via Dropout, DropConnect, or DropConnect++. The resulting performance statistics (predictive accuracy) of the evaluated methods are depicted in the first column of Table 3.1. As we observe, our approach outperforms all the considered competitors.

CIFAR-100

The CIFAR-100 dataset consists of 50,000 training and 10,000 testing color images of size 32×32, that belong to 100 categories. We retain this split of the data into a training set and a test set in the context of our experiments. The trained DNN comprises three convolutional layers of same architecture as the ones adopted in the CIFAR-10 experiment, that are followed by a dense layer comprising 512 hidden units. As we show in Table 3.1, our approach outperforms all its competitors, yielding the best predictive performance. Note also that the DropConnect method, which is closely related to our approach, yields in this experiment worse results than Dropout.

SVHN

The Street View House Numbers (SVHN) dataset consists of 73,257 training and 26,032 test color images of size 32x32; these depict house numbers collected by Google Street View. We retain this split of the data into a training set and a test set in the context of our experiments, and adopt exactly the same DNN architecture as in the CIFAR-100 experiment. As we show in Table 3.1, our method improves over the related DropConnect method.

Table 3.3: Variation of the predictive accuracy (%) of the MC-driven approach (Eq:3.21) with the number of MC samples.

#Samples, L	CIFAR-10	CIFAR-100	SVHN	NORB
1	74.57	43.28	91.32	90.04
30	75.95	46.33	91.70	90.78
50	76.01	46.33	91.72	91.04
100	76.01	46.54	91.78	91.41
500	76.36	46.94	91.78	91.58

NORB

The NORB (small) dataset comprises a collection of stereo images of 3D models that belong to 6 classes (animal, human, plane, truck, car, blank). We downsample the images from 96×96 to 32×32, and perform training and testing using the provided dataset split. We train DNNs with architecture similar to the one adopted in the context of the SVHN and CIFAR-100 datasets. As we show in Table 3.1, our method outperforms all the considered competitors.

3.4.1 Computational complexity

Another significant aspect that affects the efficacy of a regularization technique is its final computational costs, and how they compare to the competition. To allow for investigating this aspect, in Table 3.2 we illustrate the time needed to complete one iteration of the training algorithms of the evaluated networks in our implementation. As we observe, the training algorithm of our approach imposes an 11%-30% increase in the computational time per iteration, depending on the sizes of the network and the dataset. Note though that DNN training is an offline procedure; hence, a relatively small increase in the required training time is reasonable, given the observed predictive performance gains.

On the other hand, when it comes to using a trained DNN for prediction generation (test time), we emphasize that the computational costs of our approach are exactly the same as in the case of Dropout. This is, indeed, the case due to our utilization of the approximation (3.22), which results in similar feedforward computations for DropConnect++ as in the case of Dropout.

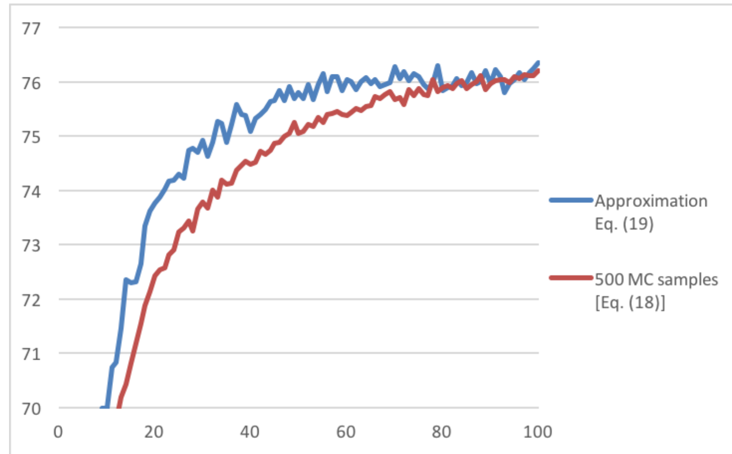


Figure 3.1: Accuracy convergence

3.4.2 Further investigation

A first issue that requires deeper investigation concerns the statistical significance of the observed performance differences. Application of the Student's-t test on the obtained sets of performances of each method (after 50 experiment repetitions from different random starts) has shown that these differences are statistically significant among all relevant pairs of methods (i.e. DropConnect++ vs. DropConnect, DropConnect++ vs. DropOut, and DropConnect++ vs. no regularization); only exception is the SVHN dataset, where DropConnect++ and DropOut are shown to be of statistically comparable performance.

Further, in Table 3.3 we show how the predictive performance of DropConnect++ changes if we perform feedforward computation via MC sampling, as described in (3.21). As we observe, using only one MC sample results in rather poor performance; this changes fast as we increase the number of samples. However, it appears that even with a high number of drawn samples, the MC-driven approach (3.21) does not yield any performance improvement over the approximation (3.22), despite imposing considerable computational overheads.

Further, in figure 3.1 we illustrate predictive accuracy convergence; for demonstration purposes, we consider the experimental case of the CIFAR-10 benchmark. Our exhibition concerns both application of the approximate feedforward computation rule (3.22), as well as resorting to MC sampling. We observe a clear and consistent convergence pattern in both cases.

Finally, it is interesting to get a feeling of the values that take the inferred posterior probabilities, $\tilde{\pi}$, of synaptic connectivity. In figure 3.2, we illustrate the inferred values of $\tilde{\pi}$ for all the network synapses, in the case of the CIFAR-10 experiment. As we observe, out of the almost

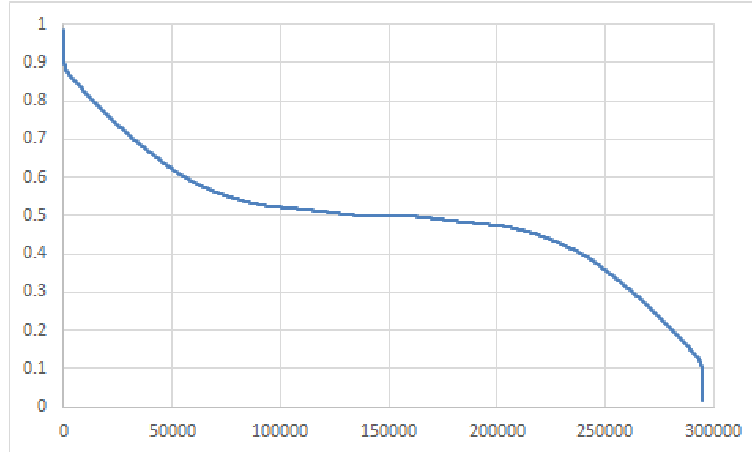


Figure 3.2: Inferred posterior probabilities, $\tilde{\pi}$.

300K synapses, around 50K take values less than 0.35, another 50K take values greater than 0.6, while the rest 200K take values approximately in the interval $[0.4, 0.6]$. This implies that, out of the total 300K postulated synapses, almost half of them are most likely to be omitted during inference. Most significantly, this figure depicts that our approach infers (in a data-driven fashion) which specific synapses are most useful to the network (thus yielding relatively high values of $\tilde{\pi}_{ij}$), and which should rather be omitted. This is in contrast to existing approaches, which merely apply a homogeneous, random omission/retention rate on each layer.

3.5 Conclusions

In this last section, we examined whether there is a feasible way of performing DNN regularization by marginalizing over network synaptic connectivity in a Bayesian manner. Specifically, we sought to derive an appropriate posterior distribution over the network synaptic connectivity, inferred from the data. To this end, we imposed a sparsity-inducing prior over the network synaptic weights, where the sparsity is induced by a set of Bernoulli-distributed binary variables. Further, we imposed appropriate Beta (hyper-)priors over the parameters of the postulated Bernoulli-distributed binary variables. Under this hierarchical Bayesian construction, we obtained appropriate posteriors over the postulated binary variables, which indicate which synaptic connections are retained and which are dropped during inference. This was effected in an efficient and elegant fashion, by resorting to BBVI. We performed an extensive experimental evaluation of our approach using several benchmark datasets. As we showed, in

most cases our approach yields a statistically significant performance improvement, without compromises in computational efficiency, especially at test time (prediction generation).

Chapter 4

Asymmetric Deep Generative Models

Amortized variational inference, whereby the inferred latent variable posterior distributions are parameterized by means of neural network functions, has invigorated a new wave of innovation in the field of generative latent variable modeling, giving rise to the family of deep generative models (DGMs). Existing DGM formulations are based on the assumption of a symmetric Gaussian posterior over the model latent variables. This assumption, although mathematically convenient, can be well-expected to undermine the eventually obtained representation power, as it imposes apparent expressiveness limitations. Indeed, it has been recently shown that even some moderate increase in the latent variable posterior expressiveness, obtained by introducing an additional level of dependencies upon auxiliary (Gaussian) latent variables, can result in significant performance improvements in the context of semi-supervised learning tasks. Inspired from these advances, in this Chapter we examine whether a more potent increase in the expressiveness and representation power of modern DGMs can be achieved by completely relaxing their typical symmetric (Gaussian) latent variable posterior assumptions: Specifically, we consider DGMs with asymmetric posteriors, formulated as restricted multivariate skew-Normal (rMSN) distributions. We derive an efficient amortized variational inference algorithm for the proposed model, and exhibit its superiority over the current state-of-the-art in several semi-supervised learning benchmarks.

The aforementioned innovative approach dubbed as AsyDGM has been published in the journal ”Neurocomputing” in 2017 under the title ”Asymmetric deep generative models”.

4.1 Introduction

Amortized variational inference [55, 129, 98, 130], whereby the inferred latent variable posteriors are parameterized via deep neural networks, is currently at the epicenter of the research on generative latent variable modeling. The class of DGMs has arisen as the outcome of this research line. Existing DGM formulations postulate symmetric (Gaussian) posteriors over the model latent variables. This assumption, although computationally efficient, may undermine the representation power of DGMs, as it imposes apparent expressiveness limitations [131]. To address these issues, in one of the most recent developments in the field, [132] proposed the skip DGM (SDGM); this model introduces an extra layer of auxiliary latent variables, also imposed symmetric Gaussian posteriors, with the original latent variable posteriors assumed to be conditioned upon the auxiliary latent variables. Apparently, such a hierarchical latent variable construction gives rise to obtained variational posteriors with more expressiveness and representation power. Indeed, [132] have provided broad empirical evidence corroborating these claims, by showing that SDGM yields the state-of-the-art performance in several semi-supervised learning benchmarks.

Inspired from these advances, in this Chapter we examine whether we can achieve a higher level of expressiveness and representation power for the latent variable posteriors of modern DGMs by completely relaxing their typical symmetric (Gaussian) latent variable posterior assumptions. Indeed, in many applied problems, the data to be analyzed may contain a group or groups of observations whose distributions are moderately or severely skewed. Unfortunately, typical DGM formulations based on Gaussian posterior assumptions cannot effectively model data of such nature: A slight deviation from normality may seriously affect the obtained estimates, subsequently misleading inference from the data. Therefore, accounting for asymmetric effects and skewness in the modeled data may allow for significant improvements in the potency of DGM models. On this basis, in this work we introduce the novel class of asymmetric DGMs (AsyDGMs), characterized by asymmetric latent variable posteriors, that are formulated as restricted multivariate skew-Normal (rMSN) distributions [133, 134].

In recent years, there has been growing interest in studying generative models based on latent variables with skew-elliptical distributions [135, 136], both in the univariate and multivariate cases. Their popularity with the statistics community mainly stems from them being regarded as a more general tool for handling heterogeneous data that involve asymmetric behavior across sub-populations. For instance, [137] and [138] proposed mixtures of multivariate skew-normal

and t-distributions based on a restricted variant of the skew-elliptical family of distributions of [134]; [139] gave a systematic overview of various existing multivariate skew distributions and clarified their conditioning-type and convolution-type representations. There also is a small corpus of works proposing generative models the latent variables of which are imposed skewed priors. These are shallow, factor analysis-type models, which provide strong motivation for the work presented in this Chapter. For instance, [140] proposed mixtures of shifted asymmetric Laplace factor analyzers; [141] proposed mixtures of generalized hyperbolic factor analyzers; [142] proposed mixtures of skew-t factor analyzers. Finally, very recently, a finite mixture model of rMSN-distributed factor analyzers was proposed in [143], and an efficient EM algorithm comprising closed-form updates was derived for model training.

We derive an efficient inference algorithm for the proposed AsyDGM approach by resorting to an elegant amortized variational inference algorithm, similar to existing DGMs. To exhibit the efficacy of our approach, and its superiority over existing symmetrically-distributed DGMs, we perform a series of experimental evaluations. Specifically, we focus on challenging semi-supervised learning tasks, where DGM-type classifier training is performed with a very limited number of labeled examples. We show that our approach yields the state-of-the-art performance in these benchmarks, with a significant improvement over the second-best method.

The remainder of this chapter is organized as follows: In the following Section, we provide a brief overview of the theoretical foundation of our work: We first present the Variational Auto Encoder (VAE), further we introduce the rMSN distribution; then, we briefly present the SDGM model, which constitutes the latest development in the field of DGMs, when it comes to addressing challenging semi-supervised learning tasks. Next, we introduce our approach, and derive its inference and prediction generation algorithms. In the subsequent experimental Section, we perform an exhaustive empirical evaluation of our approach, using well-known semi-supervised learning benchmarks. Finally, in the concluding Section of this chapter, we summarize our contribution and discuss our results.

4.2 Theoretical Foundation

4.2.1 Variational Auto-Encoder

The VAE is a generative model that attempts to learn the underlying distribution of high dimensional data with complex dynamics. To this end, VAE uses amortized variational inference (AVI). AVI represents the sought (approximate) variational posterior distribution over the model latent variables via an inference network. The imposed inference network learns an inverse map from observations to latent variables. This way it alleviates the need to compute per data point variational parameters by computing a set of global variational parameters, valid for inference at both training and test time. Thus, the cost of inference is amortized by generalizing between the posterior estimates for all latent variables through the parameters of the inference network, under a simple feedforward computation scheme with complexity O(N) where N the number of datapoints.

Let us consider a dataset $X = \{x_n\}_{n=1}^{N}$ consisting of N samples of some observed random variable x. We assume that the observed random variable is generated by some random process, involving an unobserved continuous random variable z. In this context, we introduce a conditional independence assumption for the observed variables x given the corresponding latent variables z; we adopt the conditional likelihood function $p(x|z;\theta)$.

To perform Bayesian inference for the latent variables, we impose some prior distribution $p(z;\varphi)$. Under this formulation, the log-marginal likelihood of the model with respect to the dataset X yields the following lower bound expression

$$\log p(X) \geq \mathscr{L}(\theta,\varphi,\phi|X) = \sum_{i=1}^{N}\Big\{ -\mathrm{KL}\big[q(z_i;\phi)||p(z_i;\varphi)\big] + \mathbb{E}_{q(z_i;\phi)}[\log p(x_i|z_i;\theta)]\Big\} \quad (4.1)$$

where $\mathrm{KL}\big[q||p\big]$ is the KL divergence between the distribution $q(\cdot)$ and the distribution $p(\cdot)$, $q(z;\phi)$ is the sought approximate (variational) posterior over the latent variable z, while $\mathbb{E}_{q(z;\phi)}[\cdot]$ is the (posterior) expectation of a function with respect to the random variable z, the distribution of which is taken to be the posterior $q(z;\phi)$.

AVI assumes that the adopted likelihood and prior distributions come from a parametric family, and that their probability density functions are differentiable almost everywhere with respect to the parameters θ and φ, and the (latent) variables z. Specifically, AVI assumes that the likelihood function of the model, as well as the resulting latent variable posterior, $q(z;\phi)$, are

parameterized via deep neural networks. This yields a non-conjugate model construction, which does not allow to analytically derive the expression of $\mathbb{E}_{q(z_i;\phi)}[\log p(x_i|z_i;\theta)]$, and, hence, of the derivative of $\mathscr{L}(\theta,\varphi,\phi|X)$. Besides, attempting to resolve this issue by means of a naive Monte Carlo gradient estimator is not an option in our context, due to its entailed prohibitively high variance that renders it completely impractical [144].

AVI resolves these issues by reparameterizing the random samples of $z \sim q(z;\phi)$ using an appropriate differentiable transformation of an (auxiliary) random noise variable ε. Specifically, by drawing L samples, the ELBO expression becomes

$$\mathscr{L}(\theta,\varphi,\phi|X) = \sum_{i=1}^{N} \left\{ -\mathrm{KL}\big[q(z_i;\phi)||p(z_i;\varphi)\big] + \frac{1}{L}\sum_{l=1}^{L} \log p(x_i|z_i^{(l)};\theta) \right\} \tag{4.2}$$

where, considering a Gaussian posterior of the form

$$q(z_i;\phi) = \mathscr{N}(z_i|\mu_\phi(x_i), \mathrm{diag}\, \sigma^2_\phi(x_i)) \tag{4.3}$$

we have:

$$z_i^{(l)} = \mu_\phi(x_i) + \sigma_\phi(x_i) \cdot \varepsilon_i^{(l)} \tag{4.4}$$

In Eq.(4.4), $\varepsilon_i^{(l)}$ is white random noise with unitary variance, i.e. $\varepsilon_i^{(l)} \sim \mathscr{N}(0,I)$, the $\mu_\phi(x_i)$ and $\sigma^2_\phi(x_i)$ are parameterized via deep neural networks, and $\mathrm{diag}\,\chi$ is a diagonal matrix with χ on its main diagonal.

As we observe, the key difference between AVI and, say, a naive Monte Carlo estimator, is that the drawn samples of z, used to approximate the intractable posterior expectation $\mathbb{E}_{q(z_i;\phi)}[\log p(x_i|z_i;\theta)]$, are now taken as functions of the parameters ϕ of the posterior $q(z_i;\phi)$ that we seek to optimize. As proven in [129], this formulation of the inference algorithm allows for yielding low variance estimators, under some mild conditions.

4.2.2 The rMSN distribution

We continue with a brief review of the rMSN distribution. To establish notation, let $\mathscr{N}(\cdot|\mu,\Sigma)$ be the probability density function (pdf) of multivariate Gaussian with mean vector μ and variance–covariance matrix Σ, and $\Phi(\cdot)$ be the cumulative distribution function (cdf) of the

standard normal distribution. Further, let $TN(\cdot|\mu, \sigma^2; (a,b))$ denote the truncated normal distribution for $\mathcal{N}(\cdot|\mu, \sigma^2)$ lying within a truncated interval (a,b).

Following [138], a random vector $x \in \mathbb{R}^d$ is said to follow an rMSN distribution with location vector μ, dispersion matrix Σ, and skewness vector λ, denoted by $x \sim \text{rSN}(\mu, \Sigma, \lambda)$, if it can be represented as

$$\begin{aligned} x|u &\sim \mathcal{N}(\mu + \lambda u, \Sigma) \\ u &\sim TN(0,1;(0,\infty)) \end{aligned} \tag{4.5}$$

Here, the truncated Normal distribution $TN(u|\mu_u, \sigma_u^2; (0,\infty))$ with mean μ_u, variance σ_u^2, and bounds in $(0,\infty)$, is defined as

$$TN(u|\mu_u, \sigma_u^2; (0,\infty)) = \frac{\mathcal{N}(u|\mu_u, \sigma_u^2)}{\Phi(\mu_u/\sigma_u)} I(u > 0) \tag{4.6}$$

where $I(\cdot)$ is an indicator function. Hence, we observe that an rMSN-distributed variable can be equivalently expressed under a Gaussian conditional distribution, where the introduced conditioning latent variable follows a standard truncated normal density.

On this basis, [143] have recently proposed a generalization of the traditional factor analysis (FA) model, namely the SNFA model, where the latent variables (factors) are assumed to follow an rMSN distribution within the family defined by (4.5). Let us denote as $x \in \mathbb{R}^p$ the p-dimensional observations we wish to model via an SNFA model. Denoting as $z \in \mathbb{R}^q$ the inferred latent factors vectors ($q < p$), we have [143]

$$p(x|z) = \mathcal{N}(\mu + Bz, D) \tag{4.7}$$

and

$$p(z) = \text{rSN}(z| - c\Delta^{-1/2}\lambda, \Delta^{-1}, \Delta^{-1/2}\lambda) \tag{4.8}$$

where μ is a p-dimensional location vector, B is a parameter matrix of the SNFA model (factor loadings), D is a *diagonal* covariance matrix, $c \triangleq \sqrt{2/\pi}$, λ is the skewness vector of the model, and

$$\Delta \triangleq I + (1 - c^2)\lambda\lambda^T \tag{4.9}$$

As shown in [143], by using the definition (4.5) of the rMSN distribution, this asymmetric factor analysis model can be equivalently expressed under the following three-level hierarchical representation:

$$p(x|\tilde{z}) = \mathcal{N}(\mu + Bg(\tilde{z}), D) \tag{4.10}$$

$$p(\tilde{z}|u) = \mathcal{N}(\tilde{z}|(u-c)\lambda, I) \tag{4.11}$$

and

$$u \sim TN(0,1;(0,\infty)) \tag{4.12}$$

where

$$g(\tilde{z}) = \Delta^{-1/2}\tilde{z} \tag{4.13}$$

Under this equivalent representation, the SNFA model yields a simple prior formulation that is amenable to a computationally efficient EM training algorithm with closed-form expressions [143].

4.2.3 Skip Deep Generative Models

As discussed in the Introduction, modern developments in the field of variational inference focus on using deep learning techniques to parameterize the variational posteriors of latent variable models. This gives rise to powerful probabilistic models, usually referred to as DGMs, constructed by an inference neural network that parameterizes the posterior $q(z|x)$, and a generative neural network that parameterizes the conditional likelihood $p(x|z)$.

To allow for keeping the computational requirements low, the variational distribution $q(z|x)$ is usually chosen to be a diagonal Gaussian. Despite the computational attractiveness of this approximation, it is quite apparent though that such an assumption may not allow for capturing intricate latent dynamics in the modeled data, as well as modeling data of asymmetric nature. Hence, one could expect that by relaxing these diagonal Gaussian posterior assumptions, one may yield DGMs with increased expressive power.

Recently, [132] proposed an way of ameliorating these issues of DGMs by drawing inspiration from the variational auxiliary variable approach of [145]. The so-obtained *Skip-DGM*(SDGM) extends the variational distribution with some auxiliary variables a, such that

$$q(a,z|x) = q(z|a;x)q(a|x) \tag{4.14}$$

and

$$q(z|x) = \int q(a,z|x)\mathrm{d}a \tag{4.15}$$

where both the postulated variational posteriors $q(z|a;x)$ and $q(a|x)$ are typical inference networks with diagonal Gaussian form. Under such a two-level hierarchical formulation, it becomes well-expected that the marginal distribution $q(z|x)$ will be able to fit more complicated posteriors compared to conventional (one-level) diagonal Gaussian distribution-based DGM formulations. Indeed, [132] have shown that the SDGM approach can be utilized in the context of semi-supervised learning tasks, with the goal of building a potent classifier, capable of obtaining state-of-the-art performance in challenging datasets by being trained with limited labeled examples combined with large unlabeled datasets.

More specifically, denoting as x the observed vectors presented as input to the postulated classifier, and as y the corresponding label variables, SDGM consecutively postulates the following generative (i.e., conditional likelihood and prior) assumptions [132]:

$$p_{\theta}(x|a,z,y) = f(x,a,z,y;\theta) \tag{4.16}$$

$$p_{\theta}(a|z,y) = f(a,z,y;\theta) \tag{4.17}$$

$$p(y) = \mathrm{Cat}(y|\pi) \tag{4.18}$$

$$p(z) = \mathcal{N}(z|0,I) \tag{4.19}$$

where $f(x,a,z,y;\theta)$ is a categorical or diagonal Gaussian for discrete and continuous observations x, respectively, and $f(a,z,y;\theta)$ is a diagonal Gaussian. Both $p_{\theta}(\cdot)$ distributions are

parameterized by deep neural networks with parameters θ. On the other hand, the derived variational posteriors (inference model) are assumed to take on the following form:

$$q_\phi(a|x) = \mathcal{N}(a|\mu(x;\phi), \operatorname{diag}\sigma^2(x;\phi)) \tag{4.20}$$

$$q_\phi(z|a,x,y) = \mathcal{N}(z|\mu(a,x,y;\phi), \operatorname{diag}\sigma^2(a,x,y;\phi)) \tag{4.21}$$

$$q_\phi(y|a,x) = \mathrm{Cat}(y|\pi(a,x;\phi)) \tag{4.22}$$

[the computed (output) probabilities of the network $\pi(a,x;\phi)$ are the ones used to perform the classification task]. Note that, in order to parameterize the diagonal Gaussians $p_\theta(\cdot)$ and $q_\phi(\cdot)$ in Eqs. (4.16)-(4.17) and (4.20)-(4.21), respectively, [132] define two separate outputs from the top deterministic layer in the corresponding deep neural networks, one for the distribution mean and one for the distribution (log-)variance.

An issue variational inference for DGM-type models, including SDGM, is confronted with is the analytical intractability of the expressions of the entailed expectations of the model latent variables with respect to the sought approximate (variational) posteriors. This is due to their nonconjugate formulation, as a consequence of their nonlinear parameterization via deep neural networks. Specifically, considering a training set $\mathscr{D} = \{(x_n, y_n)\}_{n=1}^N$ comprising N samples, the expression of the evidence lower bound (ELBO) of SDGM yields:

$$\begin{aligned}\mathscr{L}(\theta,\phi|\mathscr{D}) = \sum_{n=1}^{N} \Big\{ & -\mathrm{KL}\big[q_\phi(z_n|a_n,x_n,y_n)||p(z_n)\big] \\ & -\mathrm{KL}\big[q_\phi(a_n|x_n)||p_\theta(a_n|z_n,y_n)\big] \\ & -\mathrm{KL}\big[q_\phi(y_n|a_n,x_n)||p(y_n)\big] \\ & +\mathbb{E}_{q_\phi(z,a|x,y)}[\log p_\theta(x_n|a_n,z_n,y_n)]\Big\}\end{aligned} \tag{4.23}$$

where $\mathrm{KL}\big[q||p\big]$ is the KL divergence between the distribution $q(\cdot)$ and the distribution $p(\cdot)$. Under the assumed nonlinear (hence, nonconjugate) model construction, it is easy to observe that neither the ELBO $\mathscr{L}(\theta,\phi|\mathscr{D})$ nor its derivatives with respect to the parameter sets θ and ϕ can be computed analytically. In addition, opting for a naive Monte Carlo gradient estimator is not an option in our context, due to its entailed prohibitively high variance that renders it completely impractical for our purposes [144].

These issues can be addressed by resorting to the popular reparameterization trick [129], commonly employed in the context of amortized variational inference. This consists in approximating the posterior expectations in (4.23) as averages over a set of L samples from the corresponding Gaussian posteriors, $\{a_n^{(l)}, z_n^{(l)}\}_{l=1}^{L}$; the latter samples are expressed as differentiable transformations of the form $\xi_\phi(\varepsilon)$ of the posterior parameters ϕ given some random noise input ε. Specifically, we have [132]:

$$a^{(l)} = \xi_\phi(\varepsilon^{(l)}; x) = \mu(x;\phi) + \sigma(x;\phi) \circ \varepsilon^{(l)} \tag{4.24}$$

and

$$\begin{aligned} z^{(l)} =& \xi_\phi(\varepsilon^{(l)}; a, x, y) \\ =& \mu(a^{(l)}, x, y; \phi) + \sigma(a^{(l)}, x, y; \phi) \circ \varepsilon^{(l)} \end{aligned} \tag{4.25}$$

where $\circ$ is the elementwise product, and the $\varepsilon^{(l)}$ are white random noise samples with unitary variance, i.e. $\varepsilon^{(l)} \sim \mathcal{N}(0, I)$.

4.3 Proposed Approach

In the following, we introduce a DGM where the latent variables of which are assumed to follow rMSN distributions. Since in this work we are interested in semi-supervised learning tasks, our exposition and derivations will be performed in the context of the graphical model of SDGM. However, a similar asymmetric modeling scheme can be employed in the context of any desired graphical formulation for a postulated DGM.

To define our model, we elect to express the rMSN-distributed latent variables under the equivalent three-level hierarchical representation scheme adopted by [143], described by Eqs. (4.10)-(4.13). On this basis, our proposed AsyDGM consecutively postulates the following generative (i.e., conditional likelihood and prior) assumptions:

$$p_\theta(x|a, z, y) = f(x, g(a), g(z), y; \theta) \tag{4.26}$$

$$p_\theta(a|z, u, y) = f(a, g(z), u, y; \theta) \tag{4.27}$$

$$p(y) = \text{Cat}(y|\pi) \tag{4.28}$$

$$p(z|u) = \mathcal{N}(z|(u-c)\lambda, I) \tag{4.29}$$

$$p(u) = TN(u|0,1;(0,\infty)) \tag{4.30}$$

where [denoting $\xi \in \{z,a\}$]:

$$g(\xi) \triangleq \Delta^{-1/2}\xi \tag{4.31}$$

$$\Delta \triangleq I + (1-c^2)\lambda\lambda^T \tag{4.32}$$

$c \triangleq \sqrt{2/\pi}$, λ is the skewness vector of the model, and the pdf's $f(\cdot)$ in (4.26) and (4.27) are defined similar to SDGM.

On this basis, the derived variational posteriors (inference model) of AsyDGM are assumed to take on the following form:

$$q_\phi(y|a,x) = \text{Cat}(y|\pi(a,x;\phi)) \tag{4.33}$$

$$q_\phi(a|x,u) = \mathcal{N}(a|\mu(x,u;\phi), \text{diag}\,\sigma^2(x,u;\phi)) \tag{4.34}$$

$$q_\phi(z|u,a,x,y) = \mathcal{N}(z|\mu(u,a,x,y;\phi), \text{diag}\,\sigma^2(u,a,x,y;\phi)) \tag{4.35}$$

and

$$q_\phi(u|x) = TN(u|m(x;\phi), s^2(x;\phi), (0,\infty)) \tag{4.36}$$

where

$$\mu(u,a,x,y;\phi) = \mu(u,h(a,x,y);\phi) \tag{4.37}$$

and

$$\sigma^2(u,a,x,y;\phi) = \sigma^2(u,h(a,x,y);\phi) \tag{4.38}$$

All the postulated generative and inference networks, which parameterize the generative components $p_\theta(\cdot)$ and the variational posteriors $q_\phi(\cdot)$ of AsyDGM, constitute deep neural networks. In cases of Gaussian or truncated Gaussian densities, these networks define two separate outputs from their top deterministic layer, one for the distribution mean and one for the distribution (log-)variance. Specifically, we have

$$\mu(x,u;\phi) = \text{Linear}(u,h_1(x)) \tag{4.39}$$

$$\sigma^2(x,u;\phi) = \exp(\text{Linear}(u,h_1(x))) \tag{4.40}$$

$$\pi(a,x;\phi) = \text{Softmax}(h_2(a,x)) \tag{4.41}$$

$$m(x;\phi) = \text{Linear}(h_3(x)) \tag{4.42}$$

$$s^2(x;\phi) = \exp(\text{Linear}(h_3(x))) \tag{4.43}$$

$$\mu(u,a,x,y;\phi) = \text{Linear}(u,h(a,x,y)) \tag{4.44}$$

$$\sigma^2(u,a,x,y;\phi) = \exp(\text{Linear}(u,h(a,x,y))) \tag{4.45}$$

where Linear$(\cdot)$ is a linear layer, Softmax$(\cdot)$ is a softmax layer, and the $h_1(\cdot)$, $h_2(\cdot)$, $h_3(\cdot)$, and $h(\cdot)$ are deep neural networks.

Note also our introduced linear dependence assumptions for the mean and variance of $q_\phi(z|u,a,x,y)$ and $q_\phi(a|x,u)$ upon the latent variable u [Eqs. (4.44)-(4.45) and (4.39)-(4.40), respectively]. This selection is motivated by the related derivations that apply to the case of simple factor analysis-type models postulating rMSN-distributed latent factors, e.g. [143]. In

addition, it facilitates the derivation of a computationally efficient inference algorithm for the proposed model. Specifically, the ELBO expression of AsyDGM can be shown to yield

$$
\begin{aligned}
\mathscr{L}(\theta,\phi|\mathscr{D}) = \sum_{n=1}^{N} \Big\{ & -\mathrm{KL}\big[q_\phi(u_n|x_n)||p(u_n)\big] \\
& -\mathrm{KL}\big[q_\phi(z_n|u_n,a_n,x_n,y_n)||p(z_n|u_n)\big] \\
& -\mathrm{KL}\big[q_\phi(a_n|x_n,u_n)||p_\theta(a_n|z_n,u_n,y_n)\big] \\
& -\mathrm{KL}\big[q_\phi(y_n|a_n,x_n)||p(y_n)\big] \\
& +\mathbb{E}_{q_\phi(z),q_\phi(a)}[\log p_\theta(x_n|a_n,z_n,y_n)] \Big\}
\end{aligned} \tag{4.46}
$$

Computation of the term $\mathrm{KL}\big[q_\phi(u_n|x_n)||p(u_n)\big]$ in (4.46) can be tractably performed in an analytical fashion. In addition, due to the aforementioned linear dependence scheme, the same holds for the posterior expectations with respect to $q(u)$ which are entailed in the computation of $\mathrm{KL}\big[q_\phi(z_n|u_n,a_n,x_n,y_n)||p(z_n|u_n)\big]$ and $\mathrm{KL}\big[q_\phi(a_n|x_n,u_n)||p_\theta(a_n|z_n,u_n,y_n)\big]$. This way, the need of applying the reparameterization trick in the context of the inference algorithm of AsyDGM is limited to the latent vectors z and a (similar to SDGM). This clearly facilitates computational efficiency for our method, since application of the reparameterization trick in the case of truncated normal distributions would require computation of Gaussian quantile functions, which is quite complex.

4.4 Expiremental Evaluation

To exhibit the efficacy of our approach, we perform evaluation in a series of challenging semi-supervised learning tasks. We especially focus on tasks that entail high-dimensional observations with several artifacts that render the Gaussian assumption too simplistic; these include both skewness and outliers. In the experimental evaluations of subsections 4.4.1 and 4.4.2, we randomly split the available datasets into a training set and a test set that contain half of the available video frames in each case. In the experimental evaluations of subsections 4.4.1-4.4.3, we retain a randomly selected 10% of the available training data labels, and we discard the rest; the used deep neural networks [denoted as $h_k(\cdot)$ in Eqs. (4.39)-(4.45)] comprise two fully connected hidden layers, with 50 ReLU [146] units each, while the size of the latent vectors z, as well as the auxiliary latent vectors, a, is set to 50. In the experimental evaluations of subsection 4.4.4, we use the available splits of the considered datasets into a

training set and a test set; network configuration is adopted from [132], while the number of retained training data labels is provided in Table 4.3.

In all cases, to alleviate the effect of this random dataset selection, we repeat our experiments 50 times, with different splits of the data each time. To provide some comparative results, apart from our method we also evaluate in the same experiments some alternative DGM-type models, recently proposed for addressing the problem of semi-supervised learning. Specifically, we compare to the closely-related SGDM method [132], the M1+M2 and M1+TSVM approaches proposed in [98] and the VAT approach recently presented in [147].

In all our experiments, the matrix power $\Delta^{-1/2}$ entailed in (4.31) is approximated by means of a first-order Taylor expansion; this facilitates computational efficiency. Specifically, we have

$$\begin{aligned} \Delta^{-1/2} =& (I + (1-c^2)\lambda\lambda^T)^{-1/2} \\ \approx& I - \frac{1}{2}(1-c^2)\lambda\lambda^T \end{aligned} \tag{4.47}$$

To optimize the ELBO $\mathscr{L}(\theta, \phi|\mathscr{D})$ of our model with respect to its trainable parameters, we resort to the Adam optimization algorithm [84]; we use a learning rate of 3×10^{-4}, and an exponential decay rate for the first and second moment at 0.9 and 0.999, respectively. Initialization of the network parameters is performed by adopting a Glorot-style uniform initialization scheme [77]. Model training is performed using only one sample from $q_\phi(z)$ and $q_\phi(a)$, i.e. $L = 1$; it is currently well-known that using $L > 1$ samples in the context of amortized variational inference does not yield any noticeable improvement over $L = 1$, as long as the used batch-size is quite large (effectively, at least 100 samples) [129, 98, 55].

Our source codes have been developed in Python, and make use of the Theano[1] [128], Lasagne[2], and Parmesan[3] libraries, as well as source code from the authors of [132][4].

4.4.1 Workflow recognition dataset

We first consider a public benchmark dataset involving action recognition of humans, namely the Workflow Recognition database [148]. Specifically, we use the first two workflows pertaining to car assembly (see [148] for more details). The frame-level tasks to recognize in

[1] http://deeplearning.net/software/theano/
[2] https://github.com/Lasagne/Lasagne
[3] https://github.com/casperkaae/parmesan
[4] https://github.com/larsmaaloee/auxiliary-deep-generative-models

these workflows are the following:

1. Worker 1 picks up part 1 from rack 1 (upper) and places it on the welding cell; mean duration is 8-10 sec.
2. Worker 1 and worker 2 pick part 2a from rack 2 and place it on the welding cell.
3. Worker 1 and worker 2 pick part 2b from rack 3 and place it on the welding cell.
4. Worker 2 picks up spare parts 3a, 3b from rack 4 and places them on the welding cell.
5. Worker 2 picks up spare part 4 from rack 1 and places it on the welding cell.
6. Worker 1 and worker 2 pick up part 5 from rack 5 and place it on the welding cell.

Feature extraction is performed as follows: To extract the spatiotemporal variations, we use pixel change history images to capture the motion history (see, e.g., [149]), and compute the complex Zernike moments $A_{00}, A_{11}, A_{20}, A_{22}, A_{31}, A_{33}$, $A_{40}, A_{42}, A_{44}, A_{51}, A_{53}, A_{55}$, $A_{60}, A_{62}, A_{64}, A_{66}$, for each of which we compute the norm and the angle. Additionally the center of gravity and the area of the found blobs are also used. In total, this feature extraction procedure results in 31-dimensional observation vectors. Zernike moments are calculated in rectangular regions of interest of approximately 15K pixels in each image to limit the processing and allow real time feature extraction (performed at a rate of approximately 50-60 fps). In our experiments, we use a total of 40 sequences representing full assembly cycles and containing at least one of the considered behaviors, with each sequence being approximately 1K frames long. Frame annotation has been performed manually. We provide the so-obtained test error rates of the evaluated methods in Table 4.1. As we observe, our approach yields a statistically significant improvement over the competition.

4.4.2 Honeybee dance dataset

Further, we evaluate our method using the Honeybee Dance dataset [150]; it contains video sequences of honeybees which communicate the location and distance to a food source through a dance that takes place within the hive. The dance can be decomposed into three different movement patterns that must be recognized by the evaluated algorithms: waggle, right-turn, and left-turn. During the waggle dance, the bee moves roughly in a straight line while rapidly shaking its body from left to right; the duration and orientation of this phase correspond to the

Table 4.1: Activity recognition experiments: Test error (%) of the evaluated methods (means and standard deviations over multiple repetitions).

Method	Workflow Recognition	Honeybee Dance
M1+TSVM	22.12 ($\pm$0.05)	45.48 ($\pm$0.11)
M1+M2	20.58 ($\pm$0.05)	38.62 ($\pm$0.10)
VAT	17.29 ($\pm$0.05)	36.13 ($\pm$0.14)
SDGM	13.90 ($\pm$0.04)	30.38 ($\pm$0.14)
AsyDGM	13.02 ($\pm$0.03)	24.11 ($\pm$0.12)

distance and the orientation to the food source. At the endpoint of a waggle dance, the bee turns in a clockwise or counter-clockwise direction to form a turning dance. Our dataset consists of six video sequences with lengths 1058, 1125, 1054, 757, 609, and 814 frames, respectively, and is based on the raw pixel change history images, without further preprocessing, contrary to the previous experiment; this renders this experimental scenario more challenging for all the evaluated deep generative models. The obtained results are provided in Table 4.1. We observe that our approach yields a clear improvement over the competition, including an almost 20% improvement over the second best performing method.

4.4.3 Yearly song classifciation using audio features

In this experiment, we consider application of our method to automatic prediction of a song track's release year. This problem entails surprisingly challenging complexity issues, stemming from the great diversity of style and genre of the songs released each year. Under this motivation, we utilize a subset of the "Million song dataset" benchmark [151], which comprises $515,345$ tracks with available release year information (both training and test sets). The tracks are mostly western, commercial tracks ranging from 1922 to 2011, with a peak in the year 2000 and onwards. Apart from the year, the dataset provides 90 additional representative features; of these 90 attributes, 12 are timbre average and 78 are timbre covariance, all extracted from the timbre features. We use these 90-dimensional feature vectors as the observations presented to the evaluated methods.

In our experiments, our goal is to differentiate between songs written in the 1980s, 1990s, and 2000s. For this purpose, we randomly select 10% of the training set songs released in these decades as our labeled training data; the remainder of the available training data pertaining

Table 4.2: Song classification experiments: Test error (%) of the evaluated methods (means and standard deviations over multiple repetitions).

Method	Performance
M1+TSVM	38.12 ($\pm$0.12)
M1+M2	36.49 ($\pm$0.13)
VAT	37.44 ($\pm$0.13)
SDGM	33.16 ($\pm$0.11)
AsyDGM	28.30 ($\pm$0.10)

to these decades is used as our unlabeled training dataset (i.e., with their labels considered missing). Subsequently, all methods are evaluated on the grounds of correctly classifying the test set tracks (included in the dataset) that pertain to these three decades. The obtained results are provided in Table 4.2. As we observe, the proposed approach outperforms all its competitors, yielding notable and statistically significant performance differences.

4.4.4 Image classifcation benchmarks

Finally, we evaluate our method on two popular benchmark datasets dealing with image classification, namely MNIST and *small* NORB. The popularity of these datasets facilitates transparency in our comparisons with the existing literature. MNIST comprises a total of 60,000 training samples, which constitute images of handwritten digits, with size 28$\times$28. On the other hand, the small NORB dataset comprises 24,300 training samples and an equal amount of test samples; these constitute images of size 32$\times$32, and are distributed across 5 classes: animal, human, plane, truck, car.

In Table 4.3, we report the obtained performance of our method, alongside the number of retained training data labels in each case. We also cite the performances of related methods reported in the recent literature. As we observe, our method turns out to yield results merely comparable to SDGM in the case of the MNIST dataset. This outcome is probably reasonable, since MNIST is a rather easy dataset, with clear underlying structural patterns, and absence of artifacts such as skewness or outliers. Therefore, one would not expect substantial room for improvement obtained by means of a method designed to account for such artifacts.

The obtained comparative empirical outcome changes in the case of the NORB dataset, where our method does yield a statistically significant performance improvement over the second best

Table 4.3: Image classification benchmarks: Test error (%) of the evaluated methods (means and standard deviations over multiple repetitions).

Method	MNIST	NORB
#Training Labels	100	1000
M1+TSVM	11.82 ($\pm$0.25)	18.79 ($\pm$0.05)
M1+M2	3.33 ($\pm$0.14)	-
VAT	2.12	9.88
SDGM	1.32 ($\pm$0.07)	9.40 ($\pm$0.04)
AsyDGM	1.34 ($\pm$0.08)	9.03 ($\pm$0.02)

performing method. Indeed, one could claim that this performance difference is not as high as in the previously considered experimental scenarios. We argue though that this outcome could be easily expected: The nature of NORB, which comprises images of some simple objects without significant clutter, is much less likely to give rise to modeling problems related with skewness, atypical data, and outliers. Such problems though can become extremely prominent when dealing with noisy signals such as music, as well as when dealing with activity recognition in video sequences, where such artifacts are much more common.

4.4.5 A note on computational complexity

We underline that the extra computational costs of our method are solely associated with learning of the skewness vectors $\boldsymbol{\lambda}$. These costs are only limited to the training algorithm of the model, and do not constitute a significant complexity increase, due to our approximation (4.47). Hence, computational complexity for the training algorithm of our method is comparable to SDGM; indeed, we have experimentally observed requirements of the same order of magnitude in computational time. Note also that training algorithm convergence has been empirically found to be similarly fast in both the cases of our model and of its main competitor, i.e. SDGM, in all the conducted experiments. On the other hand, the computational performance of our method in test time is (almost) identical to SDGM, since both approaches essentially require the same set of feedforward computations.

4.5 Conclusions

This Chapter constitutes an attempt to increase the effectiveness and representation power of the learned latent variable posteriors of DGMs in a principled, rather than an ad hoc, fashion. To this end, we drew inspiration from recent developments in the field of multivariate analysis: It has been recently shown that shallow, factor analysis-type, latent variable models are capable of yielding a significantly increased representation power by postulating latent variables with skew-elliptical distributions. On this basis, we examined whether similar benefits could be obtained for DGMs, by introducing an asymmetric DGM formulation, based on rMSN-distributed latent variables.

Since in this work we focused on the problem of semi-supervised learning, we exhibited the derivation of our approach in the context of a graphical formulation also adopted by the recently proposed SDGM approach. To allow for the derivation of an elegant inference algorithm for our model, we utilized a three-level hierarchical representation of the rMSN distribution, inspired from [143]. We examined the efficacy of our approach in several experimental scenarios, using benchmark datasets. As we showed, our method proves to be more effective than the competition in terms of modeling and predictive performance when artifacts such as skewness and outliers are prevalent in the observed data. These empirical results corroborate our theoretical claims.

Chapter 5

Deep Learning with t-Exponential Bayesian Kitchen Sinks

As we have already discussed, Bayesian learning has been recently considered as an effective means of accounting for uncertainty in trained deep network parameters. On the other hand, shallow models that compute weighted sums of their inputs, after passing them through a bank of arbitrary randomized nonlinearities, have been recently shown to enjoy good test error bounds that depend on the number of nonlinearities. Inspired from these advances, in this Chapter we examine novel deep network architectures, where each layer comprises a bank of arbitrary nonlinearities, linearly combined using multiple alternative sets of weights. Our proposed networks are efficiently trained by means of variational inference based on a t-divergence measure; this generalizes the Kullback-Leibler divergence in the context of the t-exponential family of distributions. We extensively evaluate our approach using several challenging benchmarks, and provide comparative results to related state-of-the-art techniques.

The aforementioned innovative approach dubbed as DtBKS is in press for the journal ”Expert Systems with Applications” under the title ”Deep Learning with t-Exponential Bayesian Kitchen Sinks”.

5.1 Introduction

DNNs have experienced a resurgence of interest; this is due to recent breakthroughs in the field that offer unprecedented empirical results in several challenging real-world tasks, such as

image and video understanding [152], natural language understanding and generation [153], and game playing [154]. Most DNN models are trained by means of some variant of the backpropagation (BP) algorithm. However, despite all these successes, BP suffers from the major shortcoming of being able to obtain only point-estimates of the trained networks. This fact results in the trained networks generating predictions that do not account for uncertainty, e.g. due to the limited or sparse nature of the available training data.

A potential solution towards the amelioration of these issues consists in treating some network components under the Bayesian inference rationale, instead of stochastic optimization [155]. Specifically, in this Chapter we are interested in inference of the network feature functions. In the literature, this is effected by considering them as stochastic latent variables imposed some mathematically convenient Gaussian process prior [156]. On this basis, one proceeds to infer the corresponding posteriors, based on the available training data. To the latter end, and with the goal of combining accuracy with computational efficiency, expectation-propagation [157], mean-field [156], and probabilistic backpropagation [158] have been used.

One of the main driving forces behind the unparalleled data modeling and predictive performance of modern DNNs is their capability of effectively learning to extract informative, high-level, hierarchical representations of observed data with latent structure [18]. Nevertheless, DNNs are not the only class of models that entail this sort of functionality. Indeed, major advances in machine learning have long been dominated by the development of shallow architectures that compute weighted sums of feature functions; the latter generate nonlinear representations of their input data, which can be determined under a multitude of alternative rationales. For instance, models that belong to the family of support vector machines (SVMs) [159] essentially compute weighted sums of positive definite kernels; boosting algorithms, such as AdaBoost [160], compute weighted sums of weak learners, such as decision trees. In all cases, both the feature functions as well as the associated weights are learned under an empirical risk minimization rationale; for instance, the hinge loss is used in the context of SVMs, while AdaBoost utilizes the exponential loss.

In the same vein, the machine learning community has recently examined a bold, yet quite promising possibility: postulating weighted sums of random kitchen sinks (RKS) [161]. The main rationale of this family of approaches essentially consists in randomly drawing the employed feature functions (nonlinearities), and limiting model training to the associated (scalar) weights. Specifically, the (entailed parameters of the) postulated nonlinearities are randomly sampled from an appropriate density, which is a priori determined by the practitioners according to some assumptions [162, 163]. As it has been shown, both through theoretical

analysis as well as some empirical evidence, such a modeling approach does not yield much inferior performance for a trained classifier compared to the mainstream approach of optimally selecting the employed nonlinearities. In addition, predictive performance is shown to increase with the size of the employed bank of randomly drawn nonlinearities.

Inspired from these advances, this Chapter introduces a fresh regard towards DNNs: We formulate each DNN layer as a bank of random nonlinearities, which are linearly combined in multiple alternative fashions. This way, the postulated models eventually yield a hierarchical cascade of informative representations of their multivariate observation inputs, that can be used to effectively drive a penultimate regression or classification layer. At each layer, the employed bank of random nonlinearities is sampled from an appropriate postulated density, in a vein similar to RKS. However, in order to alleviate the burden of having to manually design these densities, we elect to infer them in a Bayesian sense. In addition, we elect not to obtain point-estimates of the weights used for combining the drawn nonlinearities. On the contrary, we perform Bayesian inference over them, so as to better account for model uncertainty.

As already discussed, Bayesian inference for DNN type models can be performed under various alternative paradigms. Here, we resort to variational inference ideas, which consist in searching for a proxy in an analytically solvable distribution family that approximates the true underlying posterior distribution. To measure the closeness between the true and the approximate posterior, the relative entropy between these two distributions is used. Specifically, under the typical Gaussian assumption, one can use the Shannon-Boltzmann-Gibbs (SBG) entropy, whereby the relative entropy yields the well known Kullback-Leibler (KL) divergence [164]. However, real world phenomena tend to entail densities with heavier tails than the simplistic Gaussian assumption.

To account for these facts, in this work we exploit the t-exponential family[1], which was first proposed by Tsallis and co-workers [165, 166, 167], and constitutes a special case of the more general φ-exponential family [168, 169, 170]. Of specific practical interest to us is the Students'-t density; this is a bell-shaped distribution with heavier tails and one more parameter (degrees of freedom - DOF) compared to the normal distribution, and tends to a normal distribution for large DOF values [171]. Hence, it provides a much more robust approach to the fitting of models with Gaussian assumptions. On top of these merits, the t-exponential family also gives rise to a new t-divergence measure; this can be used for performing approximate inference in a fashion that better accommodates heavy-tailed densities (compared to standard

[1] Also referred to as the q-exponential family or the Tsallis distribution.

KL-based solutions) [172].

To summarize, we formulate a hierarchical (multilayer) model, each layer of which comprises a bank of random feature functions (nonlinearities). Each nonlinearity is presented with the layer's input, and generates scalar outputs. These outputs are linearly combined by using multiple alternative sets of mixing weights, to produce the (multivariate) layer's output. At each layer, the postulated nonlinearities are drawn from an appropriate (posterior) density, which is inferred from the data (as opposed to the requirement of conventional RKS that the practitioners manually specify this distribution). Indeed, both the posterior density of the nonlinearities, as well as the posterior over the mixing weights, are inferred in an approximate Bayesian fashion. In order to allow for our model to account for heavy tails, we postulate that the sought densities belong to the t-exponential family, specifically they constitute (multivariate) Student's-t densities. On this basis, we conduct approximate inference by optimizing a t-divergence functional, that better leverages the advantages of the t-exponential family. We dub our proposed approach the Deep t-Exponential Bayesian Kitchen Sinks(DtBKS) model.

The remainder of this Chapter is organized as follows: In the following section, we provide a brief overview of the methodological background of our approach. Subsequently, we introduce our approach, and derive its training and inference algorithms. Then, we perform the experimental evaluation of our approach, and illustrate its merits over the current state-of-the-art. Finally, in the concluding section of this chapter, we summarize our contribution and discuss our results.

5.2 Methodological Background

5.2.1 Weighted Sums of Random Kitchen Sinks

Consider the problem of fitting a function $f : \mathbb{R}^\delta \to \mathscr{Y}$ to a training dataset comprising N input-output pairs $\{x_n, y_n\}_{n=1}^N$, drawn i.i.d. from some unknown distribution $P(x,y)$, with $x_n \in \mathbb{R}^\delta$ and $y_n \in \mathscr{Y}$. In essence, this fitting problem consists in finding a function f that minimizes the empirical risk on the training data

$$R[f] = \frac{1}{N} \sum_{n=1}^{N} c\left(f(x_n), y_n\right) \tag{5.1}$$

where the cost function $c(f(x), y)$ defines the penalty we impose on the deviation between the

prediction $f(x)$ and the actual value y.

The main underlying idea of data modeling under the weighted sums of RKS rationale consists in postulating functions of the form

$$f(x) = \sum_{s=1}^{S} \alpha_s \xi(x; \omega_s) \tag{5.2}$$

where the $\{\alpha_s\}_{s=1}^{S}$ are mixing weights, while the nonlinear feature functions ξ are parameterized by some vector $\omega \in \Omega$, and are bounded s.t.: $|\xi(x;\omega_s)| \leq 1$. Specifically, the vectors ω_s are samples drawn from an appropriate probability distribution $p(\omega)$ with support in Ω, whence we have $\xi : \mathbb{R}^\delta \times \Omega \rightarrow [-1, 1]$.

As an outcome of this construction, the fitted function $f(x)$ can be essentially viewed as a weighted sum of a bank of random nonlinearities, ξ, drawn by appropriately sampling from a selected probability distribution $p(\omega)$. On this basis, the fitting procedure reduces to selecting the weight values $\{\alpha_s\}_{s=1}^{S}$, such that we minimize the empirical risk (5.1); typically, a quadratic loss function is employed to this end. As shown in [57], for $S \rightarrow \infty$, weighted sums of RKS yield predictive models whose true risk is near the lowest true risk attainable by an infinite-dimensional class of functions with optimally selected parameter sets, ω.

Weighted sums of RKS give rise to a modeling paradigm with quite appealing properties: It allows for seamlessly and efficiently employing arbitrarily complex feature functions ξ, since model fitting is limited to the mixing weights; this way, we can easily experiment with feature functions that do not admit simple fitting procedures. On the other hand, RKS require one to (manually) design appropriate distributions $p(\omega)$ to draw the employed nonlinearities from; in real-world data modeling tasks, this might prove quite challenging a task.

5.2.2 The Student's-*t* Distribution

The adoption of the multivariate Student's-*t* distribution provides a way to broaden the Gaussian distribution for potential outliers. The probability density function (pdf) of a Student's-*t* distribution with mean vector μ, covariance matrix Σ, and $\nu > 0$ degrees of freedom is [173]

$$t(y_t; \mu, \Sigma, \nu) = \frac{\Gamma\left(\frac{\nu+\delta}{2}\right) |\Sigma|^{-1/2} (\pi\nu)^{-\delta/2}}{\Gamma(\nu/2)\{1 + d(y_t, \mu; \Sigma)/\nu\}^{(\nu+\delta)/2}} \tag{5.3}$$

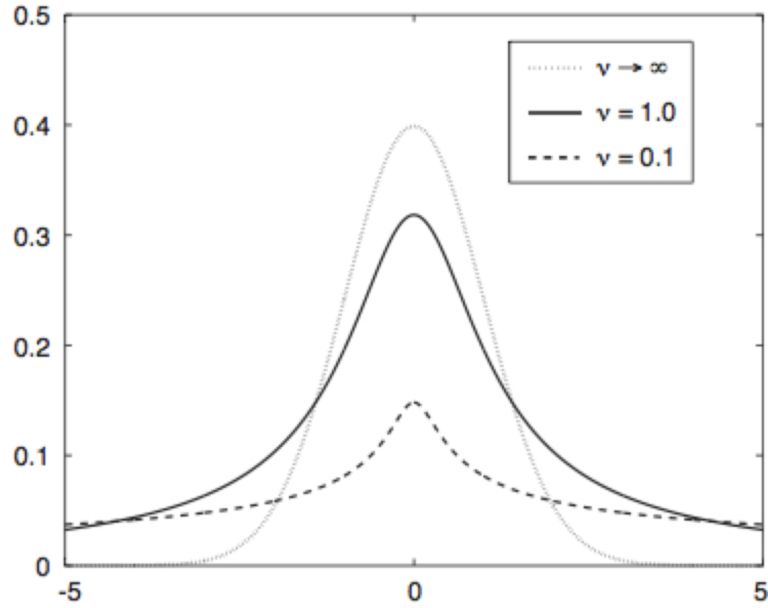


Figure 5.1: Univariate Student's-t distribution $t(y_t;\mu,\Sigma,\nu)$, with μ, Σ fixed, for various values of ν [176].

where δ is the dimensionality of the observations y_t, $d(y_t,\mu;\Sigma)$ is the squared Mahalanobis distance between y_t,μ with covariance matrix Σ

$$d(y_t,\mu;\Sigma) = (y_t-\mu)^T\Sigma^{-1}(y_t-\mu) \tag{5.4}$$

and $\Gamma(s)$ is the Gamma function, $\Gamma(s) = \int_0^\infty e^{-t}z^{s-1}dz$.

A graphical illustration of the univariate Student's-t distribution, with μ, Σ fixed, and for various values of the degrees of freedom ν, is provided in figure 5.1. As we observe, as $\nu \to \infty$, the Student's-t distribution tends to a Gaussian with the same μ and Σ. On the contrary, as ν tends to zero, the tails of the distribution become longer, thus allowing for a better handling of potential outliers, without affecting the mean or the covariance of the distribution [174, 175].

5.2.3 The t-Divergence

The t-divergence was introduced in [172] as follows:

Definition 1. The t-divergence between two distributions, $q(h)$ and $p(h)$, is defined as

$$D_t(q||p) = \int \tilde{q}(h)\log_t q(h)\mathrm{d}h - \tilde{q}(h)\log_t p(h)\mathrm{d}h \tag{5.5}$$

where $\tilde{q}(h)$ is called the escort distribution of $q(h)$, and is given by

$$\tilde{q}(h) = \frac{q(h)^t}{\int q(h)^t \mathrm{d}h},\ t \in \mathbb{R} \tag{5.6}$$

Importantly, the divergence $D_t(q||p)$ preserves the following two properties:

- $D_t(q||p) \geq 0,\ \forall q, p$. The equality holds only for $q = p$.
- $D_t(q||p) \neq D_t(p||q)$.

As discussed in [172], by leveraging the above definition of the t-divergence, $D_t(q||p)$, one can establish an advanced variational inference framework, much more appropriate for fitting heavy-tailed densities. We exploit these benefits in developing the training algorithm of the proposed D*t*BKS model, as we shall explain in the following Section.

5.3 Proposed Approach

5.3.1 Model Formulation

Let us consider a D*t*BKS model with input variables $x \in \mathbb{R}^\delta$ and output (predictable) variables y, comprising L layers. In figure 5.2, we provide a graphical illustration of the proposed configuration of one D*t*BKS model layer. Each layer, l, is presented with an input vector h^{l-1}, generated from the preceding layer; at the first layer, this vector is the model input, $h^0 \triangleq x$. This is fed into a bank comprising S randomly drawn feature functions, $\{\xi_s^l(h^{l-1})\}_{s=1}^S$. Specifically, these functions are nonlinearities parameterized by some random vector ω; i.e., $\xi_s^l(h^{l-1}) = \xi(h^{l-1};\omega_s^l)$. The used samples ω_s^l are drawn from an appropriate density, which is inferred in a Bayesian sense, as we shall explain next.

This way, we eventually obtain a set of S univariate signals, which we linearly combine to obtain the layer's output. Specifcally, we elect to (linearly) combine them in multiple alternative ways, with the goal of obtaining a potent, multidimensional, high-level representation of the original observations. These alternative combinations are computed via a mixing weight

matrix, $W^l \in \mathbb{R}^{\eta \times S}$, where η is the desired dimensionality of the layer's output (i.e., the postulated number of alternative linear combinations).

Eventually, the output vectors at the layers $l \in \{1, \ldots, L-1\}$ yield

$$h^l = W^l [\xi(h^{l-1}; \omega_s^l)]_{s=1}^S \in \mathbb{R}^\eta \tag{5.7}$$

where $[\chi_s]_{s=1}^S$ denotes the vector-concatenation of the set $\{\chi_s\}_{s=1}^S$. Turning to the penultimate layer of the model, we consider that the output variables are imposed an appropriate conditional likelihood function, the form of which depends on the type of the addressed task. Specifically, in the case of regression tasks, we postulate a multivariate Gaussian of the form

$$p(y|x) = \mathcal{N}\left(y \middle| W^L [\xi(h^{L-1}; \omega_s^L)]_{s=1}^S, \sigma_y^2 I\right) \tag{5.8}$$

where σ_y^2 is the noise variance. On the other hand, in case of classification tasks, we assume

$$p(y|x) = \text{Softmax}\left(y \middle| W^L [\xi(h^{L-1}; \omega_s^L)]_{s=1}^S\right) \tag{5.9}$$

This concludes the definition of our model. For brevity, we shall omit the layer indices, l, in the remainder of this chapter, wherever applicable.

5.3.2 Model Training

To allow for inferring the distribution that the employed feature functions, ξ, must be drawn from, we first consider that the vectors ω that parameterize them are Student's-*t* distributed latent variables. We employ the same assumption for the weight matrices, W, which we also want to infer in a Bayesian fashion, so as to account for model uncertainty. Specifically, we start by imposing a simple, zero-mean Student's-*t* prior distribution over them, with tied degrees of freedom, at each model layer:

$$p(\omega) = t(\omega | 0, I, \nu) \tag{5.10}$$

$$p(W) = t(\text{vec}(W) | 0, I, \nu) \tag{5.11}$$

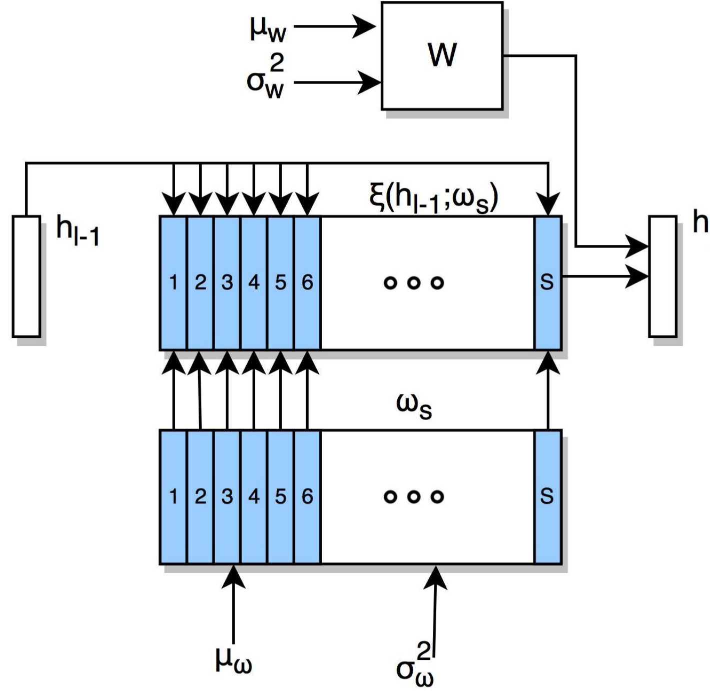


Figure 5.2: Graphical illustration of the configuration of one DtBKS model layer.

where vec$(\cdot)$ is the matrix vectorization operation, and $\nu > 0$ is the degrees of freedom hyperparameter of the imposed priors.

On this basis, we seek to devise an efficient means of inferring the corresponding posterior distributions, given the available training data. To this end, we postulate that the sought posteriors approximately take the form of Student's-t densities with means, diagonal covariance matrices, and degrees of freedom inferred from the data. Hence, we have:

$$q(\omega;\phi) = t(\omega|\mu_{\omega}, \text{diag}(\sigma_{\omega}^2), \nu_{\omega}) \tag{5.12}$$

$$q(W;\phi) = t(\text{vec}(W)|\mu_W, \text{diag}(\sigma_W^2), \nu_W) \tag{5.13}$$

where $\phi = \{\mu_i, \sigma_i^2, \nu_i\}_{i\in\{\omega,W\}}$, and $\nu_i > 0, \forall i$, for all model layers.

To perform training in a way the best exploits the heavy tails of the developed model, we minimize the t-divergence between the sought variational posterior and the postulated joint density over the observed data and the model latent variables. Thus, the proposed model training objective becomes

$$q(\omega;\phi), q(W;\phi) = \underset{q(\cdot)}{\arg\min} \quad D_t\left(q(\omega;\phi), q(W;\phi) || p(y;\omega,W)\right) \tag{5.14}$$

By application of simple algebra, the expression of the t-divergence in (5.14) yields

$$\begin{aligned} &D_t\big(q(\omega;\phi), q(W;\phi)||p(y;\omega,W)\big) = \\ &= D_t\left(q(\omega;\phi)||p(\omega)\right) + D_t\left(q(W;\phi)||p(W)\right) - \mathbb{E}_{\tilde{q}(\omega,W;\phi)}[\log p(y|x)] \end{aligned} \tag{5.15}$$

where $\tilde{q}(\omega,W;\phi)$ is the escort distribution of the sought posterior, and the t-divergence terms pertaining to the parameters ω and W are summed over all model layers. Then, following [172], and based on (5.12)-(5.13), we obtain that the t-divergence expressions in (5.15) can be written in the following form:

$$D_t\left(q(\theta;\phi)||p(\theta)\right) = \sum_i \left\{ \frac{\Psi_{qi}}{1-t}\left(1+\frac{1}{\nu_\theta}\right) - \frac{\Psi_p}{1-t} \quad \left(1+\frac{[\sigma_\theta^2]_i + [\mu_\theta]_i^2}{\nu}\right) \right\} \tag{5.16}$$

where $\theta \in \{\omega, \text{vec}(W)\}$, $[\zeta]_i$ is the ith element of a vector ζ, we denote

$$\Psi_{qi} \triangleq \left(\frac{\Gamma(\frac{\nu_\theta+1}{2})}{\Gamma(\frac{\nu_\theta}{2})(\pi\nu_\theta)^{1/2}[\sigma_\theta]_i} \right)^{-\frac{2}{\nu_\theta+1}} \tag{5.17}$$

$$\Psi_p \triangleq \left(\frac{\Gamma(\frac{\nu+1}{2})}{\Gamma(\frac{\nu}{2})(\pi\nu)^{1/2}} \right)^{-\frac{2}{\nu+1}} \tag{5.18}$$

and the free hyperparameter t is set as suggested in [177], yielding:

$$t = \frac{2}{1+\nu_\theta} + 1 \tag{5.19}$$

As we observe from the preceding discussion, the expectation of the conditional log-likelihood of our model, $\mathbb{E}_{\tilde{q}(\omega,W;\phi)}[\log p(y|x)]$, is computed with respect to the escort distributions of the sought posteriors. Therefore, at training time, the banks of the employed feature functions (i.e., the samples of their parameters, $\{\omega_s\}_{s=1}^S$), must be drawn from the escort distributions of the derived posteriors.

Turning to the employed mixing weight matrices, W, our consideration of them being latent variables with an inferred posterior perplexes computation of the expressions (5.7)-(5.9); indeed, it requires that we compute appropriate posterior expectations of these expressions

with respect to W. To circumvent this problem, we draw multiple samples of the mixing weight matrices, $\{W_s\}_{s=1}^S$, in an MC fashion, and average over the corresponding outcomes to compute the model output. At training time, Eq. (5.15) implies that these samples must also be drawn from the escort distributions of the derived posteriors.

Based on the previous results, and following [172], we can easily obtain the expressions of these escort distributions of the derived posteriors, that we need to sample from at training time. Specifically, it is easy to show that these escort distributions yield a factorized form, that reads:

$$\tilde{q}(\theta;\phi) = t\left(\theta|\mu_\theta, \frac{\nu_\theta}{\nu_\theta+2}\mathrm{diag}(\sigma_\theta^2), \nu_\theta+2\right), \forall\theta \in \{\omega, \mathrm{vec}(W)\} \tag{5.20}$$

Notably, our variational inference algorithm yields an MC estimator of the proposed D*t*BKS model. Unfortunately, MC estimators are notorious for their vulnerability to unacceptably high variance. In this work, we resolve these issues by adopting the reparameterization trick of [129], adapted to the t-exponential family. This trick consists in a smart reparameterization of the MC samples, $\{\theta_s\}_{s=1}^S$, drawn from a distribution $\theta \sim q(\theta;\phi)$; this is obtained via a differentiable transformation $g_\phi(\varepsilon)$ of an (auxiliary) random variable ε with low variance:

$$\theta = g_\phi(\varepsilon) \quad \text{with} \quad \varepsilon \sim p(\varepsilon) \tag{5.21}$$

In our case, the smart reparameterization of the MC samples drawn from the Student's-t escort densities (5.20) yields the expression:

$$\theta_s = \theta(\varepsilon_s) = \mu_\theta + \left(\frac{\nu_\theta}{\nu_\theta+2}\right)^{1/2} \sigma_\theta \varepsilon_s \tag{5.22}$$

where ε_s is random Student's-t noise with unitary variance:

$$\varepsilon_s \sim t(0, I, \nu_\theta+2) \tag{5.23}$$

On this basis, at training time, we replace the samples of both the weight matrices, W, as well as the feature function parameters, ω, with the expression (5.22), where sampling is performed with respect to the low-variance random variable ε. Then, the resulting (reparameterized) t-divergence objective (5.15) can be minimized by means of any off-the-shelf stochastic optimization algorithm, yielding low variance estimators. To this end, in this work we utilize AdaM [84]. We initialize the sought posterior hyperparameters by setting them equal to the hyperparameters of the imposed priors.

5.3.3 Inference Algorithm

Having obtained a training algorithm for our proposed DtBKS model, we can now proceed to elaborate on how inference is performed using our method. To this end, it is needed that we compute the posterior expectation of the model's output, as usual when performing Bayesian inference. Thus, at inference time, we need to draw samples from the derived posteriors, (5.12) and (5.13), in an MC fashion. This entails drawing from the posteriors, at each layer, of a set comprising S samples of: (i) the vectors ω that parameterize the employed feature functions; and (ii) the mixing weight matrices, W, used to combine the outputs of the drawn banks of feature functions. Note that this is in contrast to the training algorithm of DtBKS, where the use of the t-divergence objective (5.15) gives rise to the requirement of drawing from the associated escort distributions (5.20), while the need of training reliable estimators requires utilization of the reparameterization trick.

5.4 Experimental Evaluation

In this Section, we perform a thorough experimental evaluation of our proposed DtBKS model. We provide a quantitative assessment of the efficacy, the effectiveness, and the computational efficiency of our approach, combined with deep qualitative insights into few of its key performance characteristics. To this end, we consider several benchmarks from the UCI machine learning repository (UCI-Rep) [178] that pertain to both regression and classification tasks, as well as the well-known InfiMNIST classification benchmark [179]. The considered datasets, as well as their main characteristics (i.e., their number of training examples, N, and input dimensionality, δ) are summarized in Table 5.1a for regression and Table 5.1b for classification tasks.

With the exception of the ISOLET dataset from UCI-Rep, as well as InfiMNIST, the rest of the considered benchmarks do not provide a split into training and test sets. In these cases, we account for this lack by running our experiments 20 times, with different random data splits into training and test sets, and compute performance means and standard deviations; we use a randomly selected 90% of the data for model training, and the rest for evaluation purposes. In the case of regression tasks, we use the root mean square error (RMSE) as our performance metric; we employ the predictive error rate for the considered classification benchmarks.

To obtain some comparative results, we also evaluate an existing alternative approach for

Table 5.1: Obtained performance for best model configuration (the lower the better)

Dataset	N	δ	DtBKS	DGP	Dropout
Boston Housing	506	13	0.2939 ± 0.04 ($L = 2, \eta = 3$)	0.3897 ± 0.1	0.2516 ± 0.06
Concrete	1030	8	0.3213 ± 0.02 ($L = 2, \eta = 5$)	0.4501 ± 0.03	0.3228 ± 0.03
Energy	768	8	0.1285 ± 0.01 ($L = 2, \eta = 6$)	0.1636 ± 0.02	0.1322 ± 0.01
Power Plant	9568	4	0.2366 ± 0.01 ($L = 3, \eta = 4$)	0.2401 ± 0.01	0.2236 ± 0.01
Protein	45730	9	0.6113 ± 0.01 ($L = 2, \eta = 9$)	0.6734 ± 0.01	0.7453 ± 0.01
Wine (White)	4898	11	0.7684 ± 0.02 ($L = 2, \eta = 11$)	0.8072 ± 0.02	0.7609 ± 0.02
Wine (Red)	1588	11	0.7564 ± 0.04 ($L = 2, \eta = 5$)	0.7791 ± 0.04	0.7570 ± 0.05

(a) Regression tasks

Dataset	N	δ	DtBKS	DGP	Dropout
Breast (wdbc)	569	30	0.0116 ± 0.01 ($L = 2, \eta = 21$)	0.0116 ± 0.01	0.0710 ± 0.06
ISOLET	7797	617	$0.0552 \pm NA$ ($L = 2, \eta = 205$)	$0.0654 \pm NA$	$0.1256 \pm NA$
Gas Sensor	13910	128	0.0136 ± 0.002 ($L = 2, \eta = 106$)	0.0094 ± 0.002	0.0688 ± 0.07
Parkinson's	197	22	0.0658 ± 0.05 ($L = 2, \eta = 15$)	0.0842 ± 0.05	0.0976 ± 0.09
Spam	4601	56	0.0543 ± 0.01 ($L = 2, \eta = 46$)	0.0517 ± 0.01	0.1629 ± 0.03
LSVT	126	310	0.1375 ± 0.07 ($L = 2, \eta = 51$)	0.3250 ± 0.11	0.3782 ± 0.17
InfiMNIST	8+ Million	784	$0.0093 \pm NA$ ($L = 2, \eta = 100$)	$0.0096 \pm NA$	$0.0096 \pm NA$

(b) Classification tasks

Bayesian inference of deep network nonlinearities, namely deep Gaussian processes (DGPs) [156]. Specifically, we evaluate the most recent variant of DGPs, presented in [180], which allows for the model to efficiently scale to large datasets. Finally, we also provide comparisons to a state-of-the-art Deep Learning approach, namely a Dropout network.

In all cases, our specification of the priors imposed on DtBKS considers a low value for the degrees of freedom hyperparameter, $\nu = 2.1$. In the case of regression tasks, we employ a noise variance equal to $\sigma_y^2 = \exp(-2)$. Turning to the selection of the form of the drawn feature functions, ξ, we consider a simple trigonometric formulation, which is inspired from the theory of random Fourier projections of RBF kernels [181]. Specifically, we postulate $\xi(x;\omega) = \frac{1}{2}\cos(\omega^T x) + \frac{1}{2}\sin(\omega^T x)$. For computational efficiency, we limit the number of drawn samples to 100, during both DtBKS training and inference on the test data. AdaM is run with the default hyperparameter values.

DGP is evaluated considering multiple selections of the number of Gaussian processes per layer, as well as the number of layers, using RBF kernels and arc-cosine kernels [180]; in each experimental case, we report results pertaining to the best-performing DGP configuration. Similar is the case for Dropout networks, which are evaluated considering multiple alternatives for the number of layers and the output size of each hidden layer; we employ ReLU nonlinearities [182].

Our source codes have been developed in Python, using the Tensorflow library [43]. We have also made use of a DGP implementation provided by M. Filippone[2]. We run our experiments on an Intel Xeon 2.5GHz Quad-Core server with 64GB RAM and an NVIDIA Tesla K40 GPU.

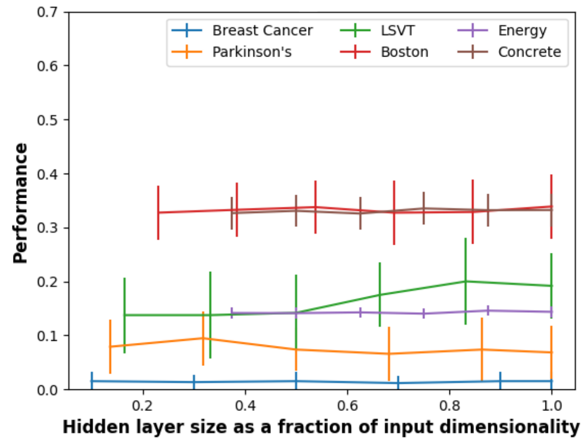
5.4.1 Comparative Results

We begin our exposition by providing the best empirical performance of our method, and showing how it compares to the competition. These outcomes have been obtained by experimenting with different selections for the number of DtBKS layers, L, and the output size of each hidden layer, η (i.e., for $l \in \{1, \dots, L-1\}$). Our results are outlined in Tables 5.1a and 5.1b; in all cases, we provide therein (in parentheses) the DtBKS model configuration that obtained the reported (best empirical) performance.

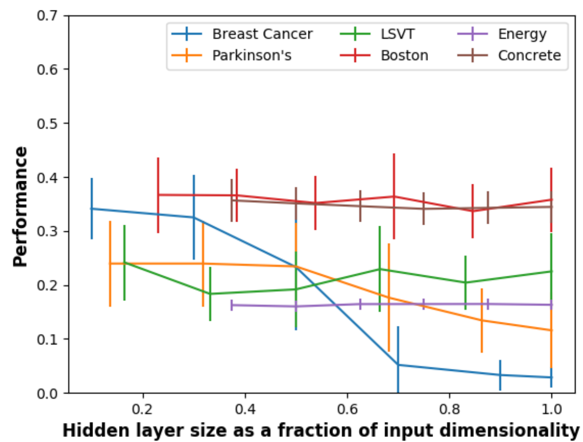
We observe that our approach outperforms DGP in the considered regression benchmarks; in

[2] `https://github.com/mauriziofilippone/deep_gp_random_features`

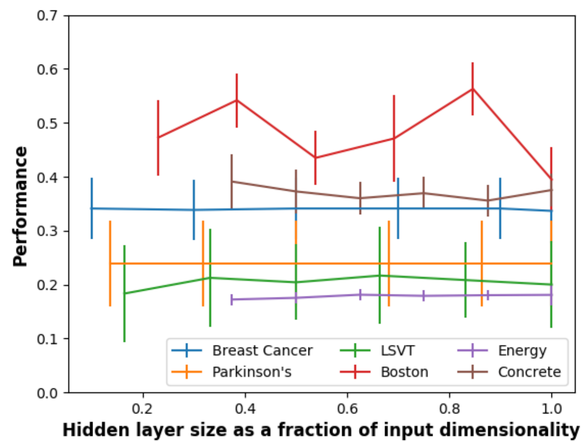
Figure 5.3: DtBKS performance fluctuation with the number of layers, L, and the output size of each hidden layer, η (as a fraction of input dimensionality, δ)



(a) $L = 2$



(b) $L = 3$



(c) $L = 4$

all cases, these empirical performance differences are found to be statistically significant, by running the paired Student's-t test. On the other hand, DtBKS outperforms DGP in only three out of the seven considered classification benchmarks, with statistically significant differences (according to the paired Student's-t test), while yielding comparable outcomes in the rest. In addition, DtBKS outperforms Dropout in all the considered classification benchmarks, except for InfiMNIST; the paired Student's-t test shows that these empirical performance differences are statistically significant. On the other hand, DtBKS significantly outperforms Dropout in the Protein regression benchmark, while yielding comparable performance in the rest considered regression tasks (according to the outcomes of the paired Student's-t test).

5.4.2 Further Investigation

Further, it is interesting to provide a feeling of how DtBKS model performance changes with the selection of the number of layers, L, and the dimensionality of each hidden layer, η (i.e., for $l \in \{1, \dots, L-1\}$). To examine these aspects, in figure 5.3 we plot model performance fluctuation with η, setting the number of layers equal to $L = 2, 3$, and 4, respectively, for few characteristic experimental cases. As we observe, DtBKS performance is significantly affected by both these selections. Note also that the associated performance fluctuation patterns of DtBKS are quite different among the illustrated examples. These findings are congruent with the behavior of all existing state-of-the-art deep learning approaches. It is also important to mention the high standard deviation of the observed performances in some cases where we set $L = 4$; we attribute this unstable behavior to overfitting due to insufficient training data.

5.4.3 Are t-Exponential Bayesian Kitchen Sinks More Potent Than Random Kitchen Sinks?

Finally, it is extremely interesting to examine how beneficial it is for DtBKS to infer a posterior distribution over the (random variables that parameterize the) employed feature functions, instead of using a simple, manually selected density. To examine this aspect, we repeat our experiments by drawing the vectors ω, that parameterize the feature functions, ξ, from the postulated simple priors, $p(\omega)$. Hence, we adopt an RKS-type rationale in drawing the feature functions, ξ, as opposed to utilizing the inferred posteriors, $q(\omega)$ [or their corresponding escort distributions, $\tilde{q}(\omega)$, during training].

Our findings are provided in Table 5.2; these results correspond to selections of the number

Dataset	Performance
Boston Housing	0.3199 ± 0.04
Concrete	0.3586 ± 0.04
Energy	0.1494 ± 0.01
Power Plant	0.2301 ± 0.01
Protein	0.7110 ± 0.01
Wine (White)	0.7787 ± 0.02
Wine (Red)	0.7720 ± 0.04
Breast Cancer Diagnostic (wdbc)	0.0188 ± 0.02
ISOLET	$0.2245 \pm NA$
Gas Sensor	0.0175 ± 0.003
Parkinson's	0.0895 ± 0.06
Spam	0.0748 ± 0.01
LSVT Voice Rehabilitation	0.1208 ± 0.04
InfiMNIST	$0.0603 \pm NA$

Table 5.2: DtBKS performance when replacing t-Exponential Bayesian Kitchen Sinks with Random Kitchen Sinks.

of layers, L, and the output size, η, similar to the values reported in Tables 5.1a and 5.1b. Our empirical evidence is quite conspicuous: (i) merely drawing the postulated nonlinearities from a simple prior, yet inferring a Student's-t posterior over the mixing weights, W, as discussed previously, yields notably competitive performance; (ii) inferring posteriors over the nonlinearities, under the discussed DtBKS rationale, gives a statistically significant boost to the obtained modeling performance, except for Power Plant and LSVT, where we reckon that overfitting is induced (due to insufficient training data availability).

5.4.4 Computational Complexity

Another significant aspect that affects the efficacy of a machine learning technique is its computational complexity. To investigate this aspect, we first scrutinize the derived DtBKS training algorithm, both regarding its asymptotic behavior, as well as in terms of its total computational costs. Our observations can be summarized as follows: For the model configurations yielding the performance statistics of Tables 5.1a and 5.1b, DtBKS takes on average 4 times longer than Dropout per training algorithm iteration, probably due to the entailed $\Gamma(\cdot)$

functions in (5.16), and their derivatives; DGP takes on average 2 times longer than Dropout. On the other hand, DtBKS training converges faster than all the considered competitors, while DGP converges much faster than Dropout. These differences are so immense that, as an outcome, the total training time of all the evaluated methods turns out to be comparable, in all the considered benchmarks. Finally, we have observed that DtBKS and DGP take similar time to generate one prediction as a Dropout network of the same size (number of parameters). This is a well-expected behavior, since feedforward computation in all approaches entails computational primitives with complexity of the same order of magnitude. Hence, we deduce that DtBKS yields the observed predictive performance improvement without undermining computational efficiency and scalability.

5.5 Conclusions

In this chapter, we introduced a fresh view towards deep learning, which consists in postulating banks of randomly drawn nonlinearities at each model layer. To alleviate the burden of having to manually specify the distribution these nonlinear feature functions are drawn from, we elected to infer them in a Bayesian sense. Specifically, we postulated that the sought posteriors constitute multivariate Student's-t densities. To allow for reaping the most out of the heavy tails of Student's-t densities, we performed variational Bayesian inference for our model under a novel objective function construction. This was based on a t-divegence functional, which better accommodates heavy-tailed densities, compared to the typically used KL divergence.

We exhaustively evaluated our approach using challenging benchmark datasets; we offered thorough insights into its key performance characteristics. This way, we illustrated that our proposed approach outperforms the existing alternatives in terms of predictive accuracy, without undermining the overall computational scalability, both in terms of training time and of prediction generation time. We also showed that data-driven inference of a posterior distribution from which we can draw the employed banks of nonlinearities yields better results than drawing from a simple prior.

One research direction that we have not considered in this work concerns the possibility of imposing nonelliptical distributions on the postulated latent variables, which allow to account for skewness in a fashion similar, e.g., to [54]. These opportunities remain to be explored in our future research.

Chapter 6

Future Endeavors

As we have exhibited in this thesis, in many real world scenarios, we depend on mathematical models to assist us in various challenging tasks. Machine learning advancements have made possible to improve these models and in some cases even exceed human counterparts. A question arises from this context: Can we derive a perfect model that can solve any real world problem. From our perspective, a perfect model is achieved when it has the plasticity needed to flexibly deal with uncertainty. Bayesian inference is a tool towards that specific goal. It can efficiently counteract the epistemic uncertainty of the observable data. In this thesis, we have explored diverse utilizations of Bayesian inference in model training. As we have shown, our approach can indeed improve the performance of models in various tasks. Specifically, this was effected by imposing Bayesian techniques on the three major components of deep networks namely synaptic weights, latent units, and feature functions.

Machine learning, is a vibrant field of study that is constantly evolving and introduces new techniques year by year. For instance, new frameworks are developed like the prominent Generative adversarial networks, memory components like neural attention, and Q-learning with deep networks for reinforcement learning. We believe that there is huge room for significant breakthroughs in the field by means of Bayesian inference. In the following sections, we will briefly present some open research areas that we are actively working on, inspired by our research work described in the previous Chapters.

6.1 Generative Adversarial Networks

Generative Adversarial Networks (GANs) is a new framework of estimating generative models [183].

The process consists in pitting the generative model against an adversary, a discriminative model. In this context, the generative models aim to capture the data distribution and produce valid samples, while the discriminative model to correctly predict the origin of the sample. In essence, if it is from the model distribution or from the data distribution. This framework has shown impressive results in image and text generation [184, 185], disentangled representation learning [186], and semi-supervised learning [187].

Given a dataset $D = \{x^{(i)}\}$ of observations, we want to train a deep network that can capture the underlying distribution $p_{data}(x)$. It is assumed that given a noise input z the generative model will be cable of producing outputs $G(z)$ of the same distribution as the given dataset. While the discriminative model D will be able to distinguish the data origin; thus if the input data x is from the data distribution, $D(x) = 1$ if $x \sim p_{data}$ or the generator. The two models are jointly trained by optimization of the criterion:

$$min_G max_D V(D,G) = E_{x \sim p_{data}}[log D(x)] + E_{z \sim noise}[log(1 - D(G(z)))] \tag{6.1}$$

There are a plethora of GAN variations such as DCGAN [185], Wassertein GANs [188], and DCGAN ensembles [189]. A typical problem in GANs is that the training procedure can lead to mode collapse, where the generator model just learns to generate a few samples that the discriminator cannot distinguish. This characteristic of GANs is not favorable due to the limitation that imposes in capturing the data distribution. Practitioners have derived variations that aim to ameliorate this issue; These are considered analogous to regularizers for maximum likelihood density estimation. In addition, there is an interest from the practitioners to treat this issue by means of Bayesian inference [186, 190]. As was expected, Bayesian inference has provided an increase in the GANs performance. Influenced by this, an improvement we are currently working on, concerns the imposition of our innovative approach of Chapter 4, AsyDGM [54], in the GAN context in order to capture skewed data dynamics.

6.2 Neural Attention

Modeling long and complex dependencies in sequential data is a daunting task in machine learning. Models that try to capture the dependencies by compressing the sequence into fixed-dimensional vector representations have shown success only for cases with short temporal dependencies. To this end, various alternatives have been proposed by practitioners, to extend the model's temporal capacity with one of the most successful one being neural attention [191]. Neural attention, is a recent trend in deep networks and consists of a variable-length memory component, that provides access to its content as needed. In order to be efficiently trainable, the memory is implemented in a differential way. The mechanism has shown impressive results in tasks such as neural machine translation [192, 193], question answering [194, 195], image captioning [196], and document summarization [197].

Neural attention uses a memory bank to read and store information. This is achieved with the assistance of addressing mechanisms that define the degree of importance of each memory state. Typical formulation includes a similarity measure between the model's feature representation and the memory states. For instance, the cosine similarity is frequently used:

$$K[u,v] = \frac{u \cdot v}{||u|| \cdot ||v||} \tag{6.2}$$

In this context, we consider imposing probability distributions at the latent units that produce the feature representations and/or at the memory states. For the latter case, it is interesting to observe how the model will react in different samples from the memory. This is analogous of our intuitions in Chapter 5.

6.3 Deep Q-networks

Reinforcement learning is an alternative type of machine learning that uses a scalar reward to train agents in order to make the correct actions in an environment. In essence, the model (agent) learns an optimal policy by trial and error. In addition, the reward is typically sparse, noisy, and delayed, which results in an increased degree of difficulty for the training algorithm. Furthermore, the environment may be considered a partially observable Markov decision process with the inputs being of a high dimensional nature. In such challenging tasks, deep networks have shown great performance due to their informative feature representation of the high dimensional input. Tasks such as games [198, 199, 200, 201, 202], robotics [203, 204]

and autonomous driving [205], are a few of various examples that deep networks have shown improvements.

A prominent technique for reinforcement learning is Q-learning [206]. The technique learns an action-value function Q which estimates the expected reward of taking a given action when the agent is in a given state and following the optimal policy. To estimate the action-value function, we use the Bellman equation as an iterative update. Thus, we have:

$$Q_{i+1}(s,a) = E[r + \gamma max Q_i(s',a')|s,a] \tag{6.3}$$

where s the state, a the action, r the reward, and γ a discount factor. Using a deep network to learn the action-value function allows for greater performance but requires careful regularization such as experience replay and freezing target networks. We believe that we can increase the performance of these agents by imposing probability distributions at the deep Q-network components. Specifically, Bayesian inference will tackle the uncertainty imposed by the noisy and delayed reward, thus resulting to a more robust action-value function. This is similar in spirit, with the Bayesian inference ideas we introduced in Chapter 3.

Bibliography

[1] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[2] Frank Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[3] David R Cox. "The regression analysis of binary sequences". In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1958), pp. 215–242.

[4] Harry Zhang. "The optimality of naive Bayes". In: *AA* 1.2 (2004), p. 3.

[5] Hiroyuki Takeda, Sina Farsiu, and Peyman Milanfar. "Kernel regression for image processing and reconstruction". In: *IEEE Transactions on image processing* 16.2 (2007), pp. 349–366.

[6] Johan AK Suykens and Joos Vandewalle. "Least squares support vector machine classifiers". In: *Neural processing letters* 9.3 (1999), pp. 293–300.

[7] Terrence S Furey et al. "Support vector machine classification and validation of cancer tissue samples using microarray expression data". In: *Bioinformatics* 16.10 (2000), pp. 906–914.

[8] Simon Tong and Daphne Koller. "Support vector machine active learning with applications to text classification". In: *Journal of machine learning research* 2.Nov (2001), pp. 45–66.

[9] Charalambos Chrysostomou, Harris Partaourides, and Huseyin Seker. "Prediction of Influenza A virus infections in humans using an Artificial Neural Network learning approach". In: *Engineering in Medicine and Biology Society (EMBC), 2017 39th Annual International Conference of the IEEE*. IEEE. 2017, pp. 1186–1189.

[10] Yoshua Bengio, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828.

[11] Yoshua Bengio, Olivier Delalleau, and Nicolas L Roux. "The curse of highly variable functions for local kernel machines". In: *Advances in neural information processing systems*. 2006, pp. 107–114.

[12] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[14] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (1989), pp. 303–314.

[15] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), pp. 106–154.

[16] Daniel J Felleman and David C Van Essen. "Distributed hierarchical processing in the primate cerebral cortex." In: *Cerebral cortex (New York, NY: 1991)* 1.1 (1991), pp. 1–47.

[17] Charles F Cadieu et al. "Deep neural networks rival the representation of primate IT cortex for core visual object recognition". In: *PLoS computational biology* 10.12 (2014), e1003963.

[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[19] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[20] Fei-Fei Li, Rob Fergus, and Pietro Perona. "One-shot learning of object categories". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.4 (2006), pp. 594–611.

[21] John Aldrich et al. "RA Fisher and the making of maximum likelihood 1912-1922". In: *Statistical Science* 12.3 (1997), pp. 162–176.

[22] Michael A Babyak. "What you see may not be what you get: a brief, nontechnical introduction to overfitting in regression-type models". In: *Psychosomatic medicine* 66.3 (2004), pp. 411–421.

[23] Igor V Tetko, David J Livingstone, and Alexander I Luik. “Neural network studies. 1. Comparison of overfitting and overtraining”. In: *Journal of chemical information and computer sciences* 35.5 (1995), pp. 826–833.

[24] Sergios Theodoridis. *Machine learning: a Bayesian and optimization perspective*. Academic Press, 2015.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[26] Clement Farabet et al. “Learning hierarchical features for scene labeling”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1915–1929.

[27] Jonathan J Tompson et al. “Joint training of a convolutional network and a graphical model for human pose estimation”. In: *Advances in neural information processing systems*. 2014, pp. 1799–1807.

[28] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[29] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. “Multi-column deep neural networks for image classification”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 3642–3649.

[30] George E Dahl et al. “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition”. In: *IEEE Transactions on audio, speech, and language processing* 20.1 (2012), pp. 30–42.

[31] Tomáš Mikolov et al. “Strategies for training large scale neural network language models”. In: *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*. IEEE. 2011, pp. 196–201.

[32] Geoffrey Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97.

[33] Tara N Sainath et al. “Deep convolutional neural networks for LVCSR”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8614–8618.

[34] Ronan Collobert et al. “Natural language processing (almost) from scratch”. In: *Journal of Machine Learning Research* 12.Aug (2011), pp. 2493–2537.

[35] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[36] Auke Jan Ijspeert. "Central pattern generators for locomotion control in animals and robots: a review". In: *Neural networks* 21.4 (2008), pp. 642–653.

[37] Jun Tani, Masato Ito, and Yuuya Sugita. "Self-organization of distributedly represented multiple behavior schemata in a mirror system: reviews of robot experiments using RNNPB". In: *Neural Networks* 17.8 (2004), pp. 1273–1289.

[38] Dean A Pomerleau. *Neural network perception for mobile robot guidance*. Vol. 239. Springer Science & Business Media, 2012.

[39] Yoshua Bengio et al. "Greedy layer-wise training of deep networks". In: *Advances in neural information processing systems*. 2007, pp. 153–160.

[40] Christopher Poultney, Sumit Chopra, Yann L Cun, et al. "Efficient learning of sparse representations with an energy-based model". In: *Advances in neural information processing systems*. 2007, pp. 1137–1144.

[41] Pierre Sermanet et al. "Pedestrian detection with unsupervised multi-stage feature learning". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2013, pp. 3626–3633.

[42] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *science* 313.5786 (2006), pp. 504–507.

[43] Martın Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[44] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[45] Ian J Goodfellow et al. "Pylearn2: a machine learning research library". In: *arXiv preprint arXiv:1308.4214* (2013).

[46] Sander Dieleman et al. *Lasagne: First release.* Aug. 2015. DOI: 10.5281/zenodo.27878. URL: http://dx.doi.org/10.5281/zenodo.27878.

[47] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[48] Sébastien Jean et al. "On using very large target vocabulary for neural machine translation". In: *arXiv preprint arXiv:1412.2007* (2014).

[49] Rajat Raina, Anand Madhavan, and Andrew Y Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 873–880.

[50] Abdel-rahman Mohamed, George E Dahl, and Geoffrey Hinton. "Acoustic modeling using deep belief networks". In: *IEEE Transactions on Audio, Speech, and Language Processing* 20.1 (2012), pp. 14–22.

[51] Harris Partaourides and Sotirios P Chatzis. "Deep Network Regularization via Bayesian Inference of Synaptic Connectivity". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2017, pp. 30–41.

[52] Li Wan et al. "Regularization of neural networks using dropconnect". In: *Proceedings of the 30th international conference on machine learning (ICML-13)*. 2013, pp. 1058–1066.

[53] Rajesh Ranganath, Sean Gerrish, and David Blei. "Black box variational inference". In: *Artificial Intelligence and Statistics*. 2014, pp. 814–822.

[54] Harris Partaourides and Sotirios P Chatzis. "Asymmetric deep generative models". In: *Neurocomputing* 241 (2017), pp. 90–96.

[55] Danilo Jimenez Rezende and Shakir Mohamed. "Variational inference with normalizing flows". In: *arXiv preprint arXiv:1505.05770* (2015).

[56] Harris Partaourides and Sotirios P. Chatzis. "Deep Learning with t-Exponential Bayesian Kitchen Sinks". In: *Expert Systems with Applications*. Vol. 98. May 2018.

[57] Ali Rahimi and Benjamin Recht. "Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning". In: *Advances in neural information processing systems*. 2009, pp. 1313–1320.

[58] Hrushikesh Narhar Mhaskar and Charles A Micchelli. "How to choose an activation function". In: *Advances in Neural Information Processing Systems*. 1994, pp. 319–326.

[59] Barry L Kalman and Stan C Kwasny. "Why tanh: choosing a sigmoidal function". In: *Neural Networks, 1992. IJCNN., International Joint Conference on*. Vol. 4. IEEE. 1992, pp. 578–581.

[60] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[61] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural networks* 6.6 (1993), pp. 861–867.

[62] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. ICML*. Vol. 30. 1. 2013.

[63] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[64] Ian J Goodfellow et al. "Maxout networks". In: *arXiv preprint arXiv:1302.4389* (2013).

[65] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[66] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech recognition with deep recurrent neural networks". In: *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE. 2013, pp. 6645–6649.

[67] Tomas Mikolov et al. "Recurrent neural network based language model." In: *Interspeech*. Vol. 2. 2010, p. 3.

[68] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[69] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. "Highway networks". In: *arXiv preprint arXiv:1505.00387* (2015).

[70] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[71] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. "Advances in optimizing recurrent networks". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8624–8628.

[72] Ilya Sutskever. "Training recurrent neural networks". In: *University of Toronto, Toronto, Ont., Canada* (2013).

[73] Yann A LeCun et al. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[74] Christian Darken, Joseph Chang, and John Moody. “Learning rate schedules for faster stochastic gradient search”. In: *Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop*. IEEE. 1992, pp. 3–12.

[75] Léon Bottou. “Large-scale machine learning with stochastic gradient descent”. In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[76] Robert Hecht-Nielsen et al. “Theory of the backpropagation neural network.” In: *Neural Networks* 1.Supplement-1 (1988), pp. 445–448.

[77] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 2010, pp. 249–256.

[78] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[79] Ning Qian. “On the momentum term in gradient descent learning algorithms”. In: *Neural networks* 12.1 (1999), pp. 145–151.

[80] Yurii Nesterov. “A method of solving a convex programming problem with convergence rate O (1/k2)”. In: *Soviet Mathematics Doklady*. Vol. 27. 2. 1983, pp. 372–376.

[81] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.

[82] Matthew D Zeiler. “ADADELTA: an adaptive learning rate method”. In: *arXiv preprint arXiv:1212.5701* (2012).

[83] Yann Dauphin, Harm de Vries, and Yoshua Bengio. “Equilibrated adaptive learning rates for non-convex optimization”. In: *Advances in neural information processing systems*. 2015, pp. 1504–1512.

[84] Diederik Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[85] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: *International conference on machine learning*. 2013, pp. 1139–1147.

[86] Arvind Neelakantan et al. “Adding gradient noise improves learning for very deep networks”. In: *arXiv preprint arXiv:1511.06807* (2015).

[87] Prabir Burman. “A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods”. In: *Biometrika* 76.3 (1989), pp. 503–514.

[88] Andrew Y Ng and Michael I Jordan. “On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes”. In: *Advances in neural information processing systems*. 2002, pp. 841–848.

[89] Y Dan Rubinstein, Trevor Hastie, et al. “Discriminative vs Informative Learning.” In: *KDD*. Vol. 5. 1997, pp. 49–53.

[90] Bradley Efron. “The efficiency of logistic regression compared to normal discriminant analysis”. In: *Journal of the American Statistical Association* 70.352 (1975), pp. 892–898.

[91] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised representation learning with deep convolutional generative adversarial networks”. In: *arXiv preprint arXiv:1511.06434* (2015).

[92] Cristian Sminchisescu, Atul Kanaujia, and Dimitris Metaxas. “Conditional models for contextual human motion recognition”. In: *Computer Vision and Image Understanding* 104.2 (2006), pp. 210–220.

[93] Graham W Taylor, Geoffrey E Hinton, and Sam T Roweis. “Modeling human motion using binary latent variables”. In: *Advances in neural information processing systems*. 2007, pp. 1345–1352.

[94] Diederik P Kingma et al. “Semi-supervised learning with deep generative models”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 3581–3589.

[95] Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. “Semi-supervised learning using gaussian fields and harmonic functions”. In: *Proceedings of the 20th International conference on Machine learning (ICML-03)*. 2003, pp. 912–919.

[96] Xiaojin Zhu. “Semi-supervised learning literature survey”. In: *Computer Science, University of Wisconsin-Madison* 2.3 (2006), p. 4.

[97] Helmut Grabner, Christian Leistner, and Horst Bischof. “Semi-supervised on-line boosting for robust tracking”. In: *Computer Vision–ECCV 2008* (2008), pp. 234–247.

[98] Diederik P Kingma et al. "Semi-supervised learning with deep generative models". In: *Advances in Neural Information Processing Systems*. 2014, pp. 3581–3589.

[99] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[100] Paul Smolensky. *Information processing in dynamical systems: Foundations of harmony theory*. Tech. rep. 1986.

[101] Ruslan Salakhutdinov and Geoffrey Hinton. "Deep boltzmann machines". In: *Artificial Intelligence and Statistics*. 2009, pp. 448–455.

[102] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[103] David Duvenaud, Dougal Maclaurin, and Ryan Adams. "Early stopping as nonparametric variational inference". In: *Artificial Intelligence and Statistics*. 2016, pp. 1070–1077.

[104] Mário Figueiredo. "Adaptive sparseness using Jeffreys prior". In: *Advances in neural information processing systems*. 2002, pp. 697–704.

[105] Andrew Gelman et al. "A weakly informative default prior distribution for logistic and other regression models". In: *The Annals of Applied Statistics* (2008), pp. 1360–1383.

[106] Mário AT Figueiredo. "Adaptive sparseness for supervised learning". In: *IEEE transactions on pattern analysis and machine intelligence* 25.9 (2003), pp. 1150–1159.

[107] Ata Kabán. "On Bayesian classification with Laplace priors". In: *Pattern Recognition Letters* 28.10 (2007), pp. 1271–1282.

[108] Jason Rennie. "On l2-norm regularization and the gaussian prior". In: (2003).

[109] Yarin Gal and Zoubin Ghahramani. "Dropout as a Bayesian approximation: Insights and applications". In: *Deep Learning Workshop, ICML*. 2015.

[110] Amr Ahmed and Eric P Xing. "Seeking the truly correlated topic posterior-on tight approximate inference of logistic-normal admixture model". In: *Artificial Intelligence and Statistics*. 2007, pp. 19–26.

[111] Mohammad E Khan et al. "Variational bounds for mixed-data factor analysis". In: *Advances in Neural Information Processing Systems*. 2010, pp. 1108–1116.

[112] Antti Honkela and Harri Valpola. "Unsupervised variational Bayesian learning of nonlinear models". In: *Advances in neural information processing systems*. 2005, pp. 593–600.

[113] TS Jaakkola and MI Jordan. "Bayesian logistic regression: a variational approach". In: *Proceedings of the 1997 Conference on Artificial Intelligence and Statistics, Ft. Lauderdale, FL*. 1997.

[114] Michael Braun and Jon McAuliffe. "Variational inference for large-scale models of discrete choice". In: *Journal of the American Statistical Association* 105.489 (2010), pp. 324–335.

[115] Joshua Clinton, Simon Jackman, and Douglas Rivers. "The statistical analysis of roll call data". In: *American Political Science Review* 98.2 (2004), pp. 355–370.

[116] David M Blei and John D Lafferty. "Dynamic topic models". In: *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 113–120.

[117] Chong Wang and David M Blei. "Variational inference in nonconjugate models". In: *Journal of Machine Learning Research* 14.Apr (2013), pp. 1005–1031.

[118] Alan E Gelfand and Adrian FM Smith. "Sampling-based approaches to calculating marginal densities". In: *Journal of the American statistical association* 85.410 (1990), pp. 398–409.

[119] Stuart Geman and Donald Geman. "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images". In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1984), pp. 721–741.

[120] Walter R Gilks, Sylvia Richardson, and David Spiegelhalter. *Markov chain Monte Carlo in practice*. CRC press, 1995.

[121] Nicholas Metropolis et al. "Equation of state calculations by fast computing machines". In: *The journal of chemical physics* 21.6 (1953), pp. 1087–1092.

[122] W Keith Hastings. "Monte Carlo sampling methods using Markov chains and their applications". In: *Biometrika* 57.1 (1970), pp. 97–109.

[123] Pierre Baldi and Peter J Sadowski. "Understanding dropout". In: *Advances in Neural Information Processing Systems*. 2013, pp. 2814–2822.

[124] Tommi S Jaakkola and Michael I Jordan. "Bayesian parameter estimation via variational methods". In: *Statistics and Computing* 10.1 (2000), pp. 25–37.

[125] Peter W Glynn. "Likelihood ratio gradient estimation for stochastic systems". In: *Communications of the ACM* 33.10 (1990), pp. 75–84.

[126] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[127] Yoshua Bengio et al. “Generalized denoising auto-encoders as generative models”. In: *Advances in Neural Information Processing Systems*. 2013, pp. 899–907.

[128] Frédéric Bastien et al. “Theano: new features and speed improvements”. In: *arXiv preprint arXiv:1211.5590* (2012).

[129] Diederik P Kingma and Max Welling. “Auto-encoding variational bayes”. In: *arXiv preprint arXiv:1312.6114* (2013).

[130] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. “Stochastic back-propagation and approximate inference in deep generative models”. In: *arXiv preprint arXiv:1401.4082* (2014).

[131] Richard E Turner and Maneesh Sahani. “Two problems with variational expectation maximisation for time-series models”. In: *Bayesian Time series models* (2011), pp. 115–138.

[132] Lars Maaløe et al. “Auxiliary Deep Generative Models”. In: *33rd International Conference on Machine Learning (ICML 2016)*. 2016.

[133] Adelchi Azzalini and A Dalla Valle. “The multivariate skew-normal distribution”. In: *Biometrika* 83.4 (1996), pp. 715–726.

[134] Sujit K Sahu, Dipak K Dey, and Márcia D Branco. “A new class of multivariate skew distributions with applications to Bayesian regression models”. In: *Canadian Journal of Statistics* 31.2 (2003), pp. 129–150.

[135] Tsung I Lin. “Maximum likelihood estimation for multivariate skew normal mixture models”. In: *Journal of Multivariate Analysis* 100.2 (2009), pp. 257–265.

[136] Tsung I Lin, Jack C Lee, and Shu Y Yen. “Finite mixture modelling using the skew normal distribution”. In: *Statistica Sinica* (2007), pp. 909–927.

[137] Saumyadipta Pyne et al. “Automated high-dimensional flow cytometric data analysis”. In: *Proceedings of the National Academy of Sciences* 106.21 (2009), pp. 8519–8524.

[138] Sharon X Lee and Geoffrey J Mclachlan. “On mixtures of skew normal and skew t-distributions”. In: *Advances in Data Analysis and Classification* 7.3 (2013), pp. 241–266.

[139] Sharon Lee and Geoffrey J McLachlan. “Finite mixtures of multivariate skew t-distributions: some recent and new results”. In: *Statistics and Computing* 24.2 (2014), pp. 181–202.

[140] Brian C Franczak et al. “Parsimonious shifted asymmetric Laplace mixtures”. In: *arXiv preprint arXiv:1311.0317* (2013).

[141] Cristina Tortora, Paul D McNicholas, and Ryan P Browne. “A mixture of generalized hyperbolic factor analyzers”. In: *Advances in Data Analysis and Classification* 10.4 (2016), pp. 423–440.

[142] Angela Montanari and Cinzia Viroli. “A skew-normal factor model for the analysis of student satisfaction towards university courses”. In: *Journal of Applied Statistics* 37.3 (2010), pp. 473–487.

[143] Tsung-I Lin, Geoffrey J McLachlan, and Sharon X Lee. “Extending mixtures of factor models using the restricted multivariate skew-normal distribution”. In: *Journal of Multivariate Analysis* 143 (2016), pp. 398–413.

[144] John Paisley, David Blei, and Michael Jordan. “Variational Bayesian inference with stochastic search”. In: *arXiv preprint arXiv:1206.6430* (2012).

[145] Felix V Agakov and David Barber. “An auxiliary variational method”. In: *International Conference on Neural Information Processing*. Springer. 2004, pp. 561–566.

[146] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. “What is the best multi-stage architecture for object recognition?” In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE. 2009, pp. 2146–2153.

[147] Takeru Miyato et al. “Distributional smoothing with virtual adversarial training”. In: *arXiv preprint arXiv:1507.00677* (2015).

[148] Athanasios Voulodimos et al. “A threefold dataset for activity and workflow recognition in complex industrial environments”. In: *IEEE MultiMedia* 19.3 (2012), pp. 42–52.

[149] Dimitrios Kosmopoulos and Sotirios P Chatzis. “Robust visual behavior recognition”. In: *IEEE Signal Processing Magazine* 27.5 (2010), pp. 34–45.

[150] Sang Min Oh et al. “Learning and inferring motion patterns using parametric segmental switching linear dynamic systems”. In: *International Journal of Computer Vision* 77.1 (2008), pp. 103–124.

[151] Thierry Bertin-Mahieux et al. “The Million Song Dataset.” In: *Ismir*. Vol. 2. 9. 2011, p. 10.

[152] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[153] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[154] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[155] Radford M Neal. *Bayesian learning for neural networks*. Vol. 118. Springer Science & Business Media, 2012.

[156] Andreas Damianou and Neil Lawrence. "Deep gaussian processes". In: *Artificial Intelligence and Statistics*. 2013, pp. 207–215.

[157] Thang Bui et al. "Deep gaussian processes for regression using approximate expectation propagation". In: *International Conference on Machine Learning*. 2016, pp. 1472–1481.

[158] Thang D Bui et al. "Training deep Gaussian processes using stochastic expectation propagation and probabilistic backpropagation". In: *arXiv preprint arXiv:1511.03405* (2015).

[159] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical learning theory*. Vol. 1. Wiley New York, 1998.

[160] Robert E Schapire. "The boosting approach to machine learning: An overview". In: *Nonlinear estimation and classification*. Springer, 2003, pp. 149–171.

[161] Ali Rahimi and Benjamin Recht. "Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning". In: *Advances in neural information processing systems*. 2009, pp. 1313–1320.

[162] Quoc Le, Tamás Sarlós, and Alex Smola. "Fastfood-approximating kernel expansions in loglinear time". In: *Proceedings of the international conference on machine learning*. Vol. 85. 2013.

[163] Zichao Yang et al. "A la carte–learning fast kernels". In: *Artificial Intelligence and Statistics*. 2015, pp. 1098–1106.

[164] Martin J Wainwright, Michael I Jordan, et al. "Graphical models, exponential families, and variational inference". In: *Foundations and Trends® in Machine Learning* 1.1–2 (2008), pp. 1–305.

[165] Constantino Tsallis. "Possible generalization of Boltzmann-Gibbs statistics". In: *Journal of statistical physics* 52.1 (1988), pp. 479–487.

[166] AndréM C de Souza and Constantino Tsallis. “Student’s t-and r-distributions: Unified derivation from an entropic variational principle”. In: *Physica A: Statistical Mechanics and its Applications* 236.1-2 (1997), pp. 52–57.

[167] Constantino Tsallis, RenioS Mendes, and Anel R Plastino. “The role of constraints within generalized nonextensive statistics”. In: *Physica A: Statistical Mechanics and its Applications* 261.3 (1998), pp. 534–554.

[168] Jan Naudts. “Deformed exponentials and logarithms in generalized thermostatistics”. In: *Physica A: Statistical Mechanics and its Applications* 316.1 (2002), pp. 323–334.

[169] Jan Naudts. “Estimators, escort probabilities, and phi-exponential families in statistical physics”. In: *J. Ineq. Pure Appl. Math* 5.4 (2004), p. 102.

[170] Jan Naudts. “Generalized thermostatistics based on deformed exponential and logarithmic functions”. In: *Physica A: Statistical Mechanics and its Applications* 340.1 (2004), pp. 32–40.

[171] Geoffrey McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2004.

[172] Nan Ding, Yuan Qi, and Svn Vishwanathan. “t-divergence based approximate inference”. In: *Advances in Neural Information Processing Systems*. 2011, pp. 1494–1502.

[173] Chuanhai Liu and Donald B Rubin. “ML estimation of the t distribution using EM and its extensions, ECM and ECME”. In: *Statistica Sinica* (1995), pp. 19–39.

[174] Sotirios P Chatzis and Dimitrios I Kosmopoulos. “A variational Bayesian methodology for hidden Markov models utilizing Student’s-t mixtures”. In: *Pattern Recognition* 44.2 (2011), pp. 295–306.

[175] Sotirios P Chatzis, Dimitrios I Kosmopoulos, and Theodora A Varvarigou. “Robust sequential data modeling using an outlier tolerant hidden Markov model”. In: *IEEE transactions on pattern analysis and machine intelligence* 31.9 (2009), pp. 1657–1669.

[176] Markus Svensén and Christopher M Bishop. “Robust Bayesian mixture modelling”. In: *Neurocomputing* 64 (2005), pp. 235–252.

[177] Nan Ding, Yuan Qi, and Svn Vishwanathan. “t-divergence based approximate inference”. In: *Advances in Neural Information Processing Systems*. 2011, pp. 1494–1502.

[178] Arthur Asuncion and David Newman. *UCI machine learning repository*. 2007.

[179] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. “Training invariant support vector machines using selective sampling”. In: *Large scale kernel machines* (2007), pp. 301–320.

[180] Kurt Cutajar et al. “Random Feature Expansions for Deep Gaussian Processes”. In: *International Conference on Machine Learning*. 2017, pp. 884–893.

[181] Walter Rudin. *Fourier analysis on groups*. Courier Dover Publications, 2017.

[182] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[183] Ian Goodfellow et al. “Generative adversarial nets”. In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[184] Xiaodan Liang et al. “Recurrent Topic-Transition GAN for Visual Paragraph Generation”. In: *arXiv preprint arXiv:1703.07022* (2017).

[185] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised representation learning with deep convolutional generative adversarial networks”. In: *arXiv preprint arXiv:1511.06434* (2015).

[186] Xi Chen et al. “Infogan: Interpretable representation learning by information maximizing generative adversarial nets”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 2172–2180.

[187] Tim Salimans et al. “Improved techniques for training gans”. In: *Advances in Neural Information Processing Systems*. 2016, pp. 2234–2242.

[188] Ishaan Gulrajani et al. “Improved training of wasserstein gans”. In: *arXiv preprint arXiv:1704.00028* (2017).

[189] Yaxing Wang, Lichao Zhang, and Joost van de Weijer. “Ensembles of generative adversarial networks”. In: *arXiv preprint arXiv:1612.00991* (2016).

[190] Yunus Saatchi and Andrew Gordon Wilson. “Bayesian GAN”. In: *arXiv preprint arXiv:1705.09558* (2017).

[191] Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural turing machines”. In: *arXiv preprint arXiv:1410.5401* (2014).

[192] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *arXiv preprint arXiv:1409.0473* (2014).

[193] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. "Effective approaches to attention-based neural machine translation". In: *arXiv preprint arXiv:1508.04025* (2015).

[194] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. "End-to-end memory networks". In: *Advances in neural information processing systems*. 2015, pp. 2440–2448.

[195] Jason Weston, Sumit Chopra, and Antoine Bordes. "Memory networks". In: *arXiv preprint arXiv:1410.3916* (2014).

[196] Kelvin Xu et al. "Show, attend and tell: Neural image caption generation with visual attention". In: *International Conference on Machine Learning*. 2015, pp. 2048–2057.

[197] Alexander M Rush, Sumit Chopra, and Jason Weston. "A neural attention model for abstractive sentence summarization". In: *arXiv preprint arXiv:1509.00685* (2015).

[198] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[199] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[200] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[201] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." In: *AAAI*. 2016, pp. 2094–2100.

[202] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. 2016, pp. 1928–1937.

[203] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[204] Shixiang Gu et al. "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *Robotics and Automation (ICRA), 2017 IEEE International Conference on*. IEEE. 2017, pp. 3389–3396.

[205] Ahmad EL Sallab et al. "Deep reinforcement learning framework for autonomous driving". In: *Electronic Imaging* 2017.19 (2017), pp. 70–76.

[206] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

APPENDIX I

Here we will present parts of the code used for the experiments. For Chapter 3, the code was adapted from Lasagne's examples. The original code can be found at https://github.com/Lasagne/Lasagne. Following are examples of the networks used:

```
def Plain(input_var=None):
    l_hid = list()
    network = lasagne.layers.InputLayer(shape=(None, 3, 32, 32),
                                        input_var=input_var)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=64, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
```

```
                                                stride=2)
    network = lasagne.layers.DenseLayer(
            network,
            num_units=512,
            nonlinearity=lasagne.nonlinearities.rectify,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.DenseLayer(
            network,
            num_units=100,
            nonlinearity=lasagne.nonlinearities.softmax)
    l_hid.append(network)
    return l_hid, network

def Dropout(input_var=None):
    l_hid = list()
    network = lasagne.layers.InputLayer(shape=(None, 3, 32, 32),
                                        input_var=input_var)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=64, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
```

```
    network = lasagne.layers.DenseLayer(
            lasagne.layers.dropout(network, p=.5),
            num_units=512,
            nonlinearity=lasagne.nonlinearities.rectify,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.DenseLayer(
            network,
            num_units=100,
            nonlinearity=lasagne.nonlinearities.softmax)
    l_hid.append(network)
    return l_hid, network

def DropConnect(input_var=None):
    l_hid = list()
    network = lasagne.layers.InputLayer(shape=(None, 3, 32, 32),
                                        input_var=input_var)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=64, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                            stride=2)
    network = lasagne.layers.WeightDecayLayer(
```

```
            network,
            num_units=512,
            nonlinearity=lasagne.nonlinearities.rectify,
            Wmean=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.DenseLayer(
            network,
            num_units=100,
            nonlinearity=lasagne.nonlinearities.softmax)
    l_hid.append(network)
    return l_hid, network

def DropConnectPlus(input_var=None):
    l_hid = list()
    input_data = lasagne.layers.InputLayer(shape=(None, 3, 32, 32),
                                    input_var=input_var)
    network = lasagne.layers.Conv2DLayer(
            input_data, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                        stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=32, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                        stride=2)
    network = lasagne.layers.Conv2DLayer(
            network, num_filters=64, filter_size=(5, 5),
            nonlinearity=lasagne.nonlinearities.rectify,
            pad=2,stride=1,
            W=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.MaxPool2DLayer(network, pool_size=(3, 3),
                                        stride=2)
    network = lasagne.layers.DenseIBPLayer(
            network,
```

```
            num_units=512,
            nonlinearity=lasagne.nonlinearities.rectify,
            Wmean=lasagne.init.GlorotUniform())
    l_hid.append(network)
    network = lasagne.layers.DenseLayer(
            network,
            num_units=100,
            nonlinearity=lasagne.nonlinearities.softmax)
    l_hid.append(network)
    return l_hid, network
```

Next, we present the implemented layers that do not exist in the Lasagne framework:

```
class DenseIBPLayer(Layer):
    def __init__(self, incoming, num_units, Wmean=init.GlorotUniform(),
                                            Wmask=init.GlorotUniform(),
                 b=init.Constant(0.), nonlinearity=nonlinearities.rectify,
                 **kwargs):
        super(DenseIBPLayer, self).__init__(incoming, **kwargs)
        self.nonlinearity = (nonlinearities.identity if nonlinearity is None
                             else nonlinearity)
        self.num_units = num_units
        self._srng = RandomStreams(get_rng().randint(1, 2147462579))
        num_inputs = int(np.prod(self.input_shape[1:]))
        self.Wmean = self.add_param(Wmean, (num_inputs, num_units),
                                    name="Wmean")
        self.Wmask = self.add_param(Wmask, (num_inputs, num_units),
                                    name="Wmask")
        self.Wmask = T.nnet.sigmoid(self.Wmask)
        self.betatilde = self.add_param(init.Constant(1.),
                                        (num_inputs,num_units),
                                        name="betatilde")
        self.betahat = self.add_param(init.Constant(1.),
                                      (num_inputs,num_units),
                                      name="betahat")
        self.rsn = RandomStreams(seed=234)
        if b is None:
            self.b = None
        else:
            self.b = self.add_param(b, (num_units,), name="b",
                                    regularizable=False)
```

```
    def get_output_shape_for(self, input_shape):
        return (input_shape[0], self.num_units)

    def get_output_for(self, input, deterministic=False,**kwargs):
        if input.ndim > 2:
            # if the input has more than two dimensions, flatten it
            # into a batch of feature vectors.
            input = input.flatten(2)
        num_inputs = int(np.prod(self.input_shape[1:]))
        if deterministic:
                        num_inputs = int(np.prod(self.input_shape[1:]))
            activation = T.dot(input, self.Wmean*self.Wmask)
            if self.b is not None:
                activation = activation + self.b.dimshuffle('x', 0)
            return self.nonlinearity(activation)
        else:
            num_inputs = int(np.prod(self.input_shape[1:]))
            self.W = self.Wmean *
                    (self.Wmask > self.rsn.uniform(size=(num_inputs,
                                                        int(self.num_units))))
            activation = T.dot(input, self.W)
            if self.b is not None:
                activation = activation + self.b.dimshuffle('x', 0)
            return self.nonlinearity(activation)

class WeightDecayLayer(Layer):
    def __init__(self, incoming, num_units, Wmean=init.GlorotUniform(),
                 Decay=0.5, b=init.Constant(0.),
                 nonlinearity=nonlinearities.rectify, **kwargs):
        super(WeightDecayLayer, self).__init__(incoming, **kwargs)
        self.nonlinearity = (nonlinearities.identity if nonlinearity is None
                             else nonlinearity)
        self.num_units = num_units
        self._srng = RandomStreams(get_rng().randint(1, 2147462579))
        num_inputs = int(np.prod(self.input_shape[1:]))
        self.Wmean = self.add_param(Wmean, (num_inputs, num_units),
                                            name="Wmean")
        self.Decay = Decay
        self._srng = RandomStreams(get_rng().randint(1, 2147462579))
        self.rsn = RandomStreams(seed=234)
        if b is None:
```

```
            self.b = None
        else:
            self.b = self.add_param(b, (num_units,), name="b",
                                    regularizable=False)

    def get_output_shape_for(self, input_shape):
        return (input_shape[0], self.num_units)

    def get_output_for(self, input, deterministic=False,**kwargs):
        if input.ndim > 2:
            # if the input has more than two dimensions, flatten it
            # into a batch of feature vectors.
            input = input.flatten(2)
        num_inputs = int(np.prod(self.input_shape[1:]))
        if deterministic:
            one = T.constant(1)
            retain_prob = self.Decay
            num_inputs = int(np.prod(self.input_shape[1:]))
            activation = T.dot(input, self.Wmean*retain_prob)
            if self.b is not None:
                activation = activation + self.b.dimshuffle('x', 0)
            return self.nonlinearity( activation )

        else:
            one = T.constant(1)
            retain_prob = self.Decay
            self.W = self.Wmean *
                    (self._srng.binomial(size=(num_inputs,
                                                int(self.num_units)),
                                                p=retain_prob,
                                                dtype=input.dtype))
            activation = T.dot(input, self.W)
            if self.b is not None:
                activation = activation + self.b.dimshuffle('x', 0)
            return self.nonlinearity(activation)
```

Finally, the loss functions for the training procedure:

```
# loss function for Plain, Dropout and DropConnect networks
prediction = lasagne.layers.get_output(network)
loss = lasagne.objectives.categorical_crossentropy(prediction,
                                                    target_var)
```

```
loss = loss.mean()

# loss function for DropConnect++ network
prediction = lasagne.layers.get_output(network)
loss = lasagne.objectives.categorical_crossentropy(prediction,
                                                     target_var)
loss = loss.mean()
psi1 = T.psi(l_hid[3].betatilde) - T.psi(l_hid[3].betatilde + \
                                         l_hid[3].betahat)
psi2 = T.psi(l_hid[3].betahat) - T.psi(l_hid[3].betatilde + \
                                       l_hid[3].betahat)
gamma = T.gammaln(l_hid[3].betatilde+l_hid[3].betahat) - \
        T.gammaln(l_hid[3].betatilde) - T.gammaln(l_hid[3].betahat)
loss2 = loss + T.sum( (l_hid[3].betatilde-1)*psi1 \
             + (l_hid[3].betahat-1)*psi2 ) + T.sum(gamma) \
             + T.sum(l_hid[3].Wmask*T.log(l_hid[3].Wmask) \
             + (1-l_hid[3].Wmask*T.log(1-l_hid[3].Wmask))) \
             - T.sum( l_hid[3].Wmask*psi1 \
             + (1 - l_hid[3].Wmask)*psi2) \
             - T.sum(T.log(l_hid[3].Wmask))
```

For Chapter 5, the code was adapted from [180] paper's source code. The original code can be found at https://github.com/mauriziofilippone/deep_gp_random_features. First, we created a bash shell script to automate the process.

```
for dataset in Concrete_Data Boston_Housing winequality-white
               yacht_hydrodynamics kin8nm Naval energy CCPP
               YearPrediction
do
for model_c in 'DtBKS' 'RKS' 'DGP_rbf' 'DGP_arcosine1'
do
for nl in 2 3 4
do
for n_rff in 100
do
for df in 5
do
for seed in 1872583848 794921487 111352301 2360782358 1869695442
            2081981515 1805465960 1376693511 1418777250 663257521
            878959199 3001592395 2659748565 515183663 1287007039
```

```
            2083814687 1146014426 2717587860 2667749500 3514257012
do
python dgp_rff_regression.py --seed=$seed --model_c=$model_c
                             --dataset=$dataset --nl=$nl
                             --n_rff=$n_rff --df=$df
                             #--learn_Omega=no
                             #--kernel_type=arccosine
                             --kernel_arccosine_degree=1
                             #--normalize_=FALSE
done
done
done
done
done
done
```

Next, we present the dataset import and the main program:

```
def import_dataset(dataset, fold, normalize=True):
    file = 'datasets/' + dataset + '.csv'
    dataset_ = np.loadtxt(file,delimiter=',')
    if dataset == 'YearPrediction':
        dataset_ = (dataset_ - dataset_.mean(axis=0)) \
                    / dataset_.std(axis=0)
        train_X = dataset_[:463715,1:]
        train_Y = dataset_[:463715,0]
        train_Y = np.reshape(train_Y, (-1, 1))
        test_X = dataset_[-51630:,1:]
        test_Y = dataset_[-51630:,0]
        test_Y = np.reshape(test_Y, (-1, 1))
    else:
        _num_examples = dataset_.shape[0]
        perm = np.arange(_num_examples)
        np.random.shuffle(perm)
        if dataset == 'Naval' or dataset == 'energy':
            n_outputs = 2
        else:
            n_outputs = 1
        if n_outputs > 1 :
            _X = dataset_[perm,:-n_outputs]
            _Y = dataset_[perm,-n_outputs:]
```

```
        else:
            _X = dataset_[perm,:-1]
            _Y = dataset_[perm,-1]
        if dataset != 'protein':
            _X = (_X - _X.mean(axis=0)) / _X.std(axis=0)
            if normalize:
                _Y = (_Y - _Y.mean(axis=0)) / _Y.std(axis=0)
        test_size = int(_num_examples*0.1)
        train_X = _X[:-test_size,:]
        if n_outputs > 1 :
            train_Y = _Y[:-test_size,:]
        else:
            train_Y = _Y[:-test_size]
            train_Y = np.reshape(train_Y, (-1, 1))
        test_X = _X[-test_size:,:]
        if n_outputs > 1 :
            test_Y = _Y[-test_size:,:]
        else:
            test_Y = _Y[-test_size:]
            test_Y = np.reshape(test_Y, (-1, 1))
    data = DataSet(train_X, train_Y)
    test = DataSet(test_X, test_Y)

    return data, test

if __name__ == '__main__':
    FLAGS = utils.get_flags()
    ## Set random seed for tensorflow and numpy operations
    tf.set_random_seed(FLAGS.seed)
    np.random.seed(FLAGS.seed)
    data, test = import_dataset(FLAGS.dataset,
                                FLAGS.fold,
                                FLAGS.normalize_)
    ## Here we define a custom loss for dgp to show
    error_rate = losses.RootMeanSqError(data.Dout)
    ## Likelihood
    like = likelihoods.Gaussian()
    ## Optimizer
    optimizer = utils.get_optimizer(FLAGS.optimizer,
                                    FLAGS.learning_rate)
    mc_samples = FLAGS.n_rff
    ## Main dgp object
```

```python
    if FLAGS.model_c == 'DGP_rbf':
        dgp = DgpRff(like, data.num_examples, data.X.shape[1],
                     data.Y.shape[1], FLAGS.nl, FLAGS.n_rff,
                     FLAGS.df, 'RBF', FLAGS.kernel_arccosine_degree,
                     FLAGS.is_ard, FLAGS.feed_forward, 1000, 4000,
                     FLAGS.learn_Omega)
    if FLAGS.model_c == 'DGP_arcosine1':
        dgp = DgpRff(like, data.num_examples, data.X.shape[1],
                     data.Y.shape[1], FLAGS.nl, FLAGS.n_rff, FLAGS.df,
                     'arccosine', 1, FLAGS.is_ard, FLAGS.feed_forward,
                     1000, 4000, FLAGS.learn_Omega)
    if FLAGS.model_c == 'DtBKS':
        dgp = DtBKS(like, data.num_examples, data.X.shape[1],
                      data.Y.shape[1], FLAGS.nl, FLAGS.n_rff,
                      FLAGS.df, FLAGS.kernel_type,
                      FLAGS.kernel_arccosine_degree,
                      FLAGS.is_ard, FLAGS.feed_forward, 0,
                      100000, FLAGS.learn_Omega)
    if FLAGS.model_c == 'RKS':
        dgp = DtBKS(like, data.num_examples, data.X.shape[1],
                      data.Y.shape[1], FLAGS.nl, FLAGS.n_rff,
                      FLAGS.df, FLAGS.kernel_type,
                      FLAGS.kernel_arccosine_degree, FLAGS.is_ard,
                      FLAGS.feed_forward, 0, 100000, 'no')
    ## Learning
    minimum, iter_ = dgp.learn(data, FLAGS.learning_rate, mc_samples,
                               FLAGS.batch_size, FLAGS.n_iterations,
                               optimizer, FLAGS.display_step, test,
                               mc_samples, error_rate, FLAGS.duration,
                               FLAGS.less_prints)
    name = 'results_' + FLAGS.model_c + '_' + FLAGS.dataset + '_nl=' \
                 + str(FLAGS.nl) + '_n_rff=' + str(FLAGS.n_rff) \
                 + '_df=' + str(FLAGS.df) + '.txt'
    f = open(name, 'a')
    writer = csv.writer(f, delimiter='\t')
    writer.writerow( [FLAGS.seed, minimum, iter_] )
    f.close()
```

Finally, the changes implemented at DgpRff model in order to use our DtBKS model.

```python
class DtBKS(object):
```

```
def __init__(self, likelihood_fun, num_examples, d_in, d_out,
             n_layers, n_rff, df, kernel_type,
             kernel_arccosine_degree, is_ard, feed_forward,
             q_Omega_fixed, theta_fixed, learn_Omega):
    self.likelihood = likelihood_fun
    self.kernel_type = kernel_type
    self.is_ard = is_ard
    self.feed_forward = feed_forward
    self.q_Omega_fixed = q_Omega_fixed
    self.theta_fixed = theta_fixed
    self.q_Omega_fixed_flag = q_Omega_fixed > 0
    self.theta_fixed_flag = theta_fixed > 0
    self.learn_Omega = learn_Omega
    self.arccosine_degree = kernel_arccosine_degree
    ## These are all scalars
    self.num_examples = num_examples
    self.nl = n_layers ## Number of hidden layers
    self.n_Omega = n_layers
    ## Number of weigh matrices is "Number of hidden layers"
    self.n_W = n_layers
    ## These are arrays to allow flexibility in the future
    self.n_rff = n_rff * np.ones(n_layers, dtype = np.int64)
    self.df = df * np.ones(n_layers, dtype=np.int64)
    ## Dimensionality of Omega matrices
    if self.feed_forward:
        self.d_in = np.concatenate([[d_in],
                                    self.df[:(n_layers - 1)] + d_in])
    else:
        self.d_in = np.concatenate([[d_in],
                                            self.df[:(n_layers - 1)]])
    self.d_out = self.n_rff
    ## Dimensionality of W matrices
    if self.kernel_type == "RBF":
        self.dhat_in = self.n_rff * 2
        self.dhat_out = np.concatenate([self.df[:-1], [d_out]])
    if self.kernel_type == "arccosine":
        self.dhat_in = self.n_rff
        self.dhat_out = np.concatenate([self.df[:-1], [d_out]])
    ## When Omega is learned variationally, define the right
    ## KL function and the way Omega are constructed
    if self.learn_Omega == "var":
        self.get_kl = self.get_kl_Omega_to_learn
```

```
        self.sample_from_Omega = self.sample_from_Omega_to_learn
    ## When Omega is optimized, fix some standard normals
    ## throughout the execution that will be used to construct Omega
    if self.learn_Omega == "optim":
        self.get_kl = self.get_kl_Omega_to_learn
        self.sample_from_Omega = self.sample_from_Omega_optim
        self.z_for_Omega_fixed = []
        for i in range(self.n_Omega):
            tmp = utils.get_normal_samples(1, self.d_in[i],
                                               self.d_out[i])
            self.z_for_Omega_fixed.append(tf.Variable(tmp[0,:,:],
                                               trainable = False))
    ## When Omega is fixed, fix some standard normals
    ## throughout the execution that will be used to construct Omega
    if self.learn_Omega == "no":
        self.get_kl = self.get_kl_Omega_fixed
        self.sample_from_Omega = self.sample_from_Omega_fixed
        self.z_for_Omega_fixed = []
        for i in range(self.n_Omega):
            tmp = utils.get_normal_samples(1, self.d_in[i],
                                             self.d_out[i])
            self.z_for_Omega_fixed.append(tf.Variable(tmp[0,:,:],
                                               trainable = False))
    ## Parameters defining prior over Omega
    self.log_theta_sigma2 = tf.Variable(tf.zeros([n_layers]),
                                     name="log_theta_sigma2")
    if self.is_ard:
        self.llscale0 = []
        for i in range(self.nl):
            self.llscale0.append(tf.constant(0.5 \
                            * np.log(self.d_in[i]), tf.float32))
    else:
        self.llscale0 = tf.constant(0.5 * np.log(self.d_in),
                                     tf.float32)
    if self.is_ard:
        self.log_theta_lengthscale = []
        for i in range(self.nl):
            self.log_theta_lengthscale.append(tf.Variable(\
            tf.mul(tf.ones([self.d_in[i]]), self.llscale0[i]),
            name="log_theta_lengthscale"))
    else:
        self.log_theta_lengthscale = tf.Variable(self.llscale0,
```

```
                                        name="log_theta_lengthscale")
    self.prior_mean_Omega, self.log_prior_var_Omega,
    self.log_prior_nu_Omega = self.get_prior_Omega(\
                                    self.log_theta_lengthscale)
    ## Set the prior over weights
    self.prior_mean_W, self.log_prior_var_W,
    self.log_prior_nu_W = self.get_prior_W()
    ## Initialize posterior parameters
    if self.learn_Omega == "var":
        self.mean_Omega, self.log_var_Omega,
        self.log_nu_Omega = self.init_posterior_Omega()
    elif self.learn_Omega == "optim":
        self.mean_Omega, self.log_var_Omega,
        self.log_nu_Omega = self.init_posterior_Omega()
    else:
        self.mean_Omega, self.log_var_Omega,
        self.log_nu_Omega = self.get_prior_Omega([-1/2] * self.nl)
    self.mean_W, self.log_var_W,
    self.log_nu_W = self.init_posterior_W()
    ## Set the number of Monte Carlo samples as a placeholder
    ## so that it can be different for training and test
    self.mc =  tf.placeholder(tf.int32)
    ## Batch data placeholders
    Din = d_in
    Dout = d_out
    self.X = tf.placeholder(tf.float32, [None, Din])
    self.Y = tf.placeholder(tf.float32, [None, Dout])
    ## Builds whole computational graph with relevant
    ## quantities as part of the class
    self.loss, self.kl, self.ell, self.layer_out = self.get_nelbo()
    config = tf.ConfigProto()
    config.gpu_options.allow_growth=True
    ## Initialize the session
    self.session = tf.Session(config=config)

## Definition of a prior for Omega - which depends on the
## lengthscale of the covariance function
def get_prior_Omega(self, log_lengthscale):
    if self.is_ard:
        prior_mean_Omega = []
        log_prior_var_Omega = []
        log_prior_nu_Omega = []
```

```python
            for i in range(self.nl):
                prior_mean_Omega.append(tf.zeros([self.d_in[i],1]))
            for i in range(self.nl):
                log_prior_var_Omega.append(-2 * log_lengthscale[i])
            for i in range(self.nl):
                log_prior_nu_Omega.append(np.log(2.1) \
                * tf.ones([self.d_in[i],1]))
        else:
            prior_mean_Omega = tf.zeros(self.nl)
            log_prior_var_Omega = -2 * log_lengthscale
            log_prior_nu_Omega = np.log(2.1) * tf.ones(self.nl)
        return prior_mean_Omega, log_prior_var_Omega,
                                  log_prior_nu_Omega

    ## Definition of a prior over W - these are standard normals
    def get_prior_W(self):
        prior_mean_W = tf.zeros(self.n_W)
        log_prior_var_W = tf.zeros(self.n_W)
        log_prior_nu_W = tf.ones(self.n_W) * np.log(2.1)
        return prior_mean_W, log_prior_var_W, log_prior_nu_W

    ## Function to initialize the posterior over omega
    def init_posterior_Omega(self):
        mu, sigma2, nu = self.get_prior_Omega(self.llscale0)
        mean_Omega = [tf.Variable(mu[i] * tf.ones([self.d_in[i],
                                  self.d_out[i]]), name="q_Omega") \
                                  for i in range(self.n_Omega)]
        log_var_Omega = [tf.clip_by_value( tf.Variable(sigma2[i] \
                         * tf.ones([self.d_in[i], self.d_out[i]]),
                                  name="q_Omega"), -5, 5 ) \
                                  for i in range(self.n_Omega)]
        log_nu_Omega = [tf.clip_by_value( tf.Variable(nu[i] \
                         * tf.ones([self.d_in[i], self.d_out[i]]), \
                                  name="q_Omega"), np.log(2.1),
                                  np.log(10) ) \
                                  for i in range(self.n_Omega)]
        return mean_Omega, log_var_Omega, log_nu_Omega

    ## Function to initialize the posterior over W
    def init_posterior_W(self):
        mean_W = [tf.Variable(tf.zeros([self.dhat_in[i],
                         self.dhat_out[i]]), name="q_W") \
```

```
                    for i in range(self.n_W)]
    log_var_W = [tf.clip_by_value( tf.Variable(tf.zeros(\
                    [self.dhat_in[i], self.dhat_out[i]]),
                    name="q_W"), -5, 5 ) \
                    for i in range(self.n_W)]
    log_nu_W = [tf.clip_by_value( tf.Variable(tf.zeros(\
                    [self.dhat_in[i], self.dhat_out[i]]),
                     name="q_W"), np.log(2.1), np.log(10) ) \
                    for i in range(self.n_W)]
    return mean_W, log_var_W, log_nu_W

## Function to compute the KL divergence between priors and
## approximate posteriors over model parameters (Omega and W)
## when q(Omega) is to be learned
def get_kl_Omega_to_learn(self):
    kl = 0
    for i in range(self.n_Omega):
        kl = kl + utils.DKL_t(self.mean_Omega[i],
                    self.log_var_Omega[i], self.log_nu_Omega[i],
                    self.prior_mean_Omega[i],
                    self.log_prior_var_Omega[i],
                    self.log_prior_nu_Omega[i])
    for i in range(self.n_W):
        kl = kl + utils.DKL_t(self.mean_W[i],
                    self.log_var_W[i], self.log_nu_W[i],
                    self.prior_mean_W[i],
                    self.log_prior_var_W[i],
                    self.log_prior_nu_W[i])
    return kl

## Function to compute the KL divergence between priors and
## approximate posteriors over model parameters (W only) when
## q(Omega) is not to be learned
def get_kl_Omega_fixed(self):
    kl = 0
    for i in range(self.n_W):
        kl = kl + utils.DKL_t(self.mean_W[i],
                    self.log_var_W[i], self.log_nu_W[i],
                    self.prior_mean_W[i],
                    self.log_prior_var_W[i],
                    self.log_prior_nu_W[i])
    return kl
```

```
## Returns samples from approximate posterior over Omega
def sample_from_Omega_to_learn(self):
    Omega_from_q = []
    for i in range(self.n_Omega):
        z = utils.get_normal_samples(self.mc, self.d_in[i],
                                     self.d_out[i])
        nu = tf.exp(self.log_nu_Omega[i])
        const = tf.sqrt( nu/(nu+2) )
        Omega_from_q.append(tf.add(tf.mul(z, const \
                * tf.exp(self.log_var_Omega[i] / 2)),
                self.mean_Omega[i]))
    return Omega_from_q

## Returns Omega values calculated from fixed random variables
## and mean and variance of q() - the latter are optimized
## and enter the calculation of the KL so also lengthscale
## parameters get optimized
def sample_from_Omega_optim(self):
    Omega_from_q = []
    for i in range(self.n_Omega):
        z = tf.mul(self.z_for_Omega_fixed[i], tf.ones([self.mc,
                          self.d_in[i], self.d_out[i]]))
        nu = tf.exp(self.log_nu_Omega[i])
        const = tf.sqrt( nu/(nu+2) )
        Omega_from_q.append(tf.add(tf.mul(z, const \
                * tf.exp(self.log_var_Omega[i] / 2)),
                self.mean_Omega[i]))
    return Omega_from_q

## Returns samples from prior over Omega - in this case,
## randomness is fixed throughout learning (and Monte Carlo
## samples)
def sample_from_Omega_fixed(self):
    Omega_from_q = []
    for i in range(self.n_Omega):
        z = tf.mul(self.z_for_Omega_fixed[i],
                tf.ones([self.mc, self.d_in[i], self.d_out[i]]))
        nu = tf.exp(self.log_nu_Omega[i])
        const = tf.sqrt( nu/(nu+2) )
        if self.is_ard == True:
            reshaped_log_prior_var_Omega = \
```

```
                tf.tile(tf.reshape(self.log_prior_var_Omega[i] / 2,
                    [self.d_in[i],1]), [1,self.d_out[i]])
            Omega_from_q.append(tf.mul(z,
                    const * tf.exp(reshaped_log_prior_var_Omega)))
        if self.is_ard == False:
            Omega_from_q.append(tf.add(tf.mul(z,
                    const * tf.exp(self.log_prior_var_Omega[i] / 2)),
                    self.prior_mean_Omega[i]))

    return Omega_from_q

## Returns samples from approximate posterior over W
def sample_from_W(self):
    W_from_q = []
    for i in range(self.n_W):
        z = utils.get_normal_samples(self.mc, self.dhat_in[i],
                                    self.dhat_out[i])
        nu = tf.exp(self.log_nu_W[i])
        const = tf.sqrt( nu/(nu+2) )
        W_from_q.append(tf.add(tf.mul(z, const \
            * tf.exp(self.log_var_W[i] / 2)), self.mean_W[i]))
    return W_from_q

## Returns the expected log-likelihood term in the variational
## lower bound
def get_ell(self):
    Din = self.d_in[0]
    MC = self.mc
    N_L = self.nl
    X = self.X
    Y = self.Y
    batch_size = tf.shape(X)[0]
    ## This is the actual batch size when X is passed to the graph
    ## of computations
    ## The representation of the information is based on
    ## 3-dimensional tensors (one for each layer)
    ## Each slice [i,:,:] of these tensors is one Monte Carlo
    ## realization of the value of the hidden units
    ## At layer zero we simply replicate the input matrix X
    ## self.mc times
    self.layer = []
    self.layer.append(tf.mul(tf.ones([self.mc, batch_size, Din]),
```

```
                                                      X))
## Forward propagate information from the input to the output
## through hidden layers
Omega_from_q  = self.sample_from_Omega()
W_from_q = self.sample_from_W()
for i in range(N_L):
    layer_times_Omega = tf.batch_matmul(self.layer[i],
                                        Omega_from_q[i])
    ## Apply the activation function corresponding to the
    ## chosen kernel - PHI
    if self.kernel_type == "RBF":
        Phi = tf.exp(0.5 * self.log_theta_sigma2[i]) \
        / (tf.to_float(tf.sqrt(1. * self.n_rff[i]))) \
        * tf.concat(2,[tf.cos(layer_times_Omega),
                        tf.sin(layer_times_Omega)])
    if self.kernel_type == "arccosine":
        if self.arccosine_degree == 0:
            Phi = tf.exp(0.5 * self.log_theta_sigma2[i]) \
            / (tf.to_float(tf.sqrt(1. * self.n_rff[i]))) \
            * tf.concat(2,
                [tf.sign(tf.maximum(layer_times_Omega, 0.0))])
        if self.arccosine_degree == 1:
            Phi = tf.exp(0.5 * self.log_theta_sigma2[i]) \
            / (tf.to_float(tf.sqrt(1. * self.n_rff[i]))) \
            * tf.concat(2,
                [tf.maximum(layer_times_Omega, 0.0)])
        if self.arccosine_degree == 2:
            Phi = tf.exp(0.5 * self.log_theta_sigma2[i]) \
            / (tf.to_float(tf.sqrt(1. * self.n_rff[i]))) \
            * tf.concat(2,
                [tf.square(tf.maximum(layer_times_Omega, 0.0))])
    F = tf.batch_matmul(Phi, W_from_q[i])
    if self.feed_forward and not (i == (N_L-1)):
        ## In the feed-forward case, no concatenation in the
        ## last layer so that F has the same dimensions of Y
        F = tf.concat([F, self.layer[0]], 2)
    self.layer.append(F)
## Output layer
layer_out = self.layer[N_L]
## Given the output layer, we compute the conditional
## likelihood across all samples
ll = self.likelihood.log_cond_prob(Y, layer_out)
```

```python
    ## Mini-batch estimation of the expected log-likelihood term
    ell = tf.reduce_sum(tf.reduce_mean(ll, 0)) * self.num_examples \
                                / tf.cast(batch_size, "float32")
    return ell, layer_out

## Maximize variational lower bound --> minimize Nelbo
def get_nelbo(self):
    kl = self.get_kl()
    ell, layer_out = self.get_ell()
    nelbo  = kl - ell
    return nelbo, kl, ell, layer_out

## Return predictions on some data
def predict(self, data, mc_test):
    out = self.likelihood.predict(self.layer_out)
    nll = - tf.reduce_sum(-np.log(mc_test) \
            + utils.logsumexp(self.likelihood.log_cond_prob(self.Y,
                                                  self.layer_out), 0))
    #nll = - tf.reduce_sum(tf.reduce_mean( \
                self.likelihood.log_cond_prob(self.Y,
                                                self.layer_out), 0))
    if type(data) == list:
        pred, neg_ll = self.session.run([out, nll],
                          feed_dict={self.X:data[0],
                                     self.Y: data[1],
                                     self.mc:mc_test})
    else:
        pred, neg_ll = self.session.run([out, nll],
                          feed_dict={self.X:data.X,
                                     self.Y: data.Y,
                                     self.mc:mc_test})
    mean_pred = np.mean(pred, 0)
    return mean_pred, neg_ll

## Return the list of TF variables that should be "free"
## to be optimized
def get_vars_fixing_some(self, all_variables):
    if (self.q_Omega_fixed_flag == True) and \
            (self.theta_fixed_flag == True):
        variational_parameters = [v for v in all_variables \
                if (not v.name.startswith("q_Omega") and \
                    not v.name.startswith("log_theta"))]
```

```
    if (self.q_Omega_fixed_flag == True) and \
                (self.theta_fixed_flag == False):
        variational_parameters = [v for v in all_variables \
                if (not v.name.startswith("q_Omega"))]
    if (self.q_Omega_fixed_flag == False) and \
                (self.theta_fixed_flag == True):
        variational_parameters = [v for v in all_variables \
                if (not v.name.startswith("log_theta"))]
    if (self.q_Omega_fixed_flag == False) and \
                (self.theta_fixed_flag == False):
        variational_parameters = all_variables
    return variational_parameters

## Function that learns the deep GP model with random Fourier
## feature approximation
def learn(self, data, learning_rate, mc_train, batch_size,
          n_iterations, optimizer = None, display_step=100,
          test = None, mc_test=None, loss_function=None,
          duration = 1000000, less_prints=False):
    total_train_time = 0
    if optimizer is None:
        optimizer = tf.train.AdadeltaOptimizer(learning_rate)
    ## Set all_variables to contain the complete set of TF variables
    ## to optimize
    all_variables = tf.trainable_variables()
    ## Define the optimizer
    train_step = optimizer.minimize(self.loss, var_list=all_variables)
    ## Initialize all variables
    init = tf.global_variables_initializer()
    ## Fix any variables that are supposed to be fixed
    train_step = optimizer.minimize(self.loss,
                    var_list=self.get_vars_fixing_some(all_variables))
    ## Initialize TF session
    self.session.run(init)
    ## Set the folder where the logs are going to be written
    summary_writer = tf.summary.FileWriter('logs/', self.session.graph)
    if not(less_prints):
        nelbo, kl, ell, _ =  self.session.run(self.get_nelbo(),
                    feed_dict={self.X: data.X, self.Y: data.Y,
                               self.mc: mc_train})
        print("Initial kl=" + repr(kl) + "  nell=" \
                + repr(-ell) + "  nelbo=" + repr(nelbo), end=" ")
```

```
    print("  log-sigma2 =", self.session.run(self.log_theta_sigma2))
minimum = 100
iter_ = 0
## Present data to DGP n_iterations times
for iteration in range(n_iterations):
    ## Stop after a given budget of minutes is reached
    if (total_train_time > 1000 * 60 * duration):
        break
    ## Present one batch of data to the DGP
    start_train_time = current_milli_time()
    batch = data.next_batch(batch_size)
    X_train_batch = batch[0]
    monte_carlo_sample_train = mc_train
    if (current_milli_time() - start_train_time) < \
                        (1000 * 60 * duration / 2.0):
        monte_carlo_sample_train = 1
    self.session.run(train_step,
                feed_dict={self.X: X_train_batch,
                           self.Y: batch[1],
                           self.mc: monte_carlo_sample_train})
    total_train_time += current_milli_time() - start_train_time
    ## Display logs every "FLAGS.display_step" iterations
    if iteration % display_step == 0:
        start_predict_time = current_milli_time()
        if less_prints:
            print("i=" + repr(iteration), end = " ,")
        else:
            nelbo, kl, ell,
                _ = self.session.run(self.get_nelbo(),
                                feed_dict={self.X: data.X,
                                           self.Y: data.Y,
                                           self.mc: mc_train})
            print("i=" + repr(iteration)  + \
                    "  kl=" + repr(kl) + "  nell=" + \
                    repr(-ell)  + "  nelbo=" + repr(nelbo),
                    end=" ")
            print(" log-sigma2=",
                    self.session.run(self.log_theta_sigma2),
                    end=" ")
        if loss_function is not None:
            pred, nll_test = self.predict(test, mc_test)
            elapsed_time = total_train_time + \
```

```
                    (current_milli_time() - start_predict_time)
                loss_t = loss_function.eval(test.Y, pred)
                print(loss_function.get_name() + "=" + \
                    "%.4f" % loss_t, end = " ,")
                print(" nll_test=" + \
                    "%.5f" % (nll_test / len(test.Y)), end = " ,")
            print(" time=" + repr(elapsed_time), end = " ")
            print("")
            if loss_t<minimum:
                minimum = loss_t
                iter_ = iteration
            if iteration > iter_ + 5000:
                return minimum, iter_
    return minimum, iter_
```