M.Sc.Eng. Thesis
Master of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Master's Thesis

Variational Optimization of Neural Networks

Author: Jakob Drachmann Havtorn
Supervisor: Ole Winther

Kongens Lyngby 2018

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads
Bygning 324
2800 Kgs. Lyngby
compute@compute.dtu.dk
+45 4525 3031
CVR-nr. 30 06 09 46
EAN-nr. 5798000428515

Abstract

This thesis presents evolution strategies as a competitive alternative to classical methods for reinforcement learning and unifies previous work within black-box optimization with deep learning. It presents *variational optimization* as an encompassing mathematical framework, which maintains a search distribution over the optimized parameters during training, and describes how it can be efficiently applied for optimization of neural networks without the use of backpropagation.
The natural gradient is derived and implemented as a way to update a search distribution subject to a similarity constraint w.r.t. the previous iterate. It is shown to be superior to using the regular gradient, especially when adapting the variance of a Gaussian search distribution. Antithetic sampling and the method of common random numbers are derived and applied to reduce the variance of the gradient. Experiments run primarily in the supervised setting on the MNIST dataset show that while antithetic sampling is rather efficient at achieving this goal, common random numbers is not. A novel approach for reduction of gradient variance as well as computation based on a local reparameterization of feedforward neural networks is presented and treated theoretically.
Different search distributions based on the Gaussian are derived and implemented and the effect of adapting the mean and variance is compared to adapting only the mean which parameterizes the network parameters. It is found that while using an isotropic Gaussian with fixed variance provides good results, adapting the variance can lead to divergence. Separable Gaussians with a variance per network layer or per network weight are shown to perform similarly to using a fixed variance but not significantly better. These results are discussed and related to the geometry of the loss surface.

Keywords: Variational optimization; Search distribution; Monte Carlo; Natural gradient; Fischer information; Sensitivity analysis; Variance reduction; Antithetic sampling; Local reparameterization; Deep learning

Preface

This Master's thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the official requirements for acquiring the degree of Master of Science in Engineering in Mathematical Modelling and Computation.

Kongens Lyngby, July 3, 2018

Jakob D. Havtorn

Jakob Drachmann Havtorn

s132315

Acknowledgements

For the time spent and many valuable insights provided, I would like to thank my supervisor Professor Ole Winther. Many interesting discussions with him and the evolutionary strategies study group have provided food for thought throughout the course of this thesis. I look forward to hopefully many more talks in the future.

I am particularly grateful for the support of my caring girlfriend and loving partner through the last decade, Rikke M. Fogh. With her wonderful being and unwavering and tolerant nature she has supported me in this undertaking at every step of the way.

I would also like to place a special thanks to friends I have met along the way. A special thanks is extended to Karl K. Toudahl and Lasse Blaabjerg without whom my time at DTU would surely have been more productive for however long I would have endured without their company.

Finally, I wish to also thank my sister, parents and family for their support and encouragement during my studies.

Abbreviations

AI artificial intelligence

BNN Bayesian neural network

CEL cross entropy loss
CMA-ES Covariance Matrix Adaptation Evolution Strategy
CNN convolutional neural network
CPU central processing unit
CRN common random numbers

DRL deep reinforcement learning

ES evolution stategy

FIM Fisher information matrix
FNN feedforward neural network

GPU graphics processing unit
GRU gated recurrent unit

HPC high performance computing

IID independent and identically distributed

KL Kullback-Leibler

LSTM long short-term memory

MDP Markov decision process
ML machine learning
MLE maximum likelihood estimate
MLP multilayer perceptron
MNIST Modified National Institute of Standards and Technology database
MSE mean squared error

NLL negative log-likelihood
NN neural network

PCA principal components analysis
PDF probability density function

ReLU rectified linear unit
RL reinforcement learning
RNN recurrent neural network

SGD stochastic gradient descent
SGVB stochastic gradient variational Bayes
SVD singular value decomposition

VO variational optimization

Notation

This section collects definitions of the notation used in the thesis. Much of this is defined in the thesis in the order that it is used but is collected here for reference and convenience. There are no major departures from convention.

	Numbers and arrays
a	A scalar; integer or real
$\mathbf{a}$	A column vector
$\mathbf{A}$	A matrix
$\mathrm{diag}(\mathbf{a})$	A square diagonal matrix of the elements of $\mathbf{a}$
$\mathbf{e}$	A vector of ones, $\mathbf{e} = \begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}^{\mathrm{T}}$
$\mathbf{I}$	The identity matrix, $\mathbf{I} = \mathrm{diag}(\mathbf{e})$
	Sets
$\mathcal{A}, \mathbb{A}$	A set
$\mathbb{R}$	Set of real numbers
$\{s_1, s_2, \dots, s_n\}$	Set of scalars $s_1, s_2, \dots, s_n$; also written as $\{s_i\}_{i=1}^n$
$(a, b]$	Semi-inclusive real interval between a and b, $(a,b] = \{x \in \mathbb{R} \mid a < x \leq b\}$
	Indexing
a_i	Element i of vector $\mathbf{a}$ with first index 1
$A_{i,j}$	Element i, j of matrix $\mathbf{A}$. Comma omitted when unambiguous.
$A_{i,:}$	Row i of matrix $\mathbf{A}$ (a row vector)
$A_{:,j}$	Column j of matrix $\mathbf{A}$ (a column vector)
$\mathbf{a}^{[l]}$	Neural network layer indexing; a vector in layer l
$\mathbf{a}^{\langle t \rangle}$	Recurrent neural network sequence indexing; vector at timestep t in sequence $\left\{\mathbf{a}^{\langle t \rangle}\right\}_{t=1}^{T}$

Linear algebra operations

$\mathbf{a}^{\mathrm{T}}$	Transpose of column vector $\mathbf{a}$ to give row vector
$\mathbf{A}^{\mathrm{T}}$	Transpose of matrix $\mathbf{A}$
$\mathbf{a}^{\mathrm{T}}\mathbf{b}$	Inner product of $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{a}^{\mathrm{T}}\mathbf{b} = \mathbf{a} \cdot \mathbf{b}$
$\mathbf{a}\mathbf{b}^{\mathrm{T}}$	Outer product of $\mathbf{a}$ and $\mathbf{b}$
$\mathbf{A}\mathbf{B}$	Matrix product of $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} * \mathbf{B}$	Convolution (actually cross-correlation) of $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \odot \mathbf{B}$	Elementwise, or Hadamard, product of $\mathbf{A}$ and $\mathbf{B}$
$\det \mathbf{A}$	Determinant of matrix $\mathbf{A}$
$\lVert\mathbf{a}\rVert$	L^2 norm of $\mathbf{a}$
$\lVert\mathbf{a}\rVert_p$	L^p norm of $\mathbf{a}$
$\mathrm{Tr}\left[\mathbf{A}\right]$	Trace of matrix $\mathbf{A}$, $\mathrm{Tr}\left[\mathbf{A}\right] = \sum_i A_{i,i}$

Calculus

$\frac{\mathrm{d}y}{\mathrm{d}x}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\mathbf{x}}$	Del, nabla or Laplacian operator, $\nabla_{\mathbf{x}} = \left[\frac{\partial}{\partial x_1} \quad \cdots \quad \frac{\partial}{\partial x_n}\right]^{\mathrm{T}}$, $\mathbf{x} \in \mathbb{R}^n$
$\nabla_{\mathbf{x}} y$	Gradient of y with respect to $\mathbf{x}$, $\nabla_{\mathbf{x}} y = \left[\frac{\partial y}{\partial x_1} \quad \cdots \quad \frac{\partial y}{\partial x_n}\right]^{\mathrm{T}}$, $\mathbf{x} \in \mathbb{R}^n$
$\mathbf{H}_{\mathbf{x}}$	Hessian operator, $\mathbf{H}_{\mathbf{x}} = \nabla_{\mathbf{x}} {\nabla_{\mathbf{x}}}^{\mathrm{T}} = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} & \frac{\partial^2}{\partial x_1 \, \partial x_2} & \cdots & \frac{\partial^2}{\partial x_1 \, \partial x_n} \\ \frac{\partial^2}{\partial x_2 \, \partial x_1} & \frac{\partial^2}{\partial x_2^2} & \cdots & \frac{\partial^2}{\partial x_2 \, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \, \partial x_1} & \frac{\partial^2}{\partial x_d \, \partial x_2} & \cdots & \frac{\partial^2}{\partial x_n^2} \end{bmatrix}$
$\mathbf{H}_{\mathbf{x}} y$	Hessian matrix of y with respect to $\mathbf{x}$; matrix of all second order partial derivatives
$\nabla^2_{\mathbf{x}}$	Laplacian operator, $\nabla^2_{\mathbf{x}} = {\nabla_{\mathbf{x}}}^{\mathrm{T}} \nabla_{\mathbf{x}} = \nabla_{\mathbf{x}} \cdot \nabla_{\mathbf{x}} = \sum_i \frac{\partial^2}{\partial x_i^2} = \mathrm{Tr}\left[\mathbf{H}_{\mathbf{x}}\right]$
$\nabla^2_{\mathbf{x}} y$	Laplacian of y with respect to $\mathbf{x}$; sum of the unmixed second order partial derivatives
$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$	Jacobian matrix of $\mathbf{y}$ w.r.t. $\mathbf{x}$, $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \nabla^{\mathrm{T}}_{\mathbf{x}} y_1 \\ \vdots \\ \nabla^{\mathrm{T}}_{\mathbf{x}} y_n \end{bmatrix}$, $\mathbf{y} \in \mathbb{R}^n$; matrix of all possible first order partial derivatives

$\int f(\mathbf{x})\,\mathrm{d}\mathbf{x}$	Definite integral over the domain of $\mathbf{x}$
$\int_{\mathcal{D}} f(\mathbf{x})\,\mathrm{d}\mathbf{x}$	Definite integral over the domain $\mathcal{D}$
	Probability theory
$a \sim p$	a is a random variable with distribution p
$\mathrm{E}[f(x)]$	Expectation of $f(x)$ with respect to $p(x)$; sometimes also $\mathrm{E}[f(x)]_{x\sim p(x)}$
$\mathrm{Var}[f(x)]$	Variance of $f(x)$ under $p(x)$
$\mathrm{Cov}[f(x), g(x)]$	Covariance of $f(x)$ and $g(x)$ under $p(x)$
$D_{\mathrm{KL}}(p \,\|\|\, q)$	Kullback-Leibler divergence of p and q
$D_{\mathrm{KL}}(p, q)$	Symmetrized Kullback-Leibler divergence of p and q
$\mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\mathbf{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
	Functions
$f : \mathcal{D} \to \mathcal{C}$	Function f with domain $\mathcal{D}$ and co-domain $\mathcal{C}$
$\log x$	Natural logarithm of x
$\mathrm{ReLU}(x)$	Linear rectifier function, $\mathrm{ReLU}(x) = \max\{0, x\}$ (Rectified Linear Unit)

Contents

CHAPTER 1 Introduction

"We have at our command computers with adequate data-handling ability and with sufficient computational speed to make use of machine-learning techniques, but our knowledge of the basic principles of these techniques is still rudimentary. Lacking such knowledge, it is necessary to specify methods of problem solution in minute and exact detail, a time-consuming and costly procedure. Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort."

- Samuel, Arthur L. (1959) [91]

1.1 Learning machines

The idea of learning machines is not a new one. Great minds have tinkered with the idea of an intelligent machine ever since Aristotle first defined his logic of syllogisms; perhaps even earlier. More recently, research into artificial intelligence (AI) has experienced periods of varying funding and public popularity. The first years of the Cold War saw increased interest in automated computer processing along with the development of the perceptron [86] in 1957 which sparked optimistic dreams of an artificial machine intelligence. However, a combination of events such as the so-called Lighthill report [58] and the book by Minsky and Papert [66] contributed to the start of a period of decreased funding in AI research. This period, starting in the 1970's and ending in the 1990's, often goes by the figurative name of the "AI winter".

In 1986, Rumelhart, Hinton and Williams popularized the backpropagation algorithm that allowed for efficiently training increasingly advanced, now multilayered, perceptrons by error gradients [87]. The following years saw the shift from a knowledge-driven approach to machine learning to a data-driven one and with it came developments such as 1D and 2D convolutional neural networks in 1988 and 1989 [37, 51] and the support vector machine in 1995 [17].

In the late 2000's, driven primarily by easy access to large datasets and increased computational power, the machine learning subfield of deep learning has seen a major revival. Many tasks that were previously dominated by other machine learning (ML) models often relying on engineered features have been bested by deep neural networks applied to raw data often rivalling or even exceeding human performance in limited domains [96]. Although an artificial general intelligence is far from being reality, new developments and new potential applications of ML seem to be everyday occurrences.

1.2 Problem types in machine learning

Any problem addressed by machine learning (ML) can generally be categorized as belonging to one of three major types of learning or an intersection between them:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

To motivate the main topic of this thesis, these are introduced below.

1.2.1 Supervised learning

In supervised learning, a model is trained to learn a mapping from inputs $\mathbf{x}_i$ to outputs y_i. A dataset is given, consisting of examples of inputs along with associated outputs, or labels. Each pair represents a specific instance of the mapping to be learned. A dataset with N example pairs can be denoted by

$$\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^{N}. \tag{1.2.1}$$

The dataset is often split in three parts used for *training*, *validation* and *testing*. The validation set is used for tuning of hyperparameters while model comparison is done on the test set. The need for a validation set does not arise for all choices of model and depends also on the problem type.

In a probabilistic sense, the learned mapping can be represented as a conditional probability distribution $p(y_i|\mathbf{x}_i, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ denotes the parameters of the model to be learned. Using an algorithm dependent on the model and the problem, the model is trained on the dataset. The main goal of training is then for the model to learn to make good predictions from previously unseen inputs. This is called the model's ability to *generalize*. Supervised learning can be split in two major cases, classification and regression [70].

1.2.1.1 Classification

In classification, the response variable is *discrete*, $y \in \{1, \dots, K\}$, and denotes a class label out of K classes. The task is then to associate each input with the correct class. Examples of classification tasks include document classification and image classification with specific instances such as e-mail spam filtering and face detection [70].

1.2.1.2 Regression

Regression is similar to classification but with a *continuous* response variable, $y \in \mathbb{R}$. This continuous response can be anything from stock prices and air pollution levels to continuous control tasks such as manipulating actuators [70]. Classification and regression can also be combined. For example, locating a certain object within an image with a bounding box can be formulated as a regression problem while predicting the type of object is a typical classification task [101, 83].

1.2.2 Unsupervised learning

The unsupervised learning problem is characterized by the need to extract information from a dataset which has no labels associated with it, i.e.

$$\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^{N} \ . \tag{1.2.2}$$

The dataset may still be divided in training, validation and testing parts as for the supervised setting while the problem can be formulated as one of *density estimation* in which the model learns the underlying distribution of the data, $p(\mathbf{x}_i|\boldsymbol{\theta})$, or at least some representation of it [70].

1.2.2.1 Clustering

Unsupervised learning has several subtasks such as clustering, where the representation of the data is used to discover different groups, or clusters, of data within the data itself. This is akin to the supervised classification task but in the unsupervised setting, the engineer must decide on the number of clusters to use ahead of time or formulate a model which is able to do this automatically [70].

1.2.2.2 Dimensionality reduction

Another subtask of unsupervised learning is dimensionality reduction, where a latent representation of the data is optimized to encode as much information about the data as possible while reducing the total number of dimensions. This can be used to reduce computation in downstream algorithms and models or, if the resulting new dimensionality is low enough, typically 2 or 3, it can be used for visualization of data [70].

1.2.3 Reinforcement learning

Although reinforcement learning (RL) is sometimes viewed as a special case of supervised learning, it can also be considered different enough in some aspects to classify as a separate setting of machine learning [112].

As in unsupervised learning, in RL there is no labelled dataset on which to train the algorithm. Rather, the model, or *agent*, learns by trial and error in simulation. Here it is alternately presented with an *observation* from the simulation, or *environment*, and required to take some *action* as a response. After taking some action, the agent receives a *reward* whose size depends on how well the agent performed before being presented with the next observation. Rewards can be awarded densely after each step or sparsely with the most extreme case being a single reward signal for an entire simulation from *initial state* to *terminal state*, also called an *episode* [112].

Similarly to supervised learning, RL then requires definition of some sort of measure that converts the reward into a learning signal. Since the definition of optimal behaviour is often impossible in complex systems, this measure is some approximation relying on certain assumptions about the environment. For example, environments are often assumed to be Markovian[1]

[1]For an environment or system to be Markovian it must satisfy that the future states of the system only depend on the present state, and not the past. In other words, given the present, the future is independent of the past. That entails that all information necessary about the past must be able to encoded into the present state. A Markovian system is also said to have the Markov property.

and decision-making in such a system can be done using the Markov decision process (MDP) formalism [112].

Recently, the surge in popularity of neural networks (NNs) has had a significant impact on RL giving rise to the field of deep reinforcement learning (DRL) which utilizes NN models to encode the learned behaviour or knowledge about the environment [57]. Examples are policy networks [119, 100, 102, 99, 67, 98] and value function networks such as Q networks [42, 68]. In all cases, the learning signal is obtained through an objective function which approximates the 'true' and undefined objective. These examples also outline two major approaches to RL: Methods searching for value functions and methods searching directly for policies. Q-learning is a value function method which tries to learn the value of an action given the current state. The optimal policy is then simply obtained by choosing the action which maximizes the optimal Q-function. Direct policy search parameterizes a policy, evaluates it on the environment and adjusts the policy according to an estimated reward gradient [42]. Both methods have their merits although recently, Q-learning and value function approaches have had great successes such as learning to play Atari from pixels [68] and mastering the game of Go [103].

So-called evolution strategies (ESs) have recently been suggested as an alternative approach to DRL by OpenAI[2] [90]. As with much of ML in general and RL in particular, ESs are widely based on well-established ideas and has a fairly rich literature. ESs dispense with approximating objectives and instead rely exclusively on the reward signal for learning. This approach is fairly easily parallelized to a large number of central processing units (CPUs) which enables fast training making it competitive with classical approaches to DRL in certain situations.

1.3 Scope and delimitations

This thesis seeks to unify previous work done within black-box and non-differentiable optimization with recent advances within deep learning. It seeks to find a broader mathematical framework for describing ES as Monte Carlo based stochastic gradient estimator and to further analyze the properties of the estimator. Effort is put towards reducing the variance of the estimator, reducing the amount of required computation and generally improving its performance. Experiments are done to validate the theoretical results of the thesis.

The investigative nature of the work of this thesis has lead to consideration of a very broad range of literature. Consequently, the associated set of related material is immense and as such, many related methods may have been only superficially noted or even overlooked.

1.4 Structure of the thesis

In Chapter 2, the neural network is introduced as the central ML model of this thesis. The three main types of network are described, along with methods for efficiently training them including the backpropagation algorithm. The chapter also includes an overview of the optimization algorithms, regularization techniques and parameter initialization schemes used in the thesis.

Chapter 3 constitutes the main part of the thesis and is concerned with variational optimization as a general and theoretically grounded framework for describing ESs. Several augmentations to this framework are considered including use of the natural gradient, inclu-

[2]OpenAI is a non-profit research company concerned with developing "safe AI", `https://openai.com/`

sion of gradient information for sensitivity rescaled perturbations and methods for variance reduction.

Second to last, Chapter 4 presents experimental results that seek to validate the variational gradient estimators as well as compare their performance. The computational scaling of the algorithm is first presented. Experiments then include validation of the effect of the deep learning methods of batch normalization and dropout when used in conjunction with variational optimization as well as the effect of rescaling perturbations according to network sensitivity. The different methods for variance reduction are also tested and finally, different variations on the variational optimization algorithm are compared.

Finally, the thesis is concluded in Chapter 6.

CHAPTER 2

Neural networks

"Since the advent of electronic computers and modern servo systems, an increasing amount of attention has been focused on the feasibility of constructing a device possessing such human-like functions as perception, recognition, concept formation, and the ability to generalize from experience."

- Rosenblatt, Frank (1957) [86]

2.1 Introduction

This thesis will focus on NNs which are a specific but very flexible class of ML models that can be applied both in the supervised and unsupervised regimes as well as for RL. This section will introduce NNs and the three primary variations, feedforward neural networks (FNNs), convolutional neural networks (CNNs) and recurrent neural networks (RNNs) along with the conventional methods for training them.

2.2 Fundamental types of neural networks

2.2.1 The perceptron

Inspired by previous work on artificial neurons by McCulloch and Pitts [65], the first network of neurons, the perceptron, was developed by Frank Rosenblatt in 1957 [86]. Although Rosenblatt did in fact study multilayered perceptrons with some success [85, 9], perceptrons are often described as a single layered type of NN [8, 29]. Since this view serves as a natural introduction to NNs and in particular FNNs, and due to the differences between the multilayered perceptrons of Rosenblatt and more modern versions, this will also be the view taken in this presentation.

Perceptrons are binary linear classifiers in that they are able to learn a linear decision boundary between two classes. Given a vector of inputs $\mathbf{x} \in \mathbb{R}^I$ and a vector of learnable weights $\mathbf{w} \in \mathbb{R}^I$ along with a scalar bias parameter $b \in \mathbb{R}$, the perceptron computes a binary response $y \in \{-1, 1\}$ by the following linear transformation.

$$y = \varphi\left(\sum_{i=1}^{I} w_i x_i + b_i\right) = \varphi\big(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b\big) \tag{2.2.1}$$

where

$$\varphi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{2.2.2}$$

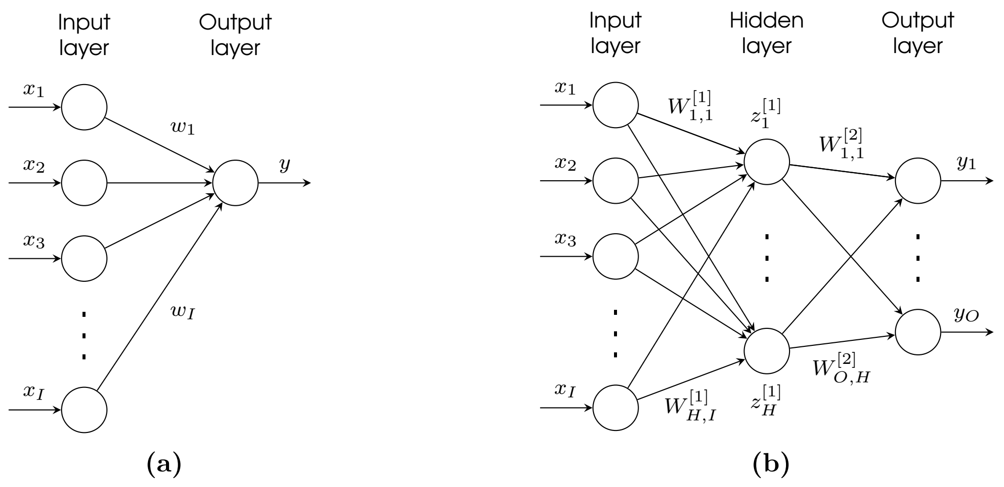


Figure 2.1: (a) A scalar computational graph of the perceptron model due to Rosenblatt [86]. The single scalar output is computed as the sign of the weighted sum of inputs. This is similar to the McCulloch-Pitts neuron [65]. **(b)** A scalar computational graph for a modern two-layer FNN with a single hidden layer and an output layer, $L = 2$. The output vector is computed through a series of affine transformations and associated nonlinear activation functions.

is a binarizing, or step, *activation function*. More compactly, the bias parameter can be included by introducing an artificial, or dummy, feature in $\mathbf{x}$ with the constant value of 1 and defining the bias to be the associated new element in $\mathbf{w}$. Then

$$y = \varphi\left(\sum_{i=1}^{I+1} w_i x_i\right) = \varphi\big(\mathbf{w}^{\mathrm{T}}\mathbf{x}\big) \tag{2.2.3}$$

where now $\mathbf{x}, \mathbf{w} \in \mathbb{R}^{I+1}$ [8].

Perceptrons, as well as all other NN models, can be represented as a computational graph. Such a representation of the perceptron without explicit biases can be seen in Figure 2.1a. Each node of the graph represents a single real number and every edge leaving some input layer node represents the multiplication of that input with a specific learnable weight. The incidence of the I edges on the output node represents the summation of these products to form a weighted sum of the inputs. As per the definition above, the output is then computed by application of the activation function φ to the weighted sum of inputs.

A straightforward way to define a learning algorithm for a perceptron is through error function minimization[1]. In error function minimization, a measure of the amount of error made by a model on a training set is defined and evaluated. Some optimization algorithm can then be applied to the gradient, and potentially the Hessian, of the error function in order to update the model parameters in the direction of decreasing error. An error function useful for training perceptrons is the so-called *perceptron criteria* given by

$$E_{\mathrm{P}}(\mathbf{w}) = -\frac{1}{N}\sum_{n\in\mathcal{M}} \mathbf{w}^{\mathrm{T}}\mathbf{x}_n t_n \tag{2.2.4}$$

[1]Within ML, *error function* is the widely used term for what is generally called the *objective function* in optimization. The name *cost function* is also often used while the term *loss function* can be used to mean the same but sometimes also refers to the error or cost associated with a single training example.

where t_n is the n'th target in a training set of N examples, and $\mathcal{M} = \{n \mid y_n \neq t_n\}$ is the set of misclassifications. Since $t_n > 0$ requires $\mathbf{w}^\mathrm{T}\mathbf{x}_n < 0$ for n to be misclassified, and vice versa, this error function measures the total amount of error made in misclassified samples and has a minimum of zero when $\mathcal{M}$ is empty. It can be noted that since E_P is zero in any region of $\mathbf{w}$ space that makes correct classifications and depends linearly on $\mathbf{w}$ in regions resulting in incorrect classifications, it is a piecewise linear function. The gradient of this error function w.r.t. the weights is

$$\nabla_\mathbf{w} E_\mathrm{P}(\mathbf{w}) = -\frac{1}{N}\sum_{n\in\mathcal{M}} \mathbf{x}_n t_n \tag{2.2.5}$$

and the Hessian is clearly zero. The gradient can be directly used for regular gradient descent or stochastic gradient descent (SGD) which uses a single training example to evaluate the gradient at each iteration, saving computation but also introducing stochasticity. The gradient descent update is

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla_\mathbf{w} E_\mathrm{P}(\mathbf{w}) = \mathbf{w} + \frac{1}{|\mathcal{B}|}\sum_{n\in\mathcal{B}} \mathbf{x}_n t_n \ , \tag{2.2.6}$$

where the batch $\mathcal{B}$ is equal to $\mathcal{M}$ for regular gradient descent. Whenever $|\mathcal{M}| > |\mathcal{B}| > 1$ the gradient descent scheme above is called mini-batch SGD. In the above, η has been set to one without loss of generality since multiplication of $\mathbf{w}$ by any positive constant does not change the perceptron output y [8].

2.2.2 Feedforward neural networks

Feedforward neural networks (FNNs) are conceptually a simple extension of perceptrons that contain multiple perceptron units in a layered feedforward fashion. Therefore, they are also known as multilayer perceptrons (MLPs).

An FNN can be represented mathematically by a series of affine transformations followed by a nonlinear activation function. Let $\mathbf{a}^{[0]} = \mathbf{x}$. The transformation applied between layers l and $l+1$ when forward propagating the input $\mathbf{x}$ can then be written as

$$\begin{aligned} \mathbf{z}^{[l+1]} &= \mathbf{W}^{[l+1]}\mathbf{a}^{[l]} + \mathbf{b}^{[l+1]} \\ \mathbf{a}^{[l+1]} &= \varphi_{l+1}\left(\mathbf{z}^{[l+1]}\right) \end{aligned} \quad , \quad l \in [0, L-1] \tag{2.2.7}$$

where φ_l is the activation function, or non-linearity, of the l'th layer, square brackets denote layer indexing and L is the total number of layers in the network. The $\mathbf{z}$ variables are often called *hidden units* while $\mathbf{a}$ are called *hidden unit activations*, or simply *activations*. Often, the NN output activations are denoted by $\mathbf{y}$ such that $\mathbf{y} = \mathbf{a}^{[L]}$ and the last layer activation is often different from the internal ones depending on application. The computation of an NN output from an input is called the *forward pass* or *forward propagation* [29]. Figure 2.1b shows the computational graph of a general two-layer FNN without explicit biases.

The first layer matrix is restricted to have I columns due to the input dimension. The remaining dimensions of all weight matrices then define the size of the FNN with biases $\mathbf{b}^{[l]}$ always being vectors of the same dimensionality as the rows in the corresponding weight matrix, $\mathbf{W}^{[l]}$.

To further illustrate the connection to perceptrons, note that the first row of the $\mathbf{W}^{[l]}$ matrix holds the weights of the perceptron model that describes the connection between $\mathbf{z}^{[l]}$

and $z_1^{[l+1]}$. For example, the a_1 activation is computed as follows.

$$\begin{aligned} z_1^{[1]} &= \sum_{i=1}^{I} W_{1,i}^{[1]} x_i = \mathbf{W}_{1,:}^{[1]\mathrm{T}} \mathbf{x} \\ a_1^{[1]} &= \varphi\left(z_1^{[1]}\right) \end{aligned} \tag{2.2.8}$$

which is equivalent to the equation of a single perceptron defined in (2.2.3). As such, each layer of an FNN consists of a number of perceptron models. Following layers then use the outputs of preceeding perceptrons as input.

Besides the multilayered structure, one major difference between FNNs and perceptrons is the use of real-valued activation functions. Where the perceptron used a binary threshold based activation function, modern FNNs typically use nonlinear, real-valued activation functions which are applied elementwise to the hidden units to form the activations. Development of new activation functions is an active field of research. Classical activation functions were the sigmoid $\sigma(\cdot)$ and the hyperbolic tangent,

$$\sigma(\mathbf{z}) = \frac{1}{1+\exp(-\mathbf{z})} = \frac{\exp(\mathbf{z})}{1+\exp(\mathbf{z})} \tag{2.2.9}$$

$$\tanh(\mathbf{z}) = \frac{\exp(\mathbf{z}) - \exp(-\mathbf{z})}{\exp(\mathbf{z}) - \exp(-\mathbf{z})} = 2\sigma(2\mathbf{z}) - 1 \ . \tag{2.2.10}$$

These functions are constrained to values in $[0, 1]$ and $[-1, 1]$, respectively, but suffer from extremely small gradients for inputs with large absolute values. Due to the way modern NNs are trained, this often risks slowing down or completely stopping training. More modern activation functions include the rectified linear unit (ReLU) [28] and the softplus,

$$\mathrm{ReLU}(\mathbf{z}) = \max\{0, \mathbf{z}\} \tag{2.2.11}$$

$$\mathrm{softplus}(\mathbf{z}) = \log\left(1 + \exp(\mathbf{z})\right) \ , \tag{2.2.12}$$

as well as many others. These generally seek to avoid small gradients while still being nonlinear, easy to compute and retaining some nice properties such as monotonicity. The chosen activations need not generally be the same throughout the network. In fact, the activation applied to the output layer is most often not and instead depends on the application. For regression tasks, the output layer often omits the activation function altogether since outputs are many times unbounded. For binary classification tasks, the sigmoid activation can be applied to an output layer with 1 unit, resulting in a logistic regression model. For multiclass classification with K classes, the softmax,

$$\mathrm{softmax}(\mathbf{z}) = \frac{\exp(\mathbf{z})}{\sum_{i=1}^{K} \exp(z_i)} \ , \tag{2.2.13}$$

is commonly applied to an output layer with K units [29].

2.2.3 Convolutional neural networks

A CNN is a type of NN that uses the convolution operation rather than the affine transformation employed by FNNs. The CNN was first introduced in the form of the Neocognitron

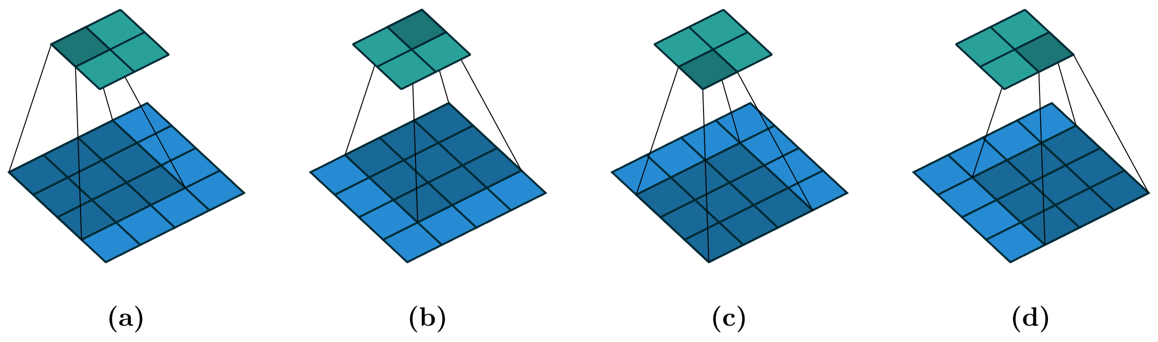


Figure 2.2: Convolution of a (4×4) input with a (3×3) kernel to form a (2×2) output in four steps. Similarly to the affine transformation of FNNs, each output element is a weighted sum of the inputs. The weights are learnable and conversely to FNNs they are shared across the input. See also Figure 2.3. Figures due to [22].

in 1980 [23]. After the popularization of backpropagation, a network with one-dimensional convolutions was first trained with this algorithm in 1988 [37] and with two-dimensional convolutions for handwritten digit recognition in 1989 [49]. The superior performance of CNNs on image classification tasks compared to alternative methods was demonstrated in a 1998 paper by LeCun et al. [50].

The discrete one-dimensional convolution is written as $\mathbf{s} = \mathbf{x} * \mathbf{k}$ for an input vector $\mathbf{x}$ and a *convolution kernel* $\mathbf{k}$. Each element of the output *feature map* $\mathbf{s}$ is computed as

$$s_i = \sum_n x_{i+n} k_n \ . \tag{2.2.14}$$

with n iterating all indices in $\mathbf{k}$. The dimension of $\mathbf{s}$ does not equal the dimension of $\mathbf{x}$. The two-dimensional convolution can be written as $\mathbf{S} = \mathbf{X} * \mathbf{K}$ with

$$S_{i,j} = \sum_n \sum_m X_{i+n,j+m} K_{m,n} \tag{2.2.15}$$

and matrix input, kernel and feature map[2] [29]. The two-dimensional convolution is illustrated graphically in Figure 2.2.

The convolution operation can be formulated as a matrix-vector multiplication. This is demonstrated here by convolution of a 3×3 input by a 2×2 kernel to form a 2×2 feature map. The input is unrolled into columns of the elements acted on by the kernel at each position. The

[2]In fact, this operation is called *cross-correlation* while the one- and two-dimensional discrete convolutions are defined as

$$s_i = \sum_n x_{i-n} k_n$$
$$S_{i,j} = \sum_n \sum_m X_{i-n,j-m} K_{m,n} \ .$$

These correspond to flipping each dimension of the kernel used in the above definition and makes the operation commutative. In practice, the kernel parameters of a CNN are learned and the commutative property is not used. Thus, the two operations are equally applicable and result in the same performance. In machine learning, the cross-correlation is often simply called convolution and used in its place. As will be done here.

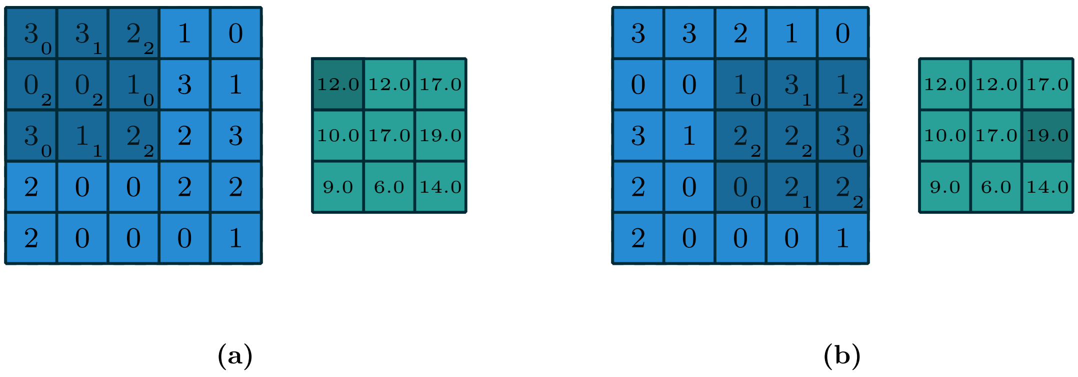


Figure 2.3: Convolution of a (5×5) input with a (3×3) kernel to form a (3×3) output exemplified by two out of a total of nine required steps. It is shown how each feature map element is a weighted sum of the input with the kernel as weights. Figures due to [22].

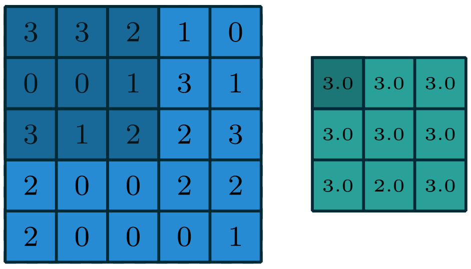
input then forms a symmetric matrix. This is multiplied by the kernel unrolled to a column vector. This can be written as,

$$
\begin{aligned}
\mathbf{S} &= \mathbf{X} * \mathbf{K} \\
&= \begin{bmatrix} X_{11} & X_{21} & X_{31} \\ X_{12} & X_{22} & X_{32} \\ X_{13} & X_{23} & X_{33} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \\
&= \begin{bmatrix} X_{11} & X_{12} & X_{21} & X_{22} \\ X_{12} & X_{13} & X_{22} & X_{23} \\ X_{21} & X_{22} & X_{31} & X_{32} \\ X_{22} & X_{23} & X_{32} & X_{33} \end{bmatrix} \begin{bmatrix} K_{11} \\ K_{12} \\ K_{21} \\ K_{22} \end{bmatrix}_{2\times 2} \\
&= \begin{bmatrix} K_{11}X_{11} + K_{12}X_{12} + K_{21}X_{21} + K_{22}X_{22} \\ K_{11}X_{12} + K_{12}X_{13} + K_{21}X_{22} + K_{22}X_{23} \\ K_{11}X_{21} + K_{12}X_{22} + K_{21}X_{31} + K_{22}X_{32} \\ K_{11}X_{22} + K_{12}X_{23} + K_{21}X_{32} + K_{22}X_{33} \end{bmatrix}_{2\times 2} .
\end{aligned}
$$

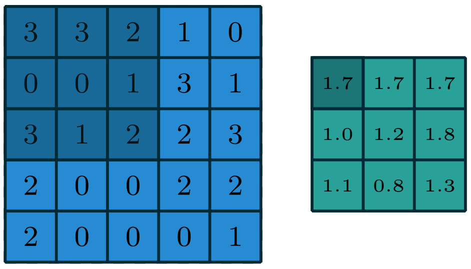
Note how each feature map element is computed as a weighted sum of the corresponding input elements with the weights defined by the kernel. This is visualized in Figure 2.3. This matrix-vector formulation easily extends to multiple kernels resulting in multiple feature maps by adding additional kernels as columns to the unrolled kernel. Efficient algorithms for computing the convolution is still an active research field [29]

The convolution operation leverages three ideas that separate it from the affine transformation [29].

1. Sparse connectivity
2. Parameter sharing
3. Equivariant representations



(a)



(b)

Figure 2.4: **(a)** Max- and **(b)** average-pooling of a (5×5) input with a (3×3) kernel to form a (3×3). Figures due to [22].

These are introduced briefly below.

The *sparse connectivity* arises from each kernel being smaller than the input. Each unit of the feature map is then a function of only part of the input proportional to the kernel size. This also gives rise to the concept of the *local receptive field* which is defined as the range of inputs which are connected to a certain feature map element. The layer that is connected to the input has a receptive field defined by its own kernel size. By stacking convolutional layers, the receptive field increases in size by indirectly connecting to a wider range of inputs through the feature maps of preceeding layers [29].

Parameter sharing is introduced by letting the kernel act on the entire input image by sliding it across it. Each element in the feature map then depends on learning a kernel that is useful for many locations in the input, This allows a CNN to exploit correlations between neighbouring elements in the input. In the case of time series or images this corresponds to exploiting temporal or spatial correlations, irrespective of their specific locations within the input. This is not possible for an FNN [29].

The convolution is *equivariant to translations* in its input in the sense that for any translation operation T applied to $\mathbf{X}$ it holds that

$$T[\mathbf{X}] * \mathbf{K} = T[\mathbf{X} * \mathbf{K}] = \mathbf{X} * T[\mathbf{K}] \tag{2.2.16}$$

i.e. applying the translation to the input (or kernel) and doing the convolution is equivalent to doing the convolution and then applying the translation to the feature map[3] [29].

Beside the input size and kernel size, some other parameters that define the convolution operation are input zero *padding*, the distance between two consecutive positions of the kernel on the input called the *stride*, and *dilation* which controls the distance between the inputs read by the kernel at a single position [22].

In order to shrink the dimension of resulting feature maps, pooling layers are often applied after convolutional layers. Generally, pooling layers summarize a neighbourhood of elements

[3]Equivariance differs from invariance. Where equivariance means that output changes in the same way as input does, invariance to a transformation means that output does not change when that transformation is applied to the input. Often, invariance can be obtained fairly easily from equivariance by disregarding the additional information about the input transformation contained in the output [29]. There has been some discussion of these properties in relation to the newly proposed capsule networks [88] as an alternative to CNNs.

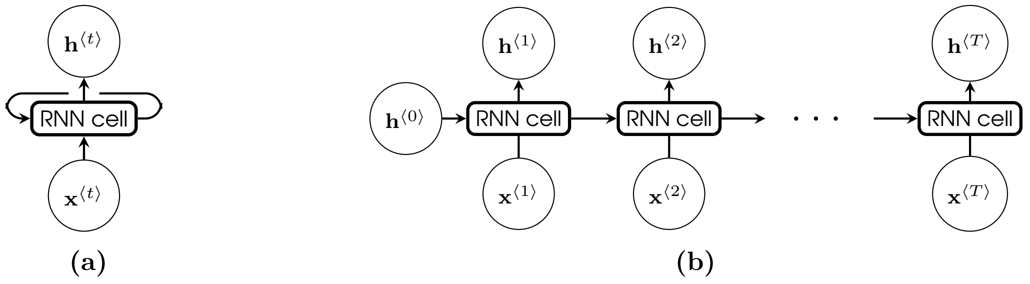

$\mathbf{h}^{\langle t \rangle}$
RNN cell
$\mathbf{x}^{\langle t \rangle}$
(a)
$\mathbf{h}^{\langle 0 \rangle}$
$\mathbf{h}^{\langle 1 \rangle}$
$\mathbf{h}^{\langle 2 \rangle}$
$\mathbf{h}^{\langle T \rangle}$
RNN cell
RNN cell
. . .
RNN cell
$\mathbf{x}^{\langle 1 \rangle}$
$\mathbf{x}^{\langle 2 \rangle}$
$\mathbf{x}^{\langle T \rangle}$
(b)

Figure 2.5: **(a)** Illustration of the feedback loop in a basic RNN. Without the feedback this would have been an ordinary FNN. **(b)** The same RNN but unrolled in time. Each recurrent cell applies the same set of operations recurrently on each input element and updated hidden state. Figures inspired by [75].

in a feature map into a single element by using some operation. The most widespread type of pooling is max-pooling [126, 81, 95]. When max-pooling is applied to a two dimensional input, each element of the resulting feature map is computed as the maximum element contained within a rectangular pooling kernel applied to the input, similarly to how convolution is computed, by sliding the kernel. Other types of pooling apply a different operation, e.g. averaging, weighted averaging according to distance from center and L_2 norm [29]. Max- and average-pooling are illustrated in Figure 2.4. These pooling operations can make convolution *invariant to translations*, such that output does not change for small translations of the input. When learning multiple kernels in a single convolutional layer and pooling over these, additional invariances, such as rotation invariance, can be learned [29].

Conventionally, CNNs consist of a series of convolutional layers each followed by a pooling layer and an activation function with the final convolutional layer feature map being flattened and input to an FNN, e.g. for classification. In [59] it was proposed to dispense with the fully connected layers by having the final convolutional layer output a feature map for each of the K classes and averaging them to form a K dimensional vector which is then fed to the softmax activation. Another attempt at making CNNs fully convolutional was made in [60].

2.2.4 Recurrent neural networks

RNNs [87] are a type of neural networks specialized for processing sequential data of variable length. Given an input sequence $\left\{\mathbf{x}^{\langle t \rangle}\right\}_{t=1}^{T} = \left\{\mathbf{x}^{\langle 1 \rangle}, \mathbf{x}^{\langle 2 \rangle}, \dots, \mathbf{x}^{\langle T \rangle}\right\}$ of length T an RNN recurrently applies the same series of operations to each sequence time step, $\mathbf{x}^{\langle t \rangle}$, starting from one end or both ends if the RNN is *bidirectional.* The power of RNNs lie in the ability to compute some hidden state which changes depending on the entire history of seen sequence vectors. This works as a kind of memory.

An RNN can be represented as a computational graph with a feedback loop as in Figure 2.5a. There, the entire sequence $\mathbf{x}$ is the input and the RNN cell outputs the hidden state $\mathbf{h}$. Such a graph can be unrolled to the length of the input sequence as seen in Figure 2.5b. It is then clear how the input sequence is read element by element while the hidden state is updated. The hidden state sequence may have a length different from the length of the input sequence. A length of 1 for the hidden state sequence is typically used for classification. There is no requirement that a hidden state must be output whenever a input is read. In fact, an RNN can

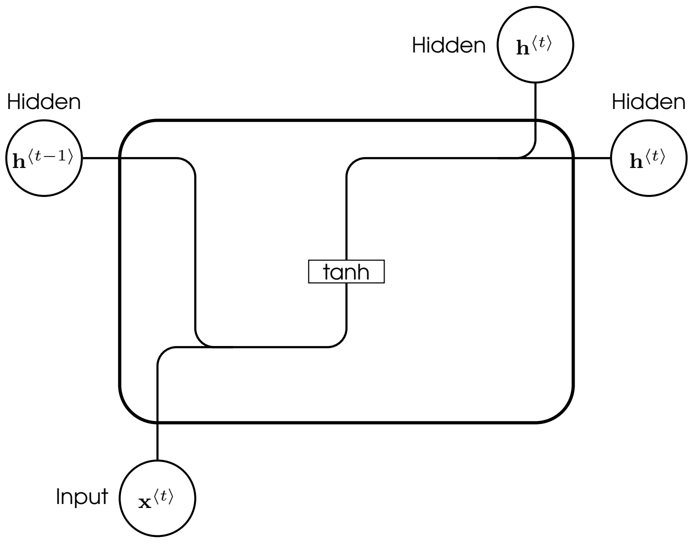

Hidden
$\mathbf{h}^{\langle t \rangle}$
Hidden
$\mathbf{h}^{\langle t-1 \rangle}$
Hidden
$\mathbf{h}^{\langle t \rangle}$
tanh
Input
$\mathbf{x}^{\langle t \rangle}$

Figure 2.6: Layout of the basic recurrent cell. Due to the small gradients of the tanh, this cell is often not able to propagate error signals far enough back in time to learn long term dependencies. Figure inspired by [75].

consist of an encoder which first reads the input sequence and a decoder which then constructs the output. The RNN cell in Figure 2.5 represents the operation applied to the input $\mathbf{x}^{\langle t \rangle}$ given the hidden state $\mathbf{h}^{\langle t-1 \rangle}$. Beside the network architecture, the operations specified by this recurrent cell are what defines the RNN. A basic RNN has a cell defined by a single fully connected layer which acts on the (concatenated) input and hidden states.

$$\mathbf{h}^{\langle t \rangle} = \tanh\left(\mathbf{W}_{hh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{W}_{hx}\mathbf{x}^{\langle t \rangle} + \mathbf{b}_h\right) \tag{2.2.17}$$

This cell is illustrated in Figure 2.6. In Figure 2.6 and the following ones, nodes represent vectors while lines and arrows illustrate the flow of these. Two lines merging denotes the concatenation of the two corresponding vectors while the splitting of a line represents the corresponding vector being copied. Square boxes represent a single layered FNN, i.e. an affine transformation of the input, and the text inside denotes the used nonlinearity. Due to the problem of *vanishing gradients* this type of RNN is not able to learn long term dependencies and is tricky to train overall. This problem arises since the gradients of the tanh activation are constrained within $(0, 1]$ and tend to zero for large and small inputs as previously described. Recurrently multiplying many small gradients results in increasingly smaller gradients the further back in time they are propagated.

A much more successful recurrent cell structure is the LSTM cell [39] illustrated in Figure 2.7. Here, small interior circular nodes represent elementwise operations by the shown operator and elliptical nodes indicate elementwise application of a function.. To avoid the ing gradients problem this cell introduces a cell state $\mathbf{c}^{\langle t \rangle}$ which passes through time without being squashed in activations. It is instead modified multiplicatively and additively by the outputs of certain gates. The gates are designed to learn which parts of the previous cell state to *forget*, which parts of the input and hidden state to *input* to the new cell state and which new hidden

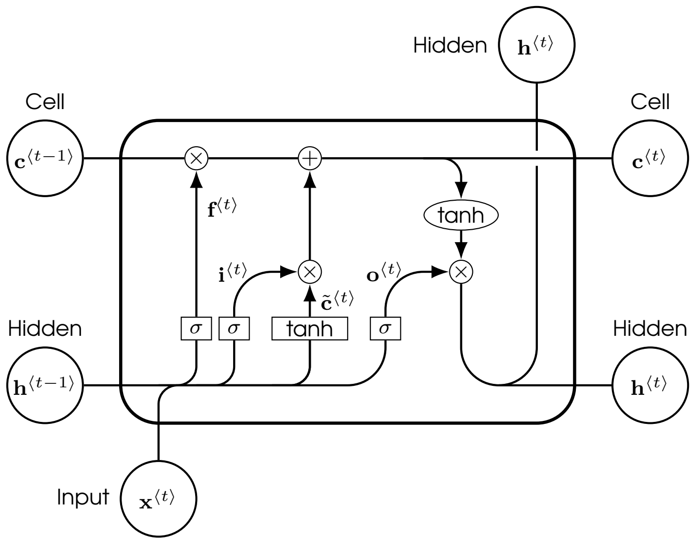

Hidden
$\mathbf{h}^{\langle t \rangle}$
Cell
$\mathbf{c}^{\langle t-1 \rangle}$
Cell
$\mathbf{c}^{\langle t \rangle}$
$\mathbf{f}^{\langle t \rangle}$
tanh
$\mathbf{i}^{\langle t \rangle}$
$\tilde{\mathbf{c}}^{\langle t \rangle}$
$\mathbf{o}^{\langle t \rangle}$
Hidden
σ
σ
tanh
σ
Hidden
$\mathbf{h}^{\langle t-1 \rangle}$
$\mathbf{h}^{\langle t \rangle}$
Input
$\mathbf{x}^{\langle t \rangle}$

Figure 2.7: Layout of the long short-term memory (LSTM) cell. The cell state $\mathbf{c}^{\langle t \rangle}$ effectively propogates gradients far backwards in time since it is never squashed in an activation function. The hidden state and input are concatenated and used to compute forget, input and output gates along with a proposed new cell state. Figure inspired by [75].

state to *output*,

$$\mathbf{f}^{\langle t \rangle} = \sigma\Big(\mathbf{W}_{fh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{W}_{fx}\mathbf{x}^{\langle t \rangle} + \mathbf{b}_f\Big) \tag{2.2.18a}$$

$$\mathbf{i}^{\langle t \rangle} = \sigma\Big(\mathbf{W}_{ih}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{W}_{ix}\mathbf{x}^{\langle t \rangle} + \mathbf{b}_i\Big) \tag{2.2.18b}$$

$$\mathbf{o}^{\langle t \rangle} = \sigma\Big(\mathbf{W}_{oh}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{W}_{ox}\mathbf{x}^{\langle t \rangle} + \mathbf{b}_o\Big) \; . \tag{2.2.18c}$$

The new cell state is computed from a proposed cell state $\tilde{\mathbf{c}}^{\langle t \rangle}$ and the forget and input gates $\mathbf{f}^{\langle t \rangle}$ and $\mathbf{i}^{\langle t \rangle}$. The new hidden state is computed from the output gate $\mathbf{o}^{\langle t \rangle}$ and the new cell state,

$$\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\Big(\mathbf{W}_{ch}\mathbf{h}^{\langle t-1 \rangle} + \mathbf{W}_{cx}\mathbf{x}^{\langle t \rangle} + \mathbf{b}_c\Big) \tag{2.2.19a}$$

$$\mathbf{c}^{\langle t \rangle} = \mathbf{f}^{\langle t \rangle} \odot \mathbf{c}\langle t-1 \rangle + \mathbf{i}^{\langle t \rangle} \odot \tilde{\mathbf{c}}^{\langle t \rangle} \tag{2.2.19b}$$

$$\mathbf{h}^{\langle t \rangle} = \mathbf{o}^{\langle t \rangle} \odot \tanh\Big(\mathbf{c}^{\langle t \rangle}\Big) \; . \tag{2.2.19c}$$

There exists a number of variations on the design of the LSTM cell. One such variation includes peephole connections [25] that allows the gates to see the cell state. This peephole LSTM is shown in Figure 2.8. Other noteworthy variations include the LSTM with coupled forget and input gates and the gated recurrent unit (GRU) [14] which simplifies the LSTM by using a joint cell and hidden state and only two gates, update and reset.

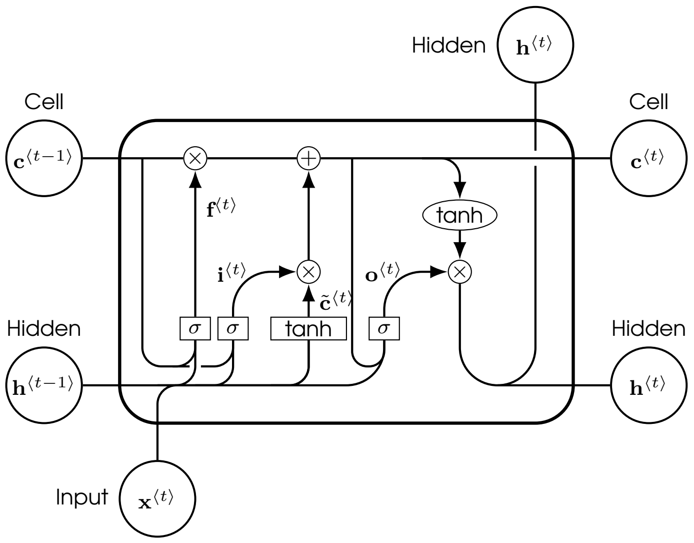

Hidden
$\mathbf{h}^{\langle t\rangle}$
Cell
$\mathbf{c}^{\langle t-1\rangle}$
Cell
$\mathbf{c}^{\langle t\rangle}$
$\mathbf{f}^{\langle t\rangle}$
tanh
$\mathbf{i}^{\langle t\rangle}$
$\mathbf{o}^{\langle t\rangle}$
$\tilde{\mathbf{c}}^{\langle t\rangle}$
Hidden
$\mathbf{h}^{\langle t-1\rangle}$
σ
σ
tanh
σ
Hidden
$\mathbf{h}^{\langle t\rangle}$
Input
$\mathbf{x}^{\langle t\rangle}$

Figure 2.8: Layout of the peephole variant of the LSTM cell. Compared to the LSTM, gates are allowed access to the cell state when computing their output. This is just one of many variants that have arisen of the LSTM. Figure inspired by [75].

2.3 Training neural networks

2.3.1 Error functions

Various error functions for different classes of problems can be derived by following the maximum likelihood approach. For a network $\mathbf{y}(\mathbf{x}, \mathbf{w})$ with parameters denoted by $\mathbf{w}$, the approach is as follows. First, the assumed distribution of targets given input $\mathbf{x}$ and network, $p(\mathbf{t}|\mathbf{x}, \mathbf{w})$, is defined. Then its likelihood $p(\mathbf{T}|\mathbf{X}, \mathbf{w})$ is formed for a batch of inputs and targets, $\mathbf{X} = \{\mathbf{x}_i \mid i \in \mathcal{B}\}$, $\mathbf{T} = \{\mathbf{t}_i \mid i \in \mathcal{B}\}$, where $\mathcal{B}$ is a batch of indices to the dataset $\mathcal{D}$. Finally the negative logarithm of the likelihood is minimized w.r.t. $\mathbf{w}$.

In deep learning it is common to derive a loss function from the log-likelihood and instead minimize that [8]. The error function should take the form of a sum over individual error terms, each of which must be a function of the network parameters [73],

$$E(\mathbf{w}) = \sum_{i\in\mathcal{B}} E_i \ . \tag{2.3.1}$$

The following will present some common error functions.

An assumed Gaussian distribution of targets with shared noise variance, $p(\mathbf{t}|\mathbf{x}, \mathbf{w}) = \mathcal{N}(\mathbf{t}|\mathbf{y}(\mathbf{x}, \mathbf{w}), \sigma^2\mathbf{I})$, results in a regression problem and the mean squared error (MSE)

$$E_{\text{MSE}}(\mathbf{w}) = \frac{1}{2}\sum_{i\in\mathcal{B}} \|\mathbf{y}_i - \mathbf{t}_i\|_2^2 \tag{2.3.2}$$

where the division by $|\mathcal{B}|$ has been neglected since it does not affect minimization. The noise variance σ^2 can be found from the regular maximum likelihood estimate (MLE).

For Bernoulli distributed targets, $t \in \{0,1\}$, the predictions are scalar, $y \in [0,1]$, and $p(t|\mathbf{x},\mathbf{w}) = y(\mathbf{x},\mathbf{w})^t(1-y(\mathbf{x},\mathbf{w}))^{1-t}$. The result is the binary cross entropy loss (CEL),

$$E_{\text{BCEL}}(\mathbf{w}) = -\sum_{i\in\mathcal{B}} t_i \log y_i + (1-t_i)\log(1-y_i) \ . \tag{2.3.3}$$

With K separate binary classifications between independent classes, the target distribution can be modelled by $p(\mathbf{t}|\mathbf{x},\mathbf{w}) = \prod_{k=1}^K y_k(\mathbf{x},\mathbf{w})^{t_k}(1-y_k(\mathbf{x},\mathbf{w}))^{1-t_k}$ where a label $t_k \in \{0,1\}$ is associated with each of the K classes. The resulting loss is the “separate” CEL

$$E_{\text{SCEL}}(\mathbf{w}) = -\sum_{i\in\mathcal{B}}\sum_{k=1}^{K} t_{ik}\log y_{ik} + (1-t_{ik})\log(1-y_{ik}) \ . \tag{2.3.4}$$

with $y_{ik} = y_k(\mathbf{x}_i,\mathbf{w})$.

If the K classes are mutually exclusive then $p(t|\mathbf{x},\mathbf{w})$ is the categorical, or multinoulli, distribution which can be written as $p(\mathbf{t}|\mathbf{x},\mathbf{w}) = \prod_{k=1}^K \mathbf{y}_k(\mathbf{x},\mathbf{w})^{t_k}$ using a one-hot encoding of the targets $\mathbf{t}$. The result is the categorical CEL

$$E_{\text{CCEL}}(\mathbf{w}) = -\sum_{i\in\mathcal{B}}\sum_{k=1}^{K} t_{ik}\log y_{ik} \ . \tag{2.3.5}$$

2.3.2 The canonical link

It should be noted that the choice of final layer activation is intimately connected to the chosen error function through the so-called *canonical link function.* When combining an error function with the corresponding canonical link function as activation, the gradient of a single contribution to the error w.r.t. the output layer hidden units $\mathbf{z}^{[L]}$ takes the form of a signed error,

$$\frac{\partial E_i}{\partial \mathbf{z}_i^{[L]}} = \frac{\partial E_i}{\partial \mathbf{a}_i^{[L]}}\frac{\partial \mathbf{a}_i^{[L]}}{\partial \mathbf{z}_i^{[L]}} = \mathbf{y}_i - \mathbf{t}_i \ . \tag{2.3.6}$$

The canonical activation for the MSE loss is the identity function, i.e. the output units are simply linear. This can easily be seen by setting $\mathbf{a}^{[L]} = \mathbf{z}^{[L]}$. Then

$$\frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}} = \mathbf{I}$$

and by definition of $\mathbf{y}$ and the MSE,

$$\begin{aligned}\frac{\partial E_i}{\partial \mathbf{z}^{[L]}} &= \frac{\partial E_i}{\partial \mathbf{a}^{[L]}}\\ &= \frac{\partial E_i}{\partial \mathbf{y}}\\ &= \frac{\partial}{\partial \mathbf{y}}\|\mathbf{y}-\mathbf{t}\|_2^2\\ &= \mathbf{y}-\mathbf{t} \ ,\end{aligned}$$

ignoring the i subscript on $\mathbf{y},\mathbf{t},\mathbf{z}^{[L]}$ and $\mathbf{a}^{[L]}$ for simplicity. For the binary and K class separate binary CEL the canonical activation is the sigmoid while for the multiclass CEL it is the softmax [8]. These relation won’t be derived here. When applying the canonical link for the error functions above, the loss is also sometimes referred to as the negative log-likelihood (NLL) loss.

2.3.3 Backpropagation

The predominant method for training of neural networks is the *backpropagation* algorithm. Much as for the perceptron, an error function is defined and the network is optimized to minimize this error. The gradient of the error function w.r.t. all learnable parameters of the network is computed and used to adjust these in the direction that minimizes error.

The backpropagation algorithm is at its core a serial application of the chain rule of calculus. As such it requires a differentiable network model in that the applied transformations and nonlinearities must be differentiable. It also requires a differentiable error function as discussed above [73]. A model satisfying these requirements is sometimes called *end-to-end differentiable*.

2.3.3.1 Backpropagation in feedforward neural networks

Consider an FNN. By the chain rule, the gradient of the loss w.r.t. to the hidden units of any layer l can be written as

$$\frac{\partial E_i}{\partial \mathbf{z}^{[l]}} = \underbrace{\underbrace{\underbrace{\frac{\partial E_i}{\partial \mathbf{a}^{[L]}} \frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}}}_{\boldsymbol{\delta}^{[L]}} \frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{a}^{[L-1]}} \frac{\partial \mathbf{a}^{[L-1]]}}{\partial \mathbf{z}^{[L-1]}}}_{\boldsymbol{\delta}^{[L-1]}} \cdots \frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}}_{\boldsymbol{\delta}^{[l]}} \tag{2.3.7}$$

where the accumulated *error signals*, $\boldsymbol{\delta}^{[l]}$, have been defined as shown. These are useful for making notation more compact and illustrating the symmetry of backpropagation. One should note that the derivative of a scalar by a vector is a vector (gradient), the derivative of a vector by a vector is a matrix (Jacobian) and the derivative of a scalar by a matrix is a matrix. It is important to be mindful of dimensions while taking matrix and vector derivatives. Except for the output layer the chain rule can be applied sequentially all through the FNN by repeating the

$$\cdots \frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}} \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} \cdots$$

pattern. The gradient w.r.t. any learnable parameter can then be found by finally appending the derivative of the appropriate hidden unit with respect to that parameter[4].

One can note that $\boldsymbol{\delta}^{[L]}$ will equal $\mathbf{y} - \mathbf{t}$ in case the canonical link activation is used. One can also note that the $\frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}}$ terms are derivatives of the l'th layer affine transformation w.r.t. its input and that the $\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}$ terms are the derivatives of the l'th layer activation function w.r.t. its inputs.

For any activation applied elementwise to an H_l-dimensional hidden unit, the $\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}$ term

[4]This is of course assuming that the learnable parameter resides in the hidden unit transformation. If for instance the activation function has some learnable parameter, the appropriate derivative of $\mathbf{a}^{[l]}$ is simply appended the $\frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}}$ term instead.

can be seen to be a diagonal matrix. This can be computed as follows.

$$\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \frac{\partial}{\partial \mathbf{z}^{[l]}}\left[\varphi_l\left(\mathbf{z}^{[l]}\right)\right] = \frac{\partial}{\partial \mathbf{z}^{[l]}}\begin{bmatrix}\varphi_l\left(z_1^{[l]}\right)\\ \varphi_l\left(z_2^{[l]}\right)\\ \vdots \\ \varphi_l\left(z_{H_l}^{[l]}\right)\end{bmatrix}$$

which is the Jacobian matrix,

$$\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \begin{bmatrix}\frac{\partial \varphi_l\left(z_1^{[l]}\right)}{\partial z_1^{[l]}} & \cdots & \frac{\partial \varphi_l\left(z_1^{[l]}\right)}{\partial z_H^{[l]}}\\ \vdots & \ddots & \vdots \\ \frac{\partial \varphi_l\left(z_{H_l}^{[l]}\right)}{\partial z_1^{[l]}} & \cdots & \frac{\partial \varphi_l\left(z_{H_l}^{[l]}\right)}{\partial z_{H_l}^{[l]}}\end{bmatrix},$$

which reduces to

$$\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} = \begin{bmatrix}\varphi_l'\left(z_1^{[l]}\right) & \cdots & 0\\ \vdots & \ddots & \vdots \\ 0 & \cdots & \varphi_l'\left(z_{H_l}^{[l]}\right)\end{bmatrix} = \operatorname{diag}\left(\varphi_l'\left(\mathbf{z}^{[l]}\right)\right) . \tag{2.3.8}$$

Thus, the derivative of the activation is applied elementwise to the hidden unit and arranged in a diagonal matrix.

Since multiplication of a vector by a diagonal matrix effectively multiplies each element of the vector by the corresponding diagonal element, the output layer error signal can be written as

$$\begin{aligned}\boldsymbol{\delta}^{[L]} &= \frac{\partial E_i}{\partial \mathbf{a}^{[L]}}\frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}}\\ &= \frac{\partial E_i}{\partial \mathbf{a}^{[L]}}\operatorname{diag}\left(\varphi_L'\left(\mathbf{z}^{[L]}\right)\right)\\ &= \frac{\partial E_i}{\partial \mathbf{a}^{[L]}} \odot \varphi_L'\left(\mathbf{z}^{[L]}\right)\end{aligned} \tag{2.3.9}$$

where $\odot$ denotes the Hadamard product and $\frac{\partial E_i}{\partial \mathbf{a}^{[L]}}$ could equivalently be written as $\nabla_{\mathbf{a}^{[L]}} E_i$ since it is the gradient of E_i. As activations are computed elementwise from hidden units, there are as many elements in the gradient of E_i w.r.t $\mathbf{a}^{[L]}$ as there are in $\varphi_L'\left(\mathbf{z}^{[L]}\right)$ and the dimensions match in the elementwise product. The result for the canonical link still holds and $\boldsymbol{\delta}^{[L]}$ simplifies to $\mathbf{y} - \mathbf{t}$ for such an output activation.

Using the forward propagation equations for a single FNN layer (2.2.7) and the previous result for the derivative of an elementwise applied activation function (2.3.8), the error signal of the l'th layer, $\boldsymbol{\delta}^{[l]}$ can be written in terms of the error signal of the next layer, $\boldsymbol{\delta}^{[l+1]}$. By (2.3.7),

$$\begin{aligned}\boldsymbol{\delta}^{[l]} &= \boldsymbol{\delta}^{[l+1]}\frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}}\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}\\ &= \boldsymbol{\delta}^{[l+1]}\frac{\partial}{\partial \mathbf{a}^{[l]}}\left[\mathbf{W}^{[l+1]}\mathbf{a}^{[l]} + \mathbf{b}^{[l+1]}\right] \odot \varphi_l'\left(\mathbf{z}^{[l]}\right)\\ &= \mathbf{W}^{[l+1]^{\mathrm{T}}}\boldsymbol{\delta}^{[l+1]} \odot \varphi_l'\left(\mathbf{z}^{[l]}\right) , \quad l \in [1, L-1] .\end{aligned} \tag{2.3.10}$$

This equation gives the means to efficiently backpropagate error signals through the computational graph of an NN. The error signal into layer l has the same dimensionality as the number of hidden units in that layer. Therefore, the matrix-vector product $\mathbf{W}^{[l+1]^{\mathrm{T}}}\boldsymbol{\delta}^{[l+1]}$ will have valid dimensions. The following elementwise multiplication by $\varphi_l'\big(\mathbf{z}^{[l]}\big)$ matches the number of rows in $\mathbf{W}^{[l+1]^{\mathrm{T}}}$ since $\mathbf{z}^{[l]}$ has dimensionality of the number of columns in $\mathbf{W}^{[l+1]}$ by (2.2.7).

For the weight of the l'th layer, the gradient is computed by (2.3.7) as follows,

$$\begin{aligned}\frac{\partial E_i}{\partial \mathbf{W}^{[l]}} &= \overbrace{\frac{\partial E}{\partial \mathbf{a}^{[L]}}\frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}}\frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{a}^{[L-1]}}\frac{\partial \mathbf{a}^{[L-1]]}}{\partial \mathbf{z}^{[L-1]}}\cdots\frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}}\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}}^{\boldsymbol{\delta}^{[l]}}\frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{W}^{[l]}} \\ &= \boldsymbol{\delta}^{[l]}\frac{\partial}{\partial \mathbf{W}^{[l]}}\Big[\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}\Big] \\ &= \boldsymbol{\delta}^{[l]}\mathbf{a}^{[l-1]^{\mathrm{T}}} \ ,\end{aligned} \tag{2.3.11}$$

and for the associated bias,

$$\begin{aligned}\frac{\partial E_i}{\partial \mathbf{b}^{[l]}} &= \overbrace{\frac{\partial E_i}{\partial \mathbf{a}^{[L]}}\frac{\partial \mathbf{a}^{[L]}}{\partial \mathbf{z}^{[L]}}\frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{a}^{[L-1]}}\frac{\partial \mathbf{a}^{[L-1]]}}{\partial \mathbf{z}^{[L-1]}}\cdots\frac{\partial \mathbf{z}^{[l+1]}}{\partial \mathbf{a}^{[l]}}\frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}}}^{\boldsymbol{\delta}^{[l]}}\frac{\partial \mathbf{z}^{[l]}}{\partial \mathbf{b}^{[l]}} \\ &= \boldsymbol{\delta}^{[l]}\frac{\partial}{\partial \mathbf{b}^{[l]}}\Big[\mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}\Big] \\ &= \boldsymbol{\delta}^{[l]} \ .\end{aligned} \tag{2.3.12}$$

Since $\boldsymbol{\delta}^{[l]}$ has dimensions of the l'th hidden layer and $\mathbf{a}^{[l-1]}$ has dimensions of the $l-1$'th hidden layer, the outer product $\boldsymbol{\delta}^{[l]}\mathbf{a}^{[l-1]^{\mathrm{T}}}$ results in a matrix with dimensionality of the l'th hidden layer in its rows and the $l-1$'th in its columns. This is exactly the dimensionality of $\mathbf{W}^{[l]}$. Since $\boldsymbol{\delta}^{[l]}$ has dimensionality of the l'th hidden layer, the gradient of the bias matches the dimension of the bias parameter of the l'th layer.

Together, (2.3.9)-(2.3.12) define backpropagation of the error signal associated with a single input $\mathbf{x} = \mathbf{a}^{[0]}$. Since the total error associated with a batch $\mathcal{B}$ of examples is given as a sum of the errors associated with each example (2.3.1), it holds for its derivative that

$$\frac{\partial E}{\partial \mathbf{W}^{[l]}} = \sum_{i\in\mathcal{B}}\frac{\partial E_i}{\partial \mathbf{W}^{[l]}} \tag{2.3.13}$$

and likewise for a derivative w.r.t. any other variable. Thus, the total gradient of any parameter is simply obtained by summing the contributions to it from individual training examples [8].

This batched approach can be efficiently implemented by concatenation of multiple examples into the rows (or columns) of an input matrix,

$$\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_{\mathcal{B}}\end{bmatrix}^{\mathrm{T}} \ . \tag{2.3.14}$$

Then, activations and hidden units become matrices as well

$$\begin{aligned}\mathbf{Z}^{[l]} &= \mathbf{W}^{[l]}\mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} = \begin{bmatrix}\mathbf{z}_1^{[l]} & \mathbf{z}_2^{[l]} & \cdots & \mathbf{z}_{\mathcal{B}}^{[l]}\end{bmatrix}^{\mathrm{T}} \\ \mathbf{A}^{[l]} &= \varphi_l\Big(\mathbf{Z}^{[l]}\Big) = \begin{bmatrix}\mathbf{a}_1^{[l]} & \mathbf{a}_2^{[l]} & \cdots & \mathbf{a}_{\mathcal{B}}^{[l]}\end{bmatrix}^{\mathrm{T}}\end{aligned} \tag{2.3.15}$$

while the weights and biases retain the same dimensions. This method allows efficient forward propagation of an entire batch of examples through the network. The loss and gradients are computed as already described.

2.3.3.2 A two layer feedforward neural network

Here, the training process of neural networks is illustrated by derivation of the backpropagation equations for a two layer FNN for regression using the MSE loss. The network architecture is as the one in Figure 2.1b The final layer nonlinearity will be the identity function according to the canonical link while the first nonlinearity is the ReLU. The forward pass of this network is

$$\begin{aligned}
\mathbf{z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \\
\mathbf{a}^{[1]} &= \mathrm{ReLU}\Big(\mathbf{z}^{[1]}\Big) \\
\mathbf{z}^{[2]} &= \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \\
\mathbf{y} = \mathbf{a}^{[2]} &= \mathbf{z}^{[2]} \ .
\end{aligned} \tag{2.3.16}$$

Due to the canonical link, $\boldsymbol{\delta}^{[2]} = \mathbf{y} - \mathbf{t}$. By (2.3.11), the gradients for the output layer are

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{W}^{[2]}} &= \boldsymbol{\delta}^{[2]} \mathbf{a}^{[1]^{\mathrm{T}}} \\
&= (\mathbf{y} - \mathbf{t}) \odot \mathrm{ReLU}\Big(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}\Big)^{\mathrm{T}} && (2.3.17) \\
\frac{\partial E}{\partial \mathbf{b}^{[2]}} &= \boldsymbol{\delta}^{[2]} \\
&= (\mathbf{y} - \mathbf{t}) \ . && (2.3.18)
\end{aligned}$$

With the definition in (2.3.10),

$$\begin{aligned}
\frac{\partial E}{\partial \mathbf{W}^{[1]}} &= \boldsymbol{\delta}^{[1]} \mathbf{a}^{[0]^{\mathrm{T}}} \\
&= \boldsymbol{\delta}^{[1]} \mathbf{x}^{\mathrm{T}} \\
&= \mathbf{W}^{[2]^{\mathrm{T}}} \boldsymbol{\delta}^{[2]} \odot \mathrm{ReLU}'\Big(\mathbf{z}^{[1]}\Big)\mathbf{x}^{\mathrm{T}} \\
&= \mathbf{W}^{[2]^{\mathrm{T}}} (\mathbf{y} - \mathbf{t}) \odot \mathrm{ReLU}'\Big(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}\Big)\mathbf{x}^{\mathrm{T}} && (2.3.19) \\
\frac{\partial E}{\partial \mathbf{b}^{[1]}} &= \boldsymbol{\delta}^{[1]} \\
&= \mathbf{W}^{[2]^{\mathrm{T}}} (\mathbf{y} - \mathbf{t}) \odot \mathrm{ReLU}'\Big(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}\Big) \ . && (2.3.20)
\end{aligned}$$

In case of ReLU activations

$$\begin{aligned}
\varphi'(z_k) &= \frac{\partial}{\partial z_k}[\max\{0, z_k\}] && (2.3.21) \\
&= \begin{cases} 1 & \text{if } z_k > 0 \\ 0 & \text{if } z_k \leq 0 \end{cases} \ . && (2.3.22)
\end{aligned}$$

Gradients are then backpropagated only for units with positive activations. In vectorized form $\varphi'(\mathbf{z}) = \frac{\partial}{\partial \mathbf{z}}[\max\{0, \mathbf{z}\}]$ and

$$\varphi'(\mathbf{z}) = \mathbf{G} \ , \quad G_{k,k} = \begin{cases} 1 & \text{if } z_k > 0 \\ 0 & \text{if } z_k \leq 0 \end{cases} \tag{2.3.23}$$

in line with the general derivation which showed that the gradient of an elementwise activation is a diagonal matrix.

2.3.3.3 Backpropagation in convolutional and recurrent networks

The principles used for FNNs above can also be applied to training convolutional and recurrent networks. Returning to the example used for discussing vectorization of CNNs, the gradient w.r.t. $\mathbf{K}$ is

$$\frac{\partial E_i}{\partial K_{jk}} = \frac{\partial E_i}{\partial S_{11}}\frac{\partial S_{11}}{\partial K_{jk}} + \frac{\partial E_i}{\partial S_{12}}\frac{\partial S_{12}}{\partial K_{jk}} + \frac{\partial E_i}{\partial S_{21}}\frac{\partial S_{21}}{\partial K_{jk}} + \frac{\partial E_i}{\partial S_{22}}\frac{\partial S_{22}}{\partial K_{jk}} \tag{2.3.24}$$

where only a single layer is considered. This reduces to

$$\begin{aligned}
\frac{\partial E_i}{\partial K_{11}} &= \frac{\partial E_i}{\partial S_{11}}X_{11} + \frac{\partial E_i}{\partial S_{12}}X_{12} + \frac{\partial E_i}{\partial S_{21}}X_{21} + \frac{\partial E_i}{\partial S_{22}}X_{22}\\
\frac{\partial E_i}{\partial K_{12}} &= \frac{\partial E_i}{\partial S_{11}}X_{12} + \frac{\partial E_i}{\partial S_{12}}X_{13} + \frac{\partial E_i}{\partial S_{21}}X_{22} + \frac{\partial E_i}{\partial S_{22}}X_{23}\\
\frac{\partial E_i}{\partial K_{21}} &= \frac{\partial E_i}{\partial S_{11}}X_{21} + \frac{\partial E_i}{\partial S_{12}}X_{22} + \frac{\partial E_i}{\partial S_{21}}X_{31} + \frac{\partial E_i}{\partial S_{22}}X_{32}\\
\frac{\partial E_i}{\partial K_{22}} &= \frac{\partial E_i}{\partial S_{11}}X_{22} + \frac{\partial E_i}{\partial S_{12}}X_{23} + \frac{\partial E_i}{\partial S_{21}}X_{32} + \frac{\partial E_i}{\partial S_{22}}X_{33}
\end{aligned} \tag{2.3.25}$$

which follows the same pattern as the convolution operation and is in fact simply the convolution of the input example $\mathbf{X}$ with the backpropagated error signals $\frac{\partial E_i}{\partial S_{jk}}$. A similar result holds for the gradient w.r.t. $\mathbf{X}$. For RNNs, backpropagation for FNNs is applied throughout the unrolled computational graph and is often called *backopragation through time* [29].

2.3.3.4 Implementation of neural network framework

The modular architecture of FNNs, CNNs and RNNs makes for a handy abstraction when implementing NN models in practice. In Python, any layer can be defined as a class with learnable parameters as attributes and `forward` and `backward` methods for propagation of respectively data and gradients through the layer.

For instance, an FNN layer can be defined as a class with e.g. `weight` and `bias` attributes. The `weight` and `bias` can be implemented as instances of a simple class which holds both the actual parameter and potentially its gradient in `data` and `grad` NumPy[5] array attributes. The `forward` method takes the previous activations as input and outputs the hidden unit.

```python
def forward(self, x):
    self.x = x
    z = np.dot(x, self.weight.data) + self.bias.data
    return z
```

The `backward` method then takes the error signal from the next layer, $\boldsymbol{\delta}^{[l+1]}$, as input and computes the gradients associated with the `weight` and `bias` attributes before computing and outputting $\boldsymbol{\delta}^{[l]}$.

```python
def backward(self, delta):
    self.weight.grad = np.dot(self.x.T, delta)
    dx = np.dot(delta, self.weight.data.T)
    self.bias.grad = delta.sum(axis=0)
    return dx
```

[5]`http://www.numpy.org/`

This recipe can be used to implement also the nonlinearities as well as other types of network layers.

As part of this thesis, a small toolbox has been implemented on the side[6]. This implementation includes sigmoid, tanh, ReLU and softplus activations along with the MSE and categorical CEL. The affine and convolutional layers are implemented along with batch normalization and dropout which will be introduced later. Finally, the SGD optimizer has been implemented with L^1 and L^2 regularizations as well as regular and Nesterov momentum. Structurally, the implementation is inspired by the PyTorch deep learning framework [78]. Although the implementation has not been used for the experimental part of the thesis, it served as a base for theoretical reasoning.

2.3.4 Optimization algorithms

This section introduces some optimization techniques commonly used for gradient descent optimization of neural networks. Throughout this section, the parameters being optimized will be denoted by $\mathbf{w}$.

2.3.4.1 Stochastic gradient descent with momentum

The most basic version of gradient based optimization is gradient descent,

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}) \ . \tag{2.3.26}$$

In deep learning, $\nabla_{\mathbf{w}} E$ is most often computed on a subset of the training set. This introduces additional noise in the gradient estimate compared to using the full training set, hence "stochastic".

Momentum [80] can be added to the gradient in order to reduce the jerky motion of following a noisy gradient and to improve the performance of gradient descent in ravines. Momentum is controlled by a forgetting factor γ which controls the amount of previous gradient information that is included,

$$\begin{aligned} \mathbf{v} &\leftarrow \gamma \mathbf{v} + \eta \nabla_{\mathbf{w}} E(\mathbf{w}) \\ \mathbf{w} &\leftarrow \mathbf{w} - \mathbf{v} \ . \end{aligned} \tag{2.3.27}$$

Typically, γ is around 0.9. Momentum can significantly improve the performance of gradient based optimization of NNs.

Nesterov accelerated momentum [71, 111] improves on regular momentum by including a sense of future direction into the momentum.

$$\begin{aligned} \mathbf{v} &\leftarrow \gamma \mathbf{v} + \eta \nabla_{\mathbf{w}} E(\mathbf{w} - \gamma \mathbf{v}) \\ \mathbf{w} &\leftarrow \mathbf{w} - \mathbf{v} \ . \end{aligned} \tag{2.3.28}$$

By computing the gradient at $\mathbf{w} - \gamma \mathbf{v}$, the algorithm is effectively looking ahead and computing the gradient at the approximate future location rather than the current. Nesterov momentum has significantly improved optimization of RNNs [5].

[6] `https://github.com/JakobHavtorn/nn`

2.3.4.2 Adam

The Adam optimization algorithm [45] relies on adaptive estimation of the first and second momentum of the gradient, i.e. its mean $\mathbf{m}$ and variance $\mathbf{v}$, in every direction. At iteration t of the optimization procedure,

$$\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla_{\mathbf{w}} E(\mathbf{w}_{t-1}) \tag{2.3.29a}$$

$$\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\nabla_{\mathbf{w}} E(\mathbf{w}_{t-1})^2 \ . \tag{2.3.29b}$$

Incidentally, when initialized as zero vectors, these moment estimates become biased. Bias corrected versions are therefore computed,

$$\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t} \tag{2.3.30a}$$

$$\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta_2^t} \ , \tag{2.3.30b}$$

and the parameters are updated,

$$\mathbf{w}_t \leftarrow \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \boldsymbol{\epsilon}}\hat{\mathbf{m}}_t \ . \tag{2.3.31}$$

Intuitively, Adam behaves like a heavy ball with friction [36] so as to take restrained steps at each iteration and not overshoot. The proposed default hyperparameter values are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\boldsymbol{\epsilon} = 10^{-8}$.

Other optimization algorithms include Adagrad [21], Adadelta [123], RMSProp, AdaMax [45], Nadam [19] and AMSgrad [82].

2.3.5 Optimization and regularization techniques

Different techniques exist to prevent NNs from overfitting to which they can be prone due to their many parameters and high capacity. This section briefly considers the two primary forms of weight norm regularization and dropout. Although primarily used to improve network training, batch normalization can also have a regularizing effect and is also considered here.

2.3.5.1 Parameter norm penalties

Parameter norm regularization introduces an additional loss term $\Omega(\mathbf{w})$ to the cost function and a regularization hyperparameter $\alpha \in [0, \infty)$,

$$E(\mathbf{w}, \alpha) = E(\mathbf{w}) + \alpha\Omega(\mathbf{w}) \ . \tag{2.3.32}$$

This regularization can then be written as an additive extra term to the gradient, $\nabla_{\mathbf{w}} E(\mathbf{w}, \alpha) = \nabla_{\mathbf{w}} E(\mathbf{w}) + \alpha\nabla_{\mathbf{w}}\Omega(\mathbf{w})$.

L^1 norm regularization uses

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1^2 = \mathbf{e}^{\mathrm{T}}\mathbf{w} \tag{2.3.33}$$

where $\mathbf{e} = \begin{bmatrix}1 & 1 & \dots & 1\end{bmatrix}^{\mathrm{T}}$. The extra term on the gradient is then

$$\nabla_{\mathbf{w}}\Omega(\mathbf{w}) = \mathrm{sgn}(\mathbf{w}) \tag{2.3.34}$$

which is constant no matter the size of the individual weight giving sparse parameter vectors [29].

L^2 norm regularization has

$$\Omega(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2 = \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{2.3.35}$$

with gradient

$$\nabla_{\mathbf{w}}\Omega(\mathbf{w}) = \mathbf{w} \; . \tag{2.3.36}$$

This nudges the parameters towards zero at each update depending on their size with larger weights being regularized more than smaller weights.

Often *weight decay* is used to be synonymous with L^2 regularization. In weight decay regularization, weights are set to decay exponentially as

$$\mathbf{w} \leftarrow (1-\alpha)\mathbf{w} \tag{2.3.37}$$

with rate α at each iteration. This can equivalently be computed by adding the term in (2.3.36) to the gradient [32] and is thus identical to L^2 regularization. However, this has recently been shown to only be the case for SGD without momentum whereas for adaptive methods such as Adam, L^2 regularization differs from weight-decay and can lead to poor regularization [61]. It is instead proposed to revert implementations to the formulation of weight-decay in (2.3.37) which behaves as expected also for adaptive methods [61] and decouples the learning rate and regularization hyperparameters. In this thesis, PyTorch is used which implements the L^2 norm penalty as in (2.3.36).

2.3.5.2 Batch normalization

Batch normalization [41] is an adaptive reparameterization technique that can significantly speed up training of deep NN architectures. For a mini-batch of hidden units $\mathbf{Z}$ as defined in (2.3.15) batch normalization applies the following transformation

$$\hat{\mathbf{Z}} = \frac{\mathbf{Z} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{2.3.38}$$

where

$$\boldsymbol{\mu} = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{Z}_{:,i} \tag{2.3.39}$$

and

$$\boldsymbol{\sigma} = \sqrt{\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} (\mathbf{Z}-\boldsymbol{\mu})^2_{:,i}} \tag{2.3.40}$$

are column vectors containing the mean and variance of each activation computed across the batch of examples. These are often computed including a momentum term from previous batches. In (2.3.38), the subtraction and division of $\mathbf{Z}$ by column vectors is done by broadcasting the operation to every column in $\mathbf{Z}$ and applying it element-wise. That is $\mathbf{Z}_{i,j}$ is normalized by subtraction of μ_i and division by σ_i for all $j \in \mathcal{B}$ [7]. Often, learnable parameters $\boldsymbol{\gamma}$ and

[7] $\mathbf{Z}$ is $H \times |\mathcal{B}|$ and $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are $H \times 1$ for a layer with H hidden units so there is one row in $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ for each row in $\mathbf{Z}$

$\boldsymbol{\beta}$ are included in the batch normalization to enable learning a new mean and variance of the hidden unit,

$$\mathbf{Z}_{\mathrm{BN}} = \boldsymbol{\gamma} \odot \hat{\mathbf{Z}} + \boldsymbol{\beta} \ . \tag{2.3.41}$$

Again, the operations are broadcast to every column in $\hat{\mathbf{Z}}$. In this way, batch normalization allows learning a useful mean and variance for the suceeding layer by adapting $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$. This has much easier learning dynamics than without batch normalization where the mean and variance depend nonlinearly on the weights and biases of all preceding layers. Batch normalization is typically applied to the hidden unit before the activation function but can alternatively be applied after the activations as well. This remains a topic of discussion [29].

An alternative to batch normalization is weight normalization [89] based on a reparameterization instead of mini-batch running averages and variances,

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|}\mathbf{v} \ . \tag{2.3.42}$$

The scale and direction of $\mathbf{w}$ are decoupled into g and $\mathbf{v}$ which are then learned instead. This leads to faster convergence similarly to batch normalization. Unlike with batch normalization, the reparameterization above is independent of the mini-batch size and thus causes only minimal computational overhead but the mean of hidden units or activations over batches are nonzero.

2.3.5.3 Dropout

Dropout [38, 105] is modern regularization technique for deep NNs. It regularizes networks by setting the activation of non-output units in the network to zero with probability p. Dropout can also be applied to input units, often with a lower p [29].

Dropout can be seen as an efficient way of training an ensemble of networks. When using dropout, all sub-networks that can be obtained from the original network by dropping any number of non-output units are in effect being trained simultaneously in an interweaved manner. In fact, contrary to ordinary model averaging, the models of the dropout ensemble also share parameters [29]. In order to make a prediction, it turns out that votes from the ensemble models can be efficiently collected in a single forward pass through the original model without dropout. This is a good estimate of the ensemble in practice [38].

Dropout has had most of its success when applied to FNNs since these are most prone to overfitting. However, some results indicate that it can potentially improve performance when applied to convolutional layers using a lower dropout rate [77].

When batch normalization and dropout are combined, it can be difficult to harness the benefits of both [56]. Hence, these won't be combined in this thesis.

2.3.6 Initialization schemes

Before training any NN model it must first have its parameters initialized to some values. Parameter initialization has a strong influence on the result of the following training but methods are to a great extent heuristic and the subject remains an active field of study [29].

One important point to note about the initial values of NN parameters is that they must be chosen to *break symmetry*. An NN with all parameters initialized to e.g. the same value will see all its parameters receive the same gradient if the loss function is deterministic and the same update if a deterministic optimization algorithm is used. Since searching for initial parameters that complement each other well, e.g. orthogonal parameters, can be expensive, random initialization is often used [29].

One commonly applied form of random initialization is *Glorot-initialization* [27]. In order to have approximately the same variance in the activations of all layers as well as in the gradients, the weights of a linear FNN can be initialized as

$$W_{i,j}^{[l]} \sim \mathcal{U}\left(-\sqrt{\frac{6}{H_l + H_{l-1}}}, \sqrt{\frac{6}{H_l + H_{l-1}}}\right) \tag{2.3.43a}$$

or

$$W_{i,j}^{[l]} \sim \mathcal{N}\left(0, \frac{2}{H_l + H_{l-1}}\right) \tag{2.3.43b}$$

where H_l and H_{l-1} are the number of hidden units in layers l and $l-1$ and $\mathcal{U}(a, b)$ denotes the continuous uniform distribution on the interval $[a, b]$. The normal distribution version arises by noting that $\mathrm{Var}[\mathcal{U}(-a, a)] = \frac{1}{3}a^2$ such that σ^2 must equal $\frac{1}{3}a^2$ for the variances to be equal. This also works well in practice for FNNs with nonlinear activation functions although a correction can be made based on the nonlinearity [35]. For instance, the ReLU nonlinearity has zero output for expectedly half of its input. To maintain a equal input and output variances a gain of 2 is then multiplied on the variance of the initial weight distribution. In this thesis, Glorot intialization is applied to both fully connected and convolutional layers with this scaling.

As an alternative to random initialization, *transfer learning* can be used. In transfer learning, the parameters from another NN are used as initial values for the network. The other NN can have the same or a different architecture but must have been trained either unsupervised on the same data or supervised on a related (or even unrelated) task [29].

CHAPTER 3 Variational Optimization

"Now the learning process may be regarded as a search for a form of behaviour which will satisfy the teacher (or some other criterion). Since there is probably a very large number of satisfactory solutions the random method seems to be better than the systematic. It should be noticed that it is used in the analogous process of evolution."

- Turing, Alan (1950) [115]

3.1 Introduction

To train an NN, some error function is defined and optimized. Whenever this error function and the network itself are differentiable, it is beneficial to include information from the error gradient during optimization e.g. by the backpropagation algorithm. As briefly discussed in Chapter 1 however, some problems do not lend themselves to differentiable objective functions for various reasons. Some problems have discrete or random elements, e.g. in the model, while others don't easily or explicitly define an objective function such as RL.

Two notable successes of DRL, learning to play Atari games from pixels [68] and learning expert level Go [103], serve as examples that use the currently most popular methods for solving RL problems. These all rely on the MDP formalism and the use of value functions. They generally attempt to simplify and model the environment to a point where a differentiable error function can be defined and a value function, which encodes information about the value of every possible action given a state, can be learned by backpropagation of these errors.

A completely different approach to DRL is to instead use black-box optimization algorithms for directly learning a policy. When applied to NNs, this approach has been called *direct policy search* [97] and in some versions *neuroevolution* [84]. The ES presented recently by OpenAI in [90] is one such method.

For the purpose of introduction, Section 3.2 derives the stochastic gradient estimator presented in [90] by Taylor expansion of the objective. It is described how the estimator can be implemented in practice by a Monte Carlo approximation and its bias and variance are evaluated. A second order estimator is then derived and an interpretation of the estimators as sample covariances is given. Finally, the estimators are generalized to the multivariate case. Section 3.3 introduces variational optimization (VO) which is a general framework for black-box optimization applicable to almost any function including the intangible reward function of RL. Section 3.4, 3.5 and 3.6 consider various methods for improving the VO gradient estimators including the natural gradient, fitness transformation, inclusion of gradient information and methods for variance reduction.

3.2 Derivation of ES gradient by Taylor expansion

The Taylor series expansion is usually used to translate derivative information about a function at a certain point into information about the output of that function near that point. However, it can also be used for the inverse translation. That is, information about the output of a function at a set of points can be translated into information about the gradient of the function at the center value of those points.

A simple way to obtain the ES gradient estimate presented in [90] is through a Taylor expansion of the objective function. Here, this derivation is made to form a simple introduction to stochastic gradient estimation and to make concrete, the simplicity of the estimator used in [90].

3.2.1 Univariate objective function

Suppose the objective function domain is one-dimensional. Then, by the second order Taylor expansion around x,

$$f(\tilde{x}) = f(x) + f'(x)(\tilde{x} - x) + \frac{1}{2}f''(x)(\tilde{x} - x)^2 + R_3 \tag{3.2.1}$$

where $\tilde{x}$ is the variable and $R_3 = {}^{f'''(\xi)(\tilde{x}-x)^3}/_6$ is Lagrange's form of the remainder term with ξ taking some specific value in the interval $[\tilde{x}, x]$. Consider now the difference $\tilde{x} - x$ to be a perturbation, $\epsilon = \tilde{x} - x$. The Taylor approximation can then be written in perturbation form as

$$f(x+\epsilon) = f(x) + f'(x)\epsilon + \frac{1}{2}f''(x)\epsilon^2 + R_3 \tag{3.2.2}$$

where ϵ is a small number and the remainder term is $R_3 = {}^{f'''(\xi)\epsilon^3}/_6$. Multiplying (3.2.2) by a factor of ϵ yields

$$\epsilon f(x) = \epsilon f(x) + f'(x)\epsilon^2 + \frac{1}{2}f(x)\epsilon^3 + \epsilon R_3 \ . \tag{3.2.3}$$

Now take $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to be a normally distributed random variable with zero mean and variance σ^2. That is, a random perturbation to the value at which the objective function is evaluated. Taking the expectation on both sides of (3.2.3) yields

$$\begin{aligned} \mathrm{E}[\epsilon f(x+\epsilon)] &= \mathrm{E}\left[f(x)\epsilon + f'(x)\epsilon^2 + \frac{1}{2}f''(x)\epsilon^3 + \epsilon R_3\right] \\ &= \mathrm{E}[\epsilon]f(x) + \mathrm{E}[\epsilon^2]f'(x) + \mathrm{E}[\epsilon^3]\frac{1}{2}f''(x) + \mathrm{E}[\epsilon R_3] \\ &= \sigma^2 f'(x) + \mathrm{E}[\epsilon R_3] \ , \end{aligned} \tag{3.2.4}$$

using the fact that for a univariate Gaussian the plain central moments are given by

$$\mathrm{E}[\epsilon^p] = \begin{cases} 0 & \text{if } p \text{ is odd} \\ \sigma^p(p-1)!! & \text{if } p \text{ is even}^1 \ . \end{cases} \tag{3.2.5}$$

The gradient estimate is obtained by rearranging (3.2.4) and neglecting the expectation of the remainder, $\mathrm{E}[\epsilon R_3]$, giving

$$f'(x) \approx \frac{1}{\sigma^2}\mathrm{E}[f(x+\epsilon)\epsilon] \ . \tag{3.2.6}$$

[1] Here, !! denotes the double factorial or semifactorial; the product of all integers from 1 to n that have the same parity as n.

Making use of the reparameterization trick to sample the perturbation from a standard Gaussian,

$$\epsilon = \sigma\hat{\epsilon}, \quad \epsilon \sim \mathcal{N}(0, \sigma^2) \ , \quad \hat{\epsilon} \sim \mathcal{N}(0, 1) \ , \tag{3.2.7}$$

the gradient estimate can also be written as

$$f'(x) \approx \frac{1}{\sigma}\mathrm{E}[f(x + \sigma\hat{\epsilon})\hat{\epsilon}] \ . \tag{3.2.8}$$

This estimator can be seen to equal the one presented in [90] in the univariate case.

The gradient can easily be estimated in practice by Monte Carlo approximation. A Monte Carlo estimate of an expectation of any function $g(\mathbf{x})$ w.r.t. any probability distribution $p(\mathbf{x}|\boldsymbol{\theta})$ in the continuous case is given by [70]

$$\mathrm{E}[g(\mathbf{x})]_{\mathbf{x}\sim p(\mathbf{x}|\boldsymbol{\theta})} = \int g(\mathbf{x})p(\mathbf{x}|\boldsymbol{\theta})\,\mathrm{d}\mathbf{x} \approx \frac{1}{N}\sum_{n=1}^{N} g(\mathbf{x}_n) \ , \tag{3.2.9}$$

where $\mathbf{x}_n \sim p(\mathbf{x}|\boldsymbol{\theta})$. By setting $g(x+\epsilon) = \frac{\epsilon}{\sigma^2}f(x+\epsilon) = \frac{\hat{\epsilon}}{\sigma}f(x+\sigma\hat{\epsilon})$ and taking the expectation w.r.t. to ϵ, the gradient in (3.2.6) and (3.2.8) can be estimated by

$$f'(x) \approx \mathrm{E}[g(x+\epsilon)]_{\epsilon\sim\mathcal{N}(0,\sigma^2)} \approx \frac{1}{N\sigma^2}\sum_{n=1}^{N} f(x+\epsilon_n)\epsilon_n = \frac{1}{N\sigma}\sum_{n=1}^{N} f(x+\sigma\hat{\epsilon}_n)\hat{\epsilon}_n \ . \tag{3.2.10}$$

3.2.2 Bias and variance of estimator

The gradient estimator in (3.2.8) is not an unbiased estimate. This is the case since generally $\mathrm{E}[\epsilon R_3] > 0$. However, two observations can be made about this bias. First note that the remainder of the Taylor expansion, R_3, depends only on the third order derivative of the objective function at some point $\xi \in [\tilde{x}, x]$. Close enough to the optimum, any objective function becomes approximately quadratic. Since the third order derivative of any quadratic function is zero, it is evident that the bias goes to zero at the optimum. As such the gradient estimator is unbiased at any optimum. Secondly, the bias can be manipulated as follows

$$\begin{aligned}\mathrm{E}[\epsilon R_3] &= \mathrm{E}\left[\frac{1}{6}f'''(\xi)(\tilde{x}-x)^3\epsilon\right] \\ &= \frac{1}{6}\mathrm{E}\left[f'''(\xi)\epsilon^4\right] \\ &= \frac{\sigma^4}{6}\mathrm{E}\left[f'''(\xi)\hat{\epsilon}^4\right] \ . \end{aligned} \tag{3.2.11}$$

From this, it can be seen that the bias is scales with σ^4. Thus, for small σ the bias will be a small number at any distance from an optimum.

The variance of the gradient estimate can be manipulated in a similar manner. Considering equation (3.2.10), the variance is[2]

$$\begin{aligned}
\mathrm{Var}\left[f'(x)\right] &= \mathrm{Var}\left[\frac{1}{N\sigma}\sum_{n=1}^{N} f(x+\sigma\hat{\epsilon}_n)\hat{\epsilon}_n\right] \\
&= \frac{1}{N^2\sigma^2}\sum_{i=1}^{N}\sum_{j=1}^{N}\mathrm{Cov}[f(x+\sigma\hat{\epsilon_i})\hat{\epsilon_i}, f(x+\sigma\hat{\epsilon_j})\hat{\epsilon_j}] \\
&= \frac{1}{N\sigma^2}\mathrm{Var}[f(x+\sigma\hat{\epsilon})\hat{\epsilon}] \ . \qquad (3.2.12)
\end{aligned}$$

where it has been used that the off diagonal terms of the covariance are zero due $\hat{\epsilon}_i$ and $\hat{\epsilon}_j$ being independent and identically distributed (IID) and that $\mathrm{Var}[\hat{\epsilon}_i]$ is the same on average for all i. If σ is small or $\sigma \to 0$, $f(x+\sigma\hat{\epsilon})$ can be Taylor expanded to first order around x as in (3.2.2). This gives

$$\begin{aligned}
\mathrm{Var}\left[f'(x)\right] &\approx \frac{1}{N\sigma^2}\mathrm{Var}\left[f(x)\hat{\epsilon} + f'(x)\sigma\hat{\epsilon}^2\right] \\
&= \frac{1}{N\sigma^2}\left(f(x)^2\mathrm{Var}[\hat{\epsilon}] + f'(x)^2\sigma^2\mathrm{Var}\left[\hat{\epsilon}^2\right] + 2\sigma f(x)f'(x)\mathrm{Cov}\left[\hat{\epsilon},\hat{\epsilon}^2\right]\right) \\
&= \frac{1}{N\sigma^2}\left(f(x)^2 + f'(x)^2\sigma^2\left(\mathrm{E}\left[\hat{\epsilon}^4\right] - \mathrm{E}\left[\hat{\epsilon}^2\right]^2\right)\right) \\
&= \frac{1}{N\sigma^2}\left(f(x)^2 + 2f'(x)^2\sigma^2\right) \\
&= \frac{1}{N\sigma^2}f(x)^2 + \frac{2}{N}f'(x)^2 \ . \qquad (3.2.13)
\end{aligned}$$

Clearly, for small σ or $\sigma \to 0$, the variance can become very large and the gradient estimator unusable in practice. These observations about the bias and variance of the estimator and their dependency σ make it clear that there is a tradeoff to be made between the bias and variance; lowering σ decreases the bias but increases the variance, and vice versa. This is a clear example of a bias-variance tradeoff. The variance will be considered closer in later sections. The dependency of the estimator on the variance will be altered by using the natural gradient in Section 3.4 and the methods for variance reduction will be considered in Section 3.6. The calculation above has been simulated for the multivariate case as described in Appendix G.

3.2.3 Second order derivative

In addition to an estimate of the gradient, the derivation above also makes second order information readily available. Again considering (3.2.2) but this time multiplying through by

[2]This is considered and argued for in more detail in Section 3.6

$\left(\frac{\epsilon^2}{\sigma^2}-1\right)$ and taking the expectation yields

$$\begin{aligned}
\mathrm{E}\left[f(x+\epsilon)\left(\frac{\epsilon^2}{\sigma^2}-1\right)\right] &= \mathrm{E}\left[\left(f(x)+f'(x)\epsilon+\frac{1}{2}f''(x)\epsilon^2+R_3\right)\left(\frac{\epsilon^2}{\sigma^2}-1\right)\right] \\
&= \frac{1}{\sigma^2}\mathrm{E}\left[f(x)\epsilon^2+f'(x)\epsilon^3+\frac{1}{2}f''(x)\epsilon^4+\epsilon^2 R_3\right] \\
&\quad -\mathrm{E}\left[f(x)+f'(x)\epsilon+\frac{1}{2}f''(x)\epsilon^2+R_3\right] \\
&= \left(f(x)+3\sigma^2 f''(x)+\mathrm{E}[\epsilon^2 R_3]\right)-\left(f(x)+\frac{1}{2}\sigma^2 f''(x)+\mathrm{E}[R_3]\right) \\
&= \frac{5}{2}\sigma^2 f''(x)+\mathrm{E}\left[R_3(\epsilon^2-1)\right] \ .
\end{aligned}$$

Neglecting the remainder term and rearranging as for the gradient yields the estimate of the second order derivative.

$$\begin{aligned}
f''(x) &\approx \frac{2}{5\sigma^2}\mathrm{E}\left[f(x+\epsilon)\left(\frac{\epsilon^2}{\sigma^2}-1\right)\right] && (3.2.14) \\
&= \frac{2}{5\sigma^2}\mathrm{E}\left[f(x+\sigma\hat{\epsilon})\left(\hat{\epsilon}^2-1\right)\right] && (3.2.15)
\end{aligned}$$

which can also be estimated in practice by a Monto Carlo method

$$f''(x) \approx \frac{2}{5N\sigma^2}\sum_{n=1}^{N} f(x+\sigma\hat{\epsilon}_n)\left(\hat{\epsilon}_n^2-1\right) \ . \qquad (3.2.16)$$

From the above expression it can be noted that no additional sampling or evaluation of the objective function is required for computing the estimate of the second order derivative compared to the gradient. As such, the derived stochastic optimization scheme accrues almost no additional computational cost by computing second order information, especially if evaluating $f(\cdot)$ is expensive.

The variance of this second order estimate may however be rather high for small values of σ.

$$\mathrm{Var}\left[f''(x)\right] = \frac{4}{25N^2\sigma^4}\sum_{i=1}^{N}\sum_{j=1}^{N}\mathrm{Cov}\left[f(x+\sigma\hat{\epsilon}_i)\left(\hat{\epsilon}_i^2-1\right), f(x+\sigma\hat{\epsilon}_j)\left(\hat{\epsilon}_j^2-1\right)\right] \ . \qquad (3.2.17)$$

In theory, the number of higher order terms that can be estimated is free to choose. Their variance, however, may become so high that their contribution to the updates of x is at best negligible with the risk of being detrimental.

3.2.4 Derivative estimates as sample covariances

To obtain an intuition for the correctness of the estimators, rewrite (3.2.6) as follows.

$$\begin{aligned}
f'(x) &\approx \frac{1}{\sigma^2}\mathrm{E}\left[\epsilon f(x+\epsilon)\right] \\
&= \frac{1}{\sigma^2}\left(\mathrm{E}\left[\epsilon\right]\mathrm{E}\left[f(x+\epsilon)\right]+\mathrm{Cov}\left[\epsilon, f(x+\epsilon)\right]\right) \\
&= \frac{1}{\sigma^2}\mathrm{Cov}\left[\epsilon, f(x+\epsilon)\right] \ .
\end{aligned}$$

In this perspective, the gradient estimator is simply a scaled version of the covariance between the perturbations and the objective function evaluated at the perturbations. The intuition behind this is fairly straightforward: If the function value increases for a positive perturbation and decreases for a negative, the covariance is positive and the gradient is positive. The opposite holds for the inverse. In an optimum, the function is approximately a quadratic function of the perturbation,

$$f(x+\epsilon) \underset{\sim}{\propto} \epsilon^2 \ ,$$

so the change in function value is approximately the same regardless of the sign of the perturbation. Then

$$\begin{aligned} f'(x) &\underset{\sim}{\propto} \frac{1}{\sigma^2}\mathrm{Cov}\left[\epsilon, \epsilon^2\right] \\ &= \frac{1}{\sigma^2}\mathrm{E}\left[(\epsilon - \mathrm{E}[\epsilon])\left(\epsilon^2 - \mathrm{E}\left[\epsilon^2\right]\right)\right] \\ &= \frac{1}{\sigma^2}\mathrm{E}\left[\epsilon\left(\epsilon^2 - \sigma^2\right)\right] \\ &= \frac{1}{\sigma^2}\mathrm{E}\left[\epsilon^3 + \sigma\epsilon\right] \\ &= 0 \ . \end{aligned}$$

As such, the covariance is zero at an optimum, as would be the case for the true gradient.

The same interpretation holds for the estimate of the second order derivative as can be seen from (3.2.14).

$$\begin{aligned} f''(x) &\approx \frac{2}{5\sigma^2}\mathrm{E}\left[f(x+\epsilon)\left(\frac{\epsilon^2}{\sigma^2} - 1\right)\right] \\ &= \frac{2}{5\sigma^2}\left(\mathrm{E}\left[f(x+\epsilon)\right]\mathrm{E}\left[\frac{\epsilon^2}{\sigma^2} - 1\right] + \mathrm{Cov}\left[\frac{\epsilon^2}{\sigma^2} - 1, f(x+\epsilon)\right]\right) \\ &= \frac{2}{5\sigma^4}\mathrm{Cov}\left[\epsilon^2, f(x+\epsilon)\right] \ . \end{aligned}$$

Since the squared perturbation is always positive it holds for any perturbation that if the function increases, the covariance between it and the squared perturbation is positive. If the function decreases it is negative. This is exactly the behaviour expected for the second order derivative which estimates the curvature of the function: At points where the function is convex, the second order derivative is positive. At points where it is concave, the second order derivative is negative. Finally, if for some set of perturbations the function does not change, this indicates a plateau which is again in line with the behaviour of the second order derivative.

3.2.5 Multivariate case

In general, the objective function can be multivariate in which case the variables can be collected in a d-dimensional vector $\mathbf{x} \in \mathbb{R}^d$. Then, the Taylor expansion in (3.2.2) reads

$$f(\mathbf{x}+\boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^{\mathrm{T}}\nabla_{\mathbf{x}}f(\mathbf{x}) + \frac{1}{2}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{H}_{\mathbf{x}}f(\mathbf{x})\boldsymbol{\epsilon} + R_3 \tag{3.2.18}$$

where $\boldsymbol{\epsilon}$ is a perturbation vector, $\nabla_{\mathbf{x}}f(\mathbf{x})$ is the gradient of f with respect to $\mathbf{x}$ and $\mathbf{H}_{\mathbf{x}}f(\mathbf{x})$ is its Hessian. The gradient is defined as a column vector of the first order partial derivatives

$$\nabla_{\mathbf{x}}f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_d} \end{bmatrix}^{\mathrm{T}} \tag{3.2.19}$$

and the Hessian is a matrix of the second order partial derivatives

$$\mathbf{H}_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \frac{\partial^2 f}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix} . \tag{3.2.20}$$

Left-multiplying (3.2.18) by the random vector $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$ and taking the expectation gives

$$\begin{aligned} \mathrm{E}\left[\boldsymbol{\epsilon} f(\mathbf{x}+\boldsymbol{\epsilon})\right] &= \mathrm{E}\left[\boldsymbol{\epsilon} f(\mathbf{x}) + \boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}} \nabla_{\mathbf{x}} f(\mathbf{x}) + \frac{1}{2}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}} \mathbf{H}_{\mathbf{x}} f(\mathbf{x}) \boldsymbol{\epsilon} + \boldsymbol{\epsilon} R_3\right] \\ &= \mathrm{E}\left[\boldsymbol{\epsilon}\right] f(\mathbf{x}) + \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right] \nabla_{\mathbf{x}} f(\mathbf{x}) + \frac{1}{2}\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}} \mathbf{H}_{\mathbf{x}} f(\mathbf{x}) \boldsymbol{\epsilon}\right] + \mathrm{E}\left[\boldsymbol{\epsilon} R_3\right] . \end{aligned} \tag{3.2.21}$$

Now, $\mathrm{E}\left[\boldsymbol{\epsilon}\right] = \mathbf{0}$, $\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right] = \boldsymbol{\Sigma}$ and $\mathbf{H}_{\mathbf{x}} f(\mathbf{x}) = \mathbf{C}^{\mathrm{T}}\mathbf{C}$ by the Cholesky factorization. Then

$$\begin{aligned} \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}} \mathbf{H}_{\mathbf{x}} f(\mathbf{x}) \boldsymbol{\epsilon}\right] &= \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}} \mathbf{C}^{\mathrm{T}}\mathbf{C}\boldsymbol{\epsilon}\right] \\ &= \mathrm{Cov}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}, \boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\right] + \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\right] \mathrm{E}\left[\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\right]^{\mathrm{T}} \qquad \text{by covariance def.} \\ &= \mathbf{C}\mathrm{Cov}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}, \boldsymbol{\epsilon}^{\mathrm{T}}\right] \mathbf{C}^{\mathrm{T}} + \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right] \mathbf{C}^{\mathrm{T}}\mathbf{C}\mathrm{E}\left[\boldsymbol{\epsilon}\right] \\ &= \mathbf{C}\left(\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] - \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right] \mathrm{E}\left[\boldsymbol{\epsilon}^{\mathrm{T}}\right]^{\mathrm{T}}\right) \mathbf{C}^{\mathrm{T}} + \boldsymbol{\Sigma}\mathbf{C}^{\mathrm{T}}\mathbf{C}\mathbf{0} \\ &= \mathbf{C}\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] \mathbf{C} , \end{aligned}$$

where it has been used that $\mathrm{E}\left[\mathbf{x}\mathbf{y}^{\mathrm{T}}\right] = \mathrm{E}\left[\mathbf{x}\right] \mathrm{E}\left[\mathbf{y}\right]^{\mathrm{T}} + \mathrm{Cov}\left[\mathbf{x}, \mathbf{y}\right]$ for random vectors $\mathbf{x}$ and $\mathbf{y}$. To evaluate $\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right]$ note that the vector product term takes the form

$$\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1^2 & \epsilon_1\epsilon_2 & \cdots & \epsilon_1\epsilon_d \\ \epsilon_1\epsilon_2 & \epsilon_2^2 & \cdots & \epsilon_2\epsilon_d \\ \vdots & \vdots & \ddots & \vdots \\ \epsilon_1\epsilon_d & \epsilon_2\epsilon_d & \cdots & \epsilon_d^2 \end{bmatrix} \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_d \end{bmatrix} = \begin{bmatrix} \epsilon_1^3 + \epsilon_1\epsilon_2^2 + \cdots + \epsilon_1\epsilon_d^2 \\ \epsilon_1^2\epsilon_2 + \epsilon_2^3 + \cdots + \epsilon_2\epsilon_d^2 \\ \vdots \\ \epsilon_1^2\epsilon_d + \epsilon_2^3 + \cdots + \epsilon_d^3 \end{bmatrix} .$$

Since the expected value applies element-wise to a random vector and is linear, the result is a vector where each element is a sum of expectations of the form

$$\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] = \begin{bmatrix} \mathrm{E}\left[\epsilon_1^3\right] + \mathrm{E}\left[\epsilon_1\epsilon_2^2\right] + \cdots + \mathrm{E}\left[\epsilon_1\epsilon_d^2\right] \\ \mathrm{E}\left[\epsilon_1^2\epsilon_2\right] + \mathrm{E}\left[\epsilon_2^3\right] + \cdots + \mathrm{E}\left[\epsilon_2\epsilon_d^2\right] \\ \vdots \\ \mathrm{E}\left[\epsilon_1^2\epsilon_d\right] + \mathrm{E}\left[\epsilon_2^3\right] + \cdots + \mathrm{E}\left[\epsilon_d^3\right] \end{bmatrix} .$$

The expectations can be recognized to be respectively the third moment in the diagonal and third order cross-moments in the off-diagonal. Since $\boldsymbol{\epsilon}$ derives from a zero mean Gaussian distribution these are all zero and

$$\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{H}_{\mathbf{x}} f(\mathbf{x})\boldsymbol{\epsilon}\right] = \mathbf{C}^{\mathrm{T}}\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] \mathbf{C} = \mathbf{0} . \tag{3.2.22}$$

It then follows from (3.2.21) that the gradient can be approximated by

$$\nabla_{\mathbf{x}} f(\mathbf{x}) \approx \boldsymbol{\Sigma}^{-1}\mathrm{E}\left[\boldsymbol{\epsilon} f(\mathbf{x}+\boldsymbol{\epsilon})\right] . \tag{3.2.23}$$

In the case of an isotropic Gaussian, the covariance matrix has the special structure $\mathbf{\Sigma} = \sigma^2\mathbf{I}$. The gradient estimate then simplifies to the one-dimensional gradient in each of the dimensions. Sampling from a standard Gaussian, $\boldsymbol{\epsilon} = \sigma\hat{\boldsymbol{\epsilon}}$ where $\hat{\boldsymbol{\epsilon}} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, the estimate can be written

$$\nabla_{\mathbf{x}} f(\mathbf{x}) \approx \frac{1}{\sigma} \mathrm{E}\left[\hat{\boldsymbol{\epsilon}} f(\mathbf{x} + \sigma\hat{\boldsymbol{\epsilon}})\right] \ . \tag{3.2.24}$$

This estimator is identical to the gradient estimator used in [90] and is also known under different names including *simultaneous perturbation stochastic approximation* [104], *parameter-exploring policy gradients* [100], or *zero-order gradient estimation* [72]. Again, a Monto Carlo method gives the estimate in practice by sampling

$$\nabla_{\mathbf{x}} f(\mathbf{x}) \approx \frac{1}{N\sigma} \sum_{n=1}^{N} \hat{\boldsymbol{\epsilon}}_n f(\mathbf{x} + \sigma\hat{\boldsymbol{\epsilon}}_n) \ . \tag{3.2.25}$$

For this estimator, the observations made about the bias, variance and covariance interpretation for the univariate case hold in each dimension.

3.3 Variational optimization

VO is an optimization technique that can be applied on almost any type of objective function including functions that are non-differentiable or discrete [106]. This section first derives the VO objective as a differentiable upper bound on the objective function and its gradient. Section 3.3.2 then discusses VO in relation to policy gradients, computational efficiency, dimensionality and loss surfaces and Bayesian neural networks (BNNs). The following sections present different possible choices of search distributions and derive the corresponding practical gradient estimators. Using the univariate Gaussian distribution, the intuitions of VO are illustrated in Section 3.3.3.2.

3.3.1 Formal derivation

Consider the general function minimization problem $\min_{\mathbf{x}} f(\mathbf{x})$ for some multivariate function $f : \mathcal{D} \to \mathcal{C}$ mapping from the domain $\mathcal{D}$ to its codomain $\mathcal{C}$ and vector input $\mathbf{x} \in \mathcal{D}$. The expected value of a function w.r.t. any probability distribution is guaranteed to always larger than or equal to the minimal value of the function. Equality is only achieved in the case of a constant function. Therefore, this expectation can be defined as an upper bound on the optimum of objective[3],

$$f(\mathbf{x}^*) = \min_{\mathbf{x}\in\mathcal{D}} f(\mathbf{x}) \leq \mathrm{E}[f(\mathbf{x})]_{p(\mathbf{x}|\boldsymbol{\theta})} \equiv U(\boldsymbol{\theta}) \ . \tag{3.3.1}$$

Clearly, this bound can be made arbitrarily tight provided that the search distribution is flexible enough to allow for all of its probability mass to be placed in the optimum $\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x})$ [106].

Now, instead of minimizing $f(\mathbf{x})$ w.r.t. $\mathbf{x}$, the upper bound $U(\boldsymbol{\theta})$ can be optimized w.r.t. the parameters $\boldsymbol{\theta}$ of the search distribution $p(\mathbf{x}|\boldsymbol{\theta})$. That is, the original objective function is replaced with an upper bound which is then minimized.

$$f(\mathbf{x}^*) \leq \min_{\boldsymbol{\theta}} \mathrm{E}[f(\mathbf{x})]_{p(\mathbf{x}|\boldsymbol{\theta})} = \min_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) \ . \tag{3.3.2}$$

[3]This bound is obviously not true for $f(\mathbf{x})$ in general but holds only at the minimum, $f(\mathbf{x}^*) = \min_{\mathbf{x}} f(\mathbf{x})$.

The gradient of the variational objective can be written as

$$\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathrm{E}[f(\mathbf{x})]_{p(\mathbf{x}|\boldsymbol{\theta})} = \nabla_{\boldsymbol{\theta}} \int_{\mathcal{D}} f(\mathbf{x}) p(\mathbf{x}|\boldsymbol{\theta}) \mathrm{d}\mathbf{x} \ . \tag{3.3.3}$$

By the probability identity trick and the log-derivative trick, the gradient of the upper bound can be rewritten as the expectation of a score function. The log-derivative trick is another name for the application of the chain rule to the logarithm of a function and can be written as

$$\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta}) = \frac{\nabla_{\boldsymbol{\theta}} p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})} \ . \tag{3.3.4}$$

The gradient of the upper bound can then be rewritten as follows

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathrm{E}[f(\mathbf{x})]_{p(\mathbf{x}|\boldsymbol{\theta})} && (3.3.5)\\
&= \nabla_{\boldsymbol{\theta}} \int f(\mathbf{x}) p(\mathbf{x}|\boldsymbol{\theta}) \mathrm{d}\mathbf{x} \\
&= \int f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} p(\mathbf{x}|\boldsymbol{\theta}) \mathrm{d}\mathbf{x} \\
&= \int f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} p(\mathbf{x}|\boldsymbol{\theta}) \frac{p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})} \mathrm{d}\mathbf{x} \\
&= \int f(\mathbf{x}) p(\mathbf{x}|\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta}) \mathrm{d}\mathbf{x} \\
&= \mathrm{E}[f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})]_{p(\mathbf{x}|\boldsymbol{\theta})} && (3.3.6)
\end{aligned}$$

where integration and differentiation can be interchanged given the weak conditions [106]:

1. $f(\mathbf{x})p(\mathbf{x}|\boldsymbol{\theta})$ is Lebesgue integrable and differentiable wrt. $\boldsymbol{\theta}$.
2. There exists an integrable function $F : \mathcal{D} \to \mathbb{R}$ such that, for all $\boldsymbol{\theta}$, $|\nabla_{\boldsymbol{\theta}} f(\mathbf{x}) p(\mathbf{x}|\boldsymbol{\theta})| < F(\mathbf{x})$.

As noted in [106], these are often satisfied and will not be considered further here.

By evaluating the expectation with respect to $p(\mathbf{x}|\boldsymbol{\theta})$ by Monte Carlo approximation, a practical estimator becomes

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) &= \mathrm{E}[f(\mathbf{x}) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})]_{p(\mathbf{x}|\boldsymbol{\theta})} \\
&\approx \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x}_n) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}_n|\boldsymbol{\theta}) \ . && (3.3.7)
\end{aligned}$$

In an algorithmic perspective, the update to the search distribution can be computed by gradient descent

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{N} \sum_{n=1}^{N} f(\mathbf{x}_n) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}_n|\boldsymbol{\theta}) \tag{3.3.8}$$

for some choice of learning rate η.

This derivation allowed the transformation of the gradient of an expectation into an expectation of a score function, $\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})$. As such, the derived estimator is a score function. Score function estimators are related to likelihood ratio methods, automated variational inference [120] and the REINFORCE rule and policy gradients [119]. Additionally, it can be noted that unlike the gradient estimator derived by the Taylor expansion, the VO gradient derived here is unbiased. Its variance is considered in Section 3.6.

3.3.2 Remarks on variational optimization

Before proceeding with different search distributions, a few remarks will be made about VO in relation to BNNs, policy gradients, computational efficiency, dimensionality and loss surfaces.

3.3.2.1 Relation to Bayesian neural networks

A BNN is a combination of a probabilistic graphical model and a neural network. The result of training is not a point estimate of the weights but a full posterior distribution over the weights [29]. With VO, a search distribution over potential weights is maintained and adjusted during training in order to minimize the VO upper bound on the objective. The result is in fact a distribution over the weights however, using e.g. a Gaussian search distribution, the variance is potentially driven to zero, effectively yielding a point estimate as in the standard maximum likelihood approach. Additionally, the variance of the search distribution does not directly reflect uncertainty in the parameter estimates but instead the degree of smoothing applied to the potentially non-smooth objective at each iteration.

3.3.2.2 Comparison to policy gradient methods

The policy gradients approach to RL can be formulated as seeking to maximize some reward R which depends on a sequence of actions $\mathbf{a} = \{a_1, \dots, a_T\}$ that are in turn determined by a policy network parameterized by a number of parameters $\boldsymbol{\mu}$. The function to be optimized is then $f(\boldsymbol{\mu}) = R(\mathbf{a}(\boldsymbol{\mu}))$. The possibility of discrete actions and deterministic policy allows $f(\boldsymbol{\mu})$ to be a non-smooth function. This, and the lack of knowledge about the environment state transition function makes direct application of gradient based optimization of the policy impossible [90].

To obtain gradient estimates, noise must be added to the objective and this is where policy gradients and VO differ. As described, VO adds noise in the parameter space and then picks the most probable action at each iteration. This gives the gradient in (3.3.7). As will be shown later (Section 3.6), the variance of the VO gradient can be written as

$$\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})] = \frac{1}{N}\mathrm{Var}[R(\mathbf{a}(\mathbf{x}))\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})] \tag{3.3.9}$$

where N is the number of perturbations used for the Monte Carlo estimation and $\mathbf{x}$ is a perturbation of the parameters, $\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{\epsilon}$. The policy parameters $\boldsymbol{\mu}$ are included in the search distribution parameter vector $\boldsymbol{\theta}$ which can also include other parameters relevant for the distribution.

Policy gradients on the other hand adds the noise in the action space by sampling the action to take at each iteration from a categorical distribution estimated by the policy network. For a discrete action space, the policy gradients objective can be written as [90]

$$\nabla_{\boldsymbol{\mu}} f(\boldsymbol{\mu}, \epsilon) = \mathrm{E}_{\epsilon}[R(\mathbf{a}(\boldsymbol{\mu}, \epsilon))\nabla_{\boldsymbol{\mu}} \log p(\mathbf{a}(\boldsymbol{\mu}, \epsilon)|\boldsymbol{\mu})] \ . \tag{3.3.10}$$

where ϵ is the random variable used for sampling. When computing this gradient by Monte Carlo estimation, its variance becomes [90]

$$\mathrm{Var}[\nabla_{\boldsymbol{\mu}} f(\boldsymbol{\mu}, \epsilon)] \approx \mathrm{Var}[R(\mathbf{a}(\boldsymbol{\mu}, \epsilon))\nabla_{\boldsymbol{\mu}} \log p(\mathbf{a}(\boldsymbol{\mu}, \epsilon)|\boldsymbol{\mu})] \ . \tag{3.3.11}$$

This follows from calculations similar to those that will be made in Section 3.6 which considers variance reduction methods for VO.

Now, the variance of the VO and policy gradients in (3.3.9) and (3.3.11) can be compared. If the total reward is only weakly correlated with each action in $\mathbf{a}$, then

$$\operatorname{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})] \approx \frac{1}{N}\operatorname{Var}[R(\mathbf{a}(\mathbf{x}))]\operatorname{Var}[\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}|\boldsymbol{\theta})] \tag{3.3.12}$$

$$\operatorname{Var}[\nabla_{\boldsymbol{\mu}} f(\boldsymbol{\mu},\epsilon)] \approx \operatorname{Var}[R(\mathbf{a}(\boldsymbol{\mu},\epsilon))]\operatorname{Var}[\nabla_{\boldsymbol{\mu}} \log p(\mathbf{a}(\boldsymbol{\mu},\epsilon)|\boldsymbol{\mu})]\ . \tag{3.3.13}$$

This is often the case in a difficult RL problem. If the amount of exploration is similar in the two methods then $\operatorname{Var}[R(\mathbf{a}(\boldsymbol{\mu},\epsilon))] \sim \operatorname{Var}[R(\mathbf{a}(\mathbf{x}))]$. The difference is then in last term. For policy gradients, $\nabla_{\boldsymbol{\mu}} \log p(\mathbf{a}(\boldsymbol{\mu},\epsilon)|\boldsymbol{\mu})$ is a sum of uncorrelated terms of each action taken and thus its variance increases linearly with the episode horizon, T. The VO variance does not depend on T but instead decreases with increasing number of evaluated perturbations. When effects are long-lasting, the conventional way to reduce the policy gradients variance by discounting rewards can introduce bias in the gradient estimate. In this case, VO may provide a better gradient estimate.

Another approach to variance reduction in policy gradients is through a hybridization with value function approximations called actor-critic policy gradients. Here, the action-value function $Q(\mathbf{s},\mathbf{a})$ is approximated which measures the value (or quality) of taking action $\mathbf{a}$ in state $\mathbf{s}$ and replaces the reward returned by the environment. This reduces variance but introduces bias by approximating the gradient. In order to further reduce variance, a baseline can be substracted from $Q(\mathbf{s},\mathbf{a})$. A good baseline is the value function $V(\mathbf{s})$ which gives the value of any state and must also be estimated. The result is the advantage function $A(\mathbf{s},\mathbf{a}) = Q(\mathbf{s},\mathbf{a}) - V(\mathbf{s})$ [3]. In VO a similar approach exists under the name of fitness baselines which also subtract a constant term from the objective function in order to reduce variance [122]. This approach does not require the approximation of one or two additional value functions but instead relies on the search distribution, its Fisher information matrix (FIM) and the objective function evaluations already made.

3.3.2.3 Computational considerations

The computation of the gradient of the VO objective by Monte Carlo approximation is easily parallelized as each function evaluation is independent of any other evaluations. If the evaluation of each $f(\mathbf{x}_n)\nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}_n|\boldsymbol{\theta}))$ is time consuming or N is very large, significant speed up can be expected by distributing these computations among many CPUs. Furthermore, the search points $\mathbf{x}_n$ are uniquely defined by their probability distribution and the random seed that was used to sample them. That is, all information required to request and return a term of the gradient estimator, is the random seed used to sample $\mathbf{x}_n$ and the function value $f(\mathbf{x}_n)$. This allows for low-bandwidth communication between individual machines, in turn making viable the distribution of the computation to a large number of CPUs on many distinct machines that do not share memory. Pseudocode for this algorithm can be seen in Algorithm 1.

3.3.2.4 Dimensionality and loss surfaces

There is a classical distinction to be made on the ratio of the dimensionality of the search space, d, and the number of perturbations on which the gradient estimate is based, N. There are two cases: When $d < N$ the sampled perturbations will likely span the entire search space. However, when $d > N$ then, even though each of the perturbations add noise to every dimension of the search space, they only span a subspace of the search space. Using standard

Algorithm 1 Parallelized Variational Optimization. Adapted from [118]

```
Require: Learning rate $\eta$, search distribution $p(\mathbf{x}|\boldsymbol{\theta})$, objective function $f(\mathbf{x})$
Initialize: $N$ workers with known random seeds
 1: repeat
 2:     for each CPU $i = 1, \dots, N$ do                                  ▷ Parallelizable
 3:         Draw random seed $s_i$
 4:         Sample $\mathbf{x}_i \sim p(\mathbf{x}|\boldsymbol{\theta})$
 5:         Evaluate fitness $f(\mathbf{x}_i)$
 6:     end for
 7:     Share $N$ scalar fitnesses, $f(\mathbf{x}_i)$ and seeds, $s_i$, between all CPUs.
 8:     for each worker $i = 1, \dots, N$ do                               ▷ Parallelizable
 9:         Reconstruct all perturbations $\mathbf{x}_j$ for $j = 1, \dots, N$ using known random seeds.
10:         Compute search distribution and upper bound gradient
                $$\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} f(\mathbf{x}_i) \nabla_{\boldsymbol{\theta}} \log p(\mathbf{x}_i|\boldsymbol{\theta})$$
11:         Update search distribution parameters
                $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})$$
12:     end for
13: until stopping criteria met
```

VO, this subspace will be selected implicitly at random at each iteration by the sampling of the perturbations.

A note can be made on the intrinsic dimension of objective landscapes encountered in different problems. Recently [55] it was shown that many problems have an intrinsic dimension much lower than the number of parameters in the NNs typically trained for solving them. By training models in a small random subspace of their parameter spaces and monitoring solutions while iteratively increasing the subspace size, the authors define the intrinsic problem dimension to be that where the first good solutions arise. In VO, the training procedure is similar to this, but the subspace of the parameter space is selected implicitly at random at each iteration instead of being fixed throughout training. In fact, in high-dimensional search spaces, each of the perturbations sampled at each iteration will likely span a unique subspace such that search is actually performed in multiple non-overlapping subspaces.

A fair amount of research has focused on describing the high-dimensional loss surfaces of neural networks. In [18], it was shown that local minima are exponentially less frequent in high dimensional non-convex optimization than are saddle points. [30] showed that for NNs there exists a linear subspace in which training can proceed by descending along a monotonically decreasing path with no barriers, in line with the rarity of local minima. These properties affect VO and are discussed further in Section 4.7.

Finally, it can be imagined that the gradient estimate of one iteration may be able to benefit from information from the previous iteration(s). A simple way to exploit this is by adding momentum to the gradient. Another rendering of this information could be by explicitly computing promising dimensions for search. Ideas related to this are investigated further in Section 3.5.2 and Section 3.5.3.

3.3.3 Search distributions

When VO is applied to optimization of NNs, $\mathbf{x}$ represents the perturbed parameters of the model while the search distribution defines how these parameters are sampled. When training NNs with Gaussian search distributions, the mean vector $\boldsymbol{\mu}$ parameterizes the best current estimate of the network parameters and constitutes what is here called the unperturbed model. The covariance matrix defines how and how much the sampled parameters $\mathbf{x}$ deviate from $\boldsymbol{\mu}$.

This section first derives a univariate Gaussian estimator in order to exemplify the intuitions of VO. Then it derives the isotropic and separable Gaussian VO gradient estimators which are central for VO and are used in the experimental section. The multivariate Gaussian is also treated and a potential approach for making its covariance matrix computationally feasible using a low-rank approximation is presented.

3.3.3.1 Univariate Gaussian search distribution

Let $p(\mathbf{x}|\boldsymbol{\theta}) = \mathcal{N}(x|\mu,\sigma^2)$ be a univariate Gaussian distribution

$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{\sigma^2}(x-\mu)^2\right) \tag{3.3.14}$$

where $\boldsymbol{\theta} = \left[\mu,\sigma^2\right]^\text{T}$ so $U(\boldsymbol{\theta}) = U(\mu,\sigma^2)$. Let $x = \mu + \sigma\epsilon$ with $\epsilon \sim \mathcal{N}(0,1)$. Then by (3.3.1)

$$U(\mu,\sigma^2) = \text{E}[f(x)]_{\mathcal{N}(x|\mu,\sigma^2)} = \text{E}[f(\mu+\sigma\epsilon)]_{\mathcal{N}(\epsilon|0,1)} \ . \tag{3.3.15}$$

The logarithm of the univariate Gaussian is given by

$$\log\mathcal{N}(x|\mu,\sigma^2) = -\frac{1}{2}\log 2\pi - \frac{1}{2}\log\sigma^2 - \frac{1}{2\sigma^2}(x-\mu)^2 \ , \tag{3.3.16}$$

with derivatives computed as

$$\begin{aligned} \frac{\partial}{\partial\mu}\log\mathcal{N}(x|\mu,\sigma^2) &= \frac{1}{\sigma^2}(x-\mu) = \frac{1}{\sigma}\epsilon \\ \frac{\partial}{\partial\sigma^2}\log\mathcal{N}(x|\mu,\sigma^2) &= -\frac{1}{2\sigma^2} + \frac{1}{2\sigma^4}(x-\mu)^2 = \frac{1}{2\sigma^2}\left(\epsilon^2-1\right) \ . \end{aligned} \tag{3.3.17}$$

The gradient of the upper bound with respect to the parameters of the search distribution is then

$$\begin{aligned} \frac{\partial}{\partial\mu}U(\mu,\sigma^2) &= \frac{1}{\sigma}\text{E}[f(\mu+\sigma\epsilon)\epsilon] \approx \frac{1}{N\sigma}\sum_{n=1}^{N} f(\mu+\sigma\epsilon_n)\epsilon_n \\ \frac{\partial}{\partial\sigma^2}U(\mu,\sigma^2) &= \frac{1}{2\sigma^2}\text{E}\left[f(\mu+\sigma\epsilon)\left(\epsilon^2-1\right)\right] \approx \frac{1}{2N\sigma^2}\sum_{n=1}^{N} f(\mu+\sigma\epsilon_n)\left(\epsilon_n^2-1\right) \end{aligned} \tag{3.3.18}$$

where it has been used that $\nabla_{\boldsymbol{\theta}}U(\boldsymbol{\theta}) = \text{E}[f(\mathbf{x})\nabla_{\boldsymbol{\theta}}\log p(\mathbf{x}|\boldsymbol{\theta})]_{p(\mathbf{x}|\boldsymbol{\theta})}$ as derived in (3.3.6) with $\boldsymbol{\theta} = \left[\mu \quad \sigma^2\right]^\text{T}$.

It can be noted that the derived first order derivative of the upper bound w.r.t. μ in (3.3.18) is in fact identical to the estimator for the first order derivative of the objective function as derived by Taylor expansion in (3.2.8). This establishes a clear connection between the gradient estimate of [90] and VO: The estimator in [90] can be seen as a special case of VO using a Gaussian search distribution and a fixed variance.

The Taylor estimator in (3.2.8) is biased due to the negligence of the remainder term but the estimator derived here is in fact unbiased. Since the two estimators are equal it must be concluded that the remainder of Taylor expansion estimator has an expected value of zero.

During optimization of the search distribution parameters, it must be ensured that $\sigma > 0$ at every iteration. A simple way to achieve this is by an exponential reparameterization of the variance. Let

$$\sigma^2 = e^{\beta} \ . \tag{3.3.19}$$

Since $e^{\beta} > 0 \ \forall \beta$ then by optimizing β rather than σ the constraint $\sigma > 0$ is guaranteed to be satisfied. With this parameterization

$$\log \mathcal{N}(x|\mu, \beta) = -\frac{1}{2}\log 2\pi - \frac{1}{2}\beta - \frac{1}{2}e^{-\beta}(x-\mu)^2, \tag{3.3.20}$$

such that

$$\begin{aligned} \frac{\partial}{\partial \mu}\log \mathcal{N}(x|\mu, \beta) &= e^{-\beta}(x-\mu) = e^{-\frac{1}{2}\beta}\epsilon \\ \frac{\partial}{\partial \beta}\log \mathcal{N}(x|\mu, \beta) &= -\frac{1}{2} + \frac{1}{2}e^{-\beta}(x-\mu)^2 = \frac{1}{2}\big(\epsilon^2 - 1\big) \end{aligned} \tag{3.3.21}$$

and

$$\begin{aligned} \frac{\partial}{\partial \mu}U(\mu, \beta) &= \frac{1}{\sigma}\mathrm{E}\Big[f\Big(\mu + e^{\frac{1}{2}\beta}\epsilon\Big)\epsilon\Big] \approx \frac{e^{-\beta}}{N}\sum_{n=1}^{N} f\Big(\mu + e^{\frac{1}{2}\beta}\epsilon_n\Big)\epsilon_n \\ \frac{\partial}{\partial \beta}U(\mu, \beta) &= \frac{1}{2}\mathrm{E}\Big[f\Big(\mu + e^{\frac{1}{2}\beta}\epsilon\Big)\left(\epsilon^2 - 1\right)\Big] \approx \frac{1}{2N}\sum_{n=1}^{N} f\Big(\mu + e^{\frac{1}{2}\beta}\epsilon_n\Big)\left(\epsilon_n^2 - 1\right) \ . \end{aligned} \tag{3.3.22}$$

This is essentially the same result for μ as before. The derivative w.r.t. β however only depends on β through the function evaluations and not in the factor in front of the sum, as was the case for the derivative w.r.t. σ^2. This type of reparameterization can be directly applied for all Gaussians with diagonal covariance. This is done throughout the thesis.

3.3.3.2 Examples for the univariate Gaussian

For the univariate Gaussian distribution considered here, it is possible to visualize the intuition of VO. The effect of applying VO with a univariate Gaussian to a univariate objective function is to turn the one-dimensional optimization problem $\min_x f(x)$ into a two-dimensional one, $\min_{\{\mu,\sigma^2\}} U(\mu, \sigma^2)$.

Specifically, consider the negative sinc function, defined as $\text{sinc}(x) = \sin(\pi x)/\pi x$ seen in Figure 3.1a. If one were to optimize the sinc function using gradient descent, the resulting minimum would be highly dependent on the choice of the initial point with the algorithm converging to the global minimum only for initial points between about ± 1.43. The Gaussian VO objective function corresponding to the sinc is visualized in Figure 3.1b for different values of the search distribution variables μ and σ^2. Note especially that optimization is now over

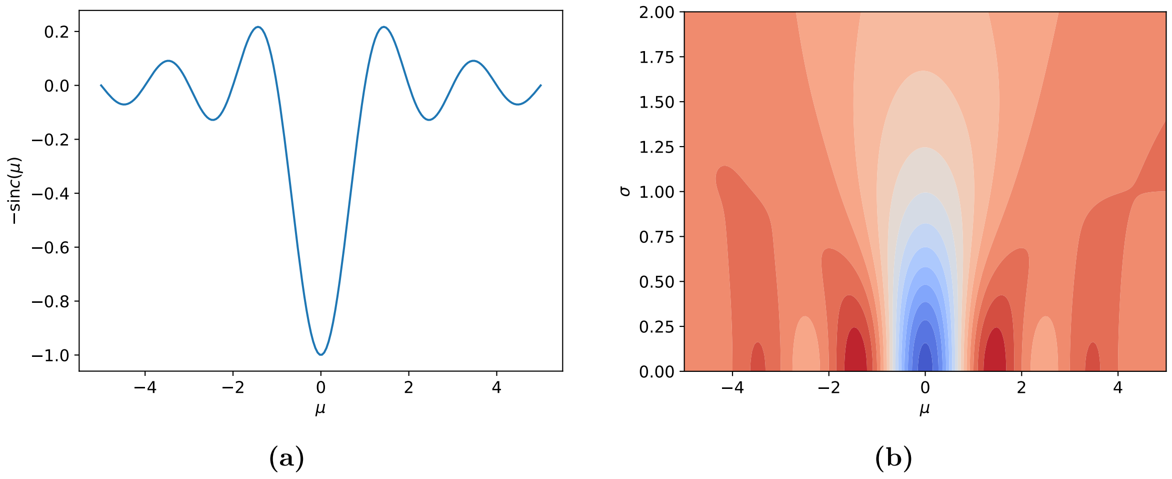


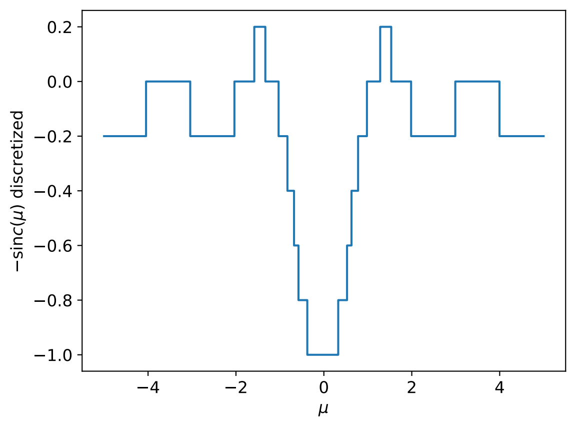
Figure 3.1: **(a)** The negative sinc function. **(b)** The contours of the Gaussian VO objective for the sinc function for different values of μ and σ. Here, the VO objective is a two-dimensional upper bounding version of the sinc function. It tends to a constant for $\sigma \to \infty$ and the sinc function for $\sigma \to 0$. The VO objective is slightly asymmetrical due to being computed by sampling. Red denotes high values, blue denotes low values and the darker the colour, the higher the absolute value. Actual values are omitted for simplicity. Figures inspired by [40].

the parameters of the search distribution. The global minimum is at $\mu = 0$ for any value of σ. The new objective still has the exact form of the original for $\sigma = 0$ and as such still has challenging local minima. However, in this specific case, minimizing the VO objective with a fixed value of σ of about 1 or larger would uniformly result in approximate[4] convergence on the global minimum regardless of initial point. Additionally, notice that around the global minimum, sinc is convex which is also true for the VO objective. In fact, for any expectation affine search distribution, which includes Gaussians, if the original objective function is convex then so is the the variational upper bound [106].

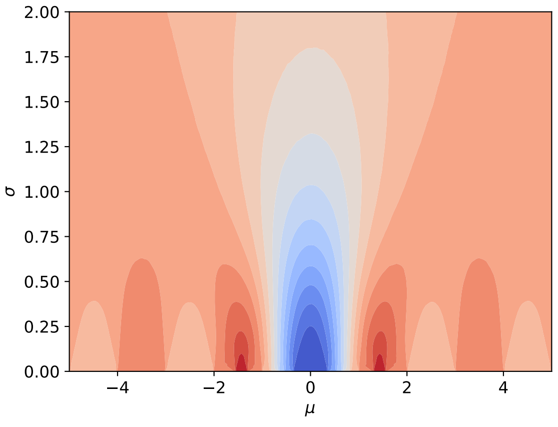
A quantized version of the sinc function is seen in Figure 3.2a. Although this function is non-differentiable, the VO bound is still differentiable for any non-zero variance as can be seen in Figure 3.2b. As the search distribution variance approaches zero, the VO objective tends towards the original objective and non-differentiability. Letting $\sigma \to \infty$ results in the VO objective approaching a constant yielding zero gradient everywhere.

A final point should be made about the smoothing of the original objective function done by VO. Figure 3.3a shows a function that has a wide local minimum and a narrow global minimum. As seen in Figure 3.3b, the VO objective has a clear preference toward the wide local minimum. The reason for this preference is that the expectation is computed based on a specific value of the variance. The larger the variance, the wider is the range of values on which the function is evaluated and the expectation computed. For large values of σ, narrow minima are simply averaged out due to the high function values for most sampled points. Thus, Gaussian VO has a bias towards minima with low curvature compared to minima with high curvature. This bias is similarly present for other search distributions. In relation to this, it is interesting to note that the local structure of the minima of NN loss surfaces has been shown

[4]The convergence will only be to an approximation of the minimum when σ is held fixed. This is discussed further in Section 3.4

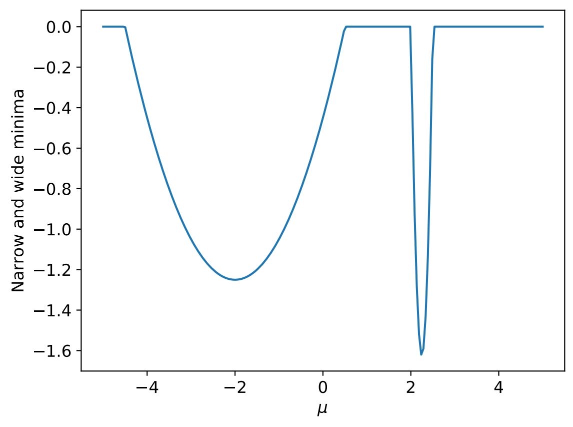


(a)

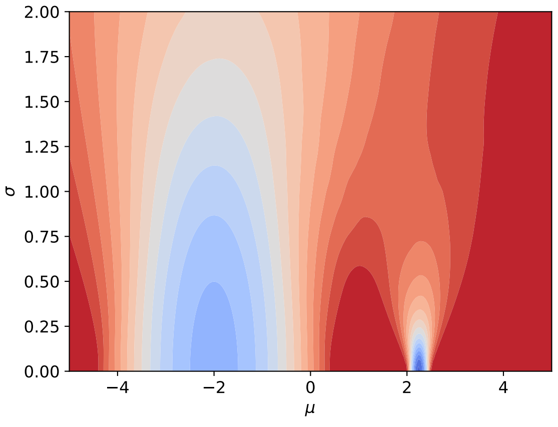


(b)

Figure 3.2: **(a)** A quantized version of the sinc function shown in Figure 3.1a. **(b)** The contours of the corresponding Gaussian VO objective for different values of μ and σ. The VO objective is a smoothed version of the original nondifferentiable objective tending towards a constant for $\sigma \to \infty$ and non-differentiability for $\sigma \to 0$. Figures inspired by [40].



(a)



(b)

Figure 3.3: **(a)** A function with a low curvature local minimum and a high curvature global minimum. **(b)** The contours of the corresponding Gaussian VO objective. The Gaussian VO objective has a different global minimum than the original function for almost all values of $\sigma > 0$. This illustrates the tendency of VO to prefer low curvature minima over high curvature minima if situated near each other. Figures inspired by [40].

to have strong connections to the ability of the resulting network to generalize well. Flatter minima have been claimed to yield better generalization than sharp although the way to define flatness in high dimensional geometries remains somewhat intangible [24].

3.3.3.3 The difference between the VO and Taylor series gradients

Following the observation that the Taylor gradient and the VO gradient are equivalent for a Gaussian perturbation/search distribution, it should be noted the two approaches are not in general equivalent for different choices of search distributions. As an example, take the Laplace distribution,

$$p(x|\mu,b) = \frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right) = \frac{1}{\sqrt{2}\sigma}\exp\left(-\frac{\sqrt{2}}{\sigma}|x-\mu|\right) \tag{3.3.23}$$

with mean μ and variance $\sigma = \sqrt{2}b$. The first three moments of the Laplace distribution are needed for the Taylor gradient estimator and can be computed as follows. Let $\epsilon = x - \mu$ from which it follows that $d\epsilon = dx$. The first moment is

$$\begin{aligned}
\mathrm{E}[x] &= \mathrm{E}[\epsilon] + \mu \\
&= \frac{1}{2b}\int_{-\infty}^{\infty} \epsilon \exp\left(-\frac{|\epsilon|}{b}\right) \mathrm{d}\epsilon + \mu \\
&= \frac{1}{2b}\left(\int_{-\infty}^{0} \epsilon \exp\left(\frac{\epsilon}{b}\right) \mathrm{d}\epsilon + \int_{0}^{\infty} \epsilon \exp\left(-\frac{\epsilon}{b}\right) \mathrm{d}\epsilon\right) + \mu \\
&= \mu\ .
\end{aligned} \tag{3.3.24}$$

Using that $\mathrm{E}[\epsilon] = 0$ as computed above, the second moment becomes

$$\begin{aligned}
\mathrm{E}\left[x^2\right] &= \mathrm{E}\left[(\epsilon+\mu)^2\right] \\
&= \mathrm{E}\left[\epsilon^2\right] + \mu^2 \\
&= \frac{1}{2b}\int_{-\infty}^{\infty} \epsilon^2 \exp\left(-\frac{|\epsilon|}{b}\right) \mathrm{d}\epsilon + \mu^2 \\
&= \frac{1}{2b}\left(\int_{-\infty}^{0} \epsilon^2 \exp\left(\frac{\epsilon}{b}\right) \mathrm{d}\epsilon + \int_{0}^{\infty} \epsilon^2 \exp\left(-\frac{\epsilon}{b}\right) \mathrm{d}\epsilon\right) + \mu^2 \\
&= \frac{1}{b}\int_{0}^{\infty} \epsilon^2 \exp\left(-\frac{\epsilon}{b}\right) \mathrm{d}\epsilon + \mu^2,
\end{aligned}$$

and substituting $v = \frac{\epsilon}{b}$, $\mathrm{d}v = \mathrm{d}\epsilon$,

$$\begin{aligned}
\mathrm{E}\left[x^2\right] &= b^2\int_{0}^{\infty} v^2 \exp\left(-v\right) \mathrm{d}\epsilon + \mu^2 \\
&= 2b^2 + \mu\ .
\end{aligned} \tag{3.3.25}$$

Finally, the third moment is

$$\begin{aligned}
\mathrm{E}\left[x^3\right] &= \mathrm{E}\left[(\epsilon+\mu)^3\right] \\
&= \mathrm{E}\left[\epsilon^3\right] + 3\mu\mathrm{E}\left[\epsilon^2\right] + 3\mu^2\mathrm{E}[\epsilon] - \mu^3 \\
&= \frac{1}{2b}\int_{-\infty}^{\infty} \epsilon^3 \exp\left(-\frac{|\epsilon|}{b}\right) \mathrm{d}\epsilon + 6b^2\mu + \mu^3 \\
&= \frac{1}{2b}\left(\int_{-\infty}^{0} \epsilon^3 \exp\left(\frac{\epsilon}{b}\right) \mathrm{d}\epsilon + \int_{0}^{\infty} \epsilon^3 \exp\left(-\frac{\epsilon}{b}\right) \mathrm{d}\epsilon\right) + 6\mu(b^2+\mu^2) \\
&= 6\mu(b^2+\mu^2)\ .
\end{aligned} \tag{3.3.26}$$

For a Laplace distributed perturbation $\epsilon \sim p(\epsilon|0, b)$, it follows that the first moment is zero, the second moment is $2b^2 = \sigma^2$ and the third moment is also zero, similarly to the Gaussian. The Taylor series gradient then becomes identical to that using a zero mean Gaussian perturbation[5],

$$f'(x) \approx \frac{1}{2Nb^2} \sum_{n=1}^{N} f(x + \epsilon_n)\epsilon_n = \frac{1}{N\sigma^2} \sum_{n=1}^{N} f(x + \epsilon_n)\epsilon_n \ , \tag{3.3.27}$$

only now with a Laplacian perturbation.

The VO gradient however differs from the Taylor gradient. The log of the Laplace probability density function (PDF) is

$$\log p(x|\mu, b) = -\log(2b) - \frac{1}{b}|x - \mu| \tag{3.3.28}$$

such that the search distribution gradient is

$$\begin{aligned} \frac{\partial}{\partial\mu} \log p(x|\mu, b) &= \frac{1}{b}\text{sgn}(x - \mu) \\ \frac{\partial}{\partial b} \log p(x|\mu, b) &= -\frac{1}{b} + \frac{1}{b^2}|x - \mu| \end{aligned} \tag{3.3.29}$$

where $\text{sgn}(\cdot)$ is the sign function. It then follows from (3.3.6) that $f'(x) = \frac{1}{b}\text{E}[f(x)\,\text{sgn}(\epsilon)]$ and by Monte Carlo sampling,

$$f'(x) \approx \frac{1}{Nb} \sum_{i=1}^{N} f(x + \epsilon_n)\,\text{sgn}(\epsilon_n) = \frac{\sqrt{2}}{N\sigma} \sum_{i=1}^{N} f(x + \epsilon_n)\,\text{sgn}(\epsilon_n) \tag{3.3.30}$$

with $\epsilon \sim p(x|0, b)$.

Clearly, the VO gradient in (3.3.30) differs from the Taylor gradient in (3.3.27) by using the sign of the perturbation rather than the perturbation itself (and a factor of $\sqrt{2}$). The Taylor gradient is slightly biased, which follows from the derivation in Section 3.2. The Taylor gradient also implicitly assumes that the objective function is differentiable although practically, this does not have to be the case when using a Monte Carlo estimator as long as the estimator converges. The VO gradient on the other hand is an unbiased estimate of the gradient of the differentiable upper bound to a possibly non-differentiable objective function defined by (3.3.1). The choice is then between a biased estimate of the gradient of the objective or an unbiased estimate of an (arbitrarily tight) upper bound of the objective.

It can be noted that for distributions with undefined moments, e.g. the heavy tailed Cauchy distribution, the Taylor series gradient estimator does not exist while the VO search gradient does. This difference is due to the Taylor series derivation relying on the existence of the moments of the chosen distribution as well as the exploitation of some of these moments being zero, e.g. the odd moments of the Gaussian.

[5]The derivation of the Gaussian perturbation Taylor gradient in Section 3.2 only exploits that the odd moments of the Gaussian search distribution are zero. As such, that derivation holds equally well for the Laplace distribution which also has odd moments equal to zero.

3.3.3.4 Multivariate Gaussian search distribution

In the most general case, the Gaussian search distribution is multivariate and parameterizes the full covariance matrix. Then $p(\mathbf{x}|\boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is given by

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \tag{3.3.31}$$

with $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$. It follows that

$$\log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \ . \tag{3.3.32}$$

For a matrix $\mathbf{A}$ and column vectors $\mathbf{a}$ and $\mathbf{b}$, the following relations exist for the gradients w.r.t. $\mathbf{A}$ of the determinant of $\mathbf{A}$ and the quadratic form $\mathbf{a}^{\mathrm{T}}\mathbf{A}\mathbf{b}$ [79, (49) and (61)].

$$\begin{aligned} \nabla_{\mathbf{A}}|\mathbf{A}| &= |\mathbf{A}|\big(\mathbf{A}^{-1}\big)^{\mathrm{T}} \\ \nabla_{\mathbf{A}}\mathbf{a}^{\mathrm{T}}\mathbf{A}\mathbf{b} &= -\mathbf{A}\mathbf{a}\mathbf{b}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}} \ . \end{aligned}$$

Using these for the computing the gradient w.r.t. $\boldsymbol{\Sigma}$, the gradients of the logarithm of the Gaussian are

$$\nabla_{\boldsymbol{\mu}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) \tag{3.3.33}$$

$$\nabla_{\boldsymbol{\Sigma}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2}\boldsymbol{\Sigma}^{-1} + \frac{1}{2}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1} \ . \tag{3.3.34}$$

As for the univariate case, a change of variables can be made such that $\mathbf{x} = \boldsymbol{\mu} + \mathbf{L}\boldsymbol{\epsilon}$ where $\mathbf{L}$ is the Cholesky factor of the covariance matrix, $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ and $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The Cholesky factor is guaranteed to exist since the covariance matrix is symmetric and positive-definite. Then (3.3.1) becomes

$$U(\boldsymbol{\mu}, \mathbf{L}) = \mathrm{E}[f(\mathbf{x})]_{\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})} = \mathrm{E}[f(\boldsymbol{\mu}+\mathbf{L}\boldsymbol{\epsilon})]_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},\mathbf{I})}. \tag{3.3.35}$$

Applying the Cholesky factorization and this change of variables, the gradients can be simplified as follows.

$$\begin{aligned} \nabla_{\boldsymbol{\mu}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= (\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1}\mathbf{L}\boldsymbol{\epsilon} \\ &= (\mathbf{L}^{\mathrm{T}})^{-1}\mathbf{L}^{-1}\mathbf{L}\boldsymbol{\epsilon} \\ &= (\mathbf{L}^{-1})^{\mathrm{T}}\boldsymbol{\epsilon} \end{aligned} \tag{3.3.36}$$

and

$$\begin{aligned} \nabla_{\boldsymbol{\Sigma}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) &= -\frac{1}{2}(\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1} + \frac{1}{2}(\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1}\mathbf{L}\boldsymbol{\epsilon}(\mathbf{L}\boldsymbol{\epsilon})^{\mathrm{T}}(\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1} \\ &= -\frac{1}{2}(\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1} + \frac{1}{2}(\mathbf{L}^{\mathrm{T}})^{-1}\mathbf{L}^{-1}\mathbf{L}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{L}^{\mathrm{T}}(\mathbf{L}^{\mathrm{T}})^{-1}\mathbf{L}^{-1} \\ &= -\frac{1}{2}(\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1} + \frac{1}{2}(\mathbf{L}^{\mathrm{T}})^{-1}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{L}^{-1} \ . \end{aligned} \tag{3.3.37}$$

With these gradients, the VO gradient estimator becomes

$$\begin{aligned} \nabla_{\boldsymbol{\mu}}U(\boldsymbol{\mu}, \mathbf{L}) &= (\mathbf{L}^{-1})^{\mathrm{T}}\mathrm{E}[f(\boldsymbol{\mu}+\mathbf{L}\boldsymbol{\epsilon})\boldsymbol{\epsilon}] \approx \frac{(\mathbf{L}^{-1})^{\mathrm{T}}}{N}\sum_{n=1}^{N} f(\boldsymbol{\mu}+\mathbf{L}\boldsymbol{\epsilon}_n)\boldsymbol{\epsilon}_n \\ \nabla_{\boldsymbol{\Sigma}}U(\boldsymbol{\mu}, \mathbf{L}) &= \mathrm{E}\Big[f(\boldsymbol{\mu}+\mathbf{L}\boldsymbol{\epsilon})\Big(-\frac{1}{2}(\mathbf{L}\mathbf{L}^{\mathrm{T}})^{-1} + \frac{1}{2}(\mathbf{L}^{\mathrm{T}})^{-1}\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\mathbf{L}^{-1}\Big)\Big] \ . \end{aligned} \tag{3.3.38}$$

Using the full covariance matrix will result in the most flexible Gaussian search distribution. However, too much flexibility could be harmful as it runs the risk of overfitting the search distribution [63]. Additionally, the full covariance matrix can be inhibitingly large to use with its $d(d+1)/2$ parameters. If optimizing in high dimensional spaces, such as the parameter spaces of NNs, it is computationally infeasible as these can have d anywhere from thousands to hundreds of millions. This infeasibility arises, if not as a memory problem, then due to the computational time complexity of matrix inversion being cubic as a function of matrix dimension, d. Another concern for full covariance matrices is sample efficiency. Due to the large number of parameters of the covariance matrix, obtaining a good estimate of its gradient may require many function evaluations, which can be costly. Therefore, the optimization algorithm may not have time enough to adapt the search distribution well [94]. Finally, sampling from a Gaussian distribution with d very large can be slow due to the large matrix vector product, $\mathbf{L}\boldsymbol{\epsilon}$. Thus, for high dimensional search spaces, alternatives to the full covariance are needed.

3.3.3.5 Isotropic Gaussian search distribution

The most radical alternative to the full covariance is to use an isotropic parameterization of the covariance matrix. An isotropic parameterization is given by a single variance repeated in every dimension with no covariances, $\boldsymbol{\Sigma} = \sigma^2\mathbf{I}$. In this case, $p(\mathbf{x}|\boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I})$ is called an isotropic Gaussian distribution and can be written as

$$\begin{aligned}\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I}) &= \frac{1}{(2\pi)^{d/2}|\sigma^2\mathbf{I}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}(\sigma^2\mathbf{I})^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \\ &= \frac{1}{(2\pi)^{d/2}\sigma^d} \exp\left(-\frac{1}{2\sigma^2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu})\right)\end{aligned} \tag{3.3.39}$$

since

$$\left|\sigma^2\mathbf{I}\right|^{1/2} = \sqrt{\prod_{i=1}^{d}\sigma^2} = \sqrt{\sigma^{2d}} = \sigma^d \ . \tag{3.3.40}$$

The log-PDF is

$$\log\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I}) = -\frac{d}{2}\log(2\pi) - \frac{d}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu}) \ . \tag{3.3.41}$$

Again, let $\mathbf{x} = \boldsymbol{\mu} + \sigma^2\boldsymbol{\epsilon}$ so (3.3.1) becomes

$$U(\boldsymbol{\mu}, \sigma^2) = \mathrm{E}[f(\mathbf{x})]_{\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\sigma^2\mathbf{I})} = \mathrm{E}[f(\boldsymbol{\mu}+\sigma\boldsymbol{\epsilon})]_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},\mathbf{I})} \ . \tag{3.3.42}$$

It follows that

$$\begin{aligned}\nabla_{\boldsymbol{\mu}}\log\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I}) &= \frac{1}{\sigma^2}(\mathbf{x}-\boldsymbol{\mu}) = \frac{1}{\sigma}\boldsymbol{\epsilon} \\ \nabla_{\sigma^2}\log\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I}) &= -\frac{d}{2\sigma^2} + \frac{1}{2\sigma^4}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu}) = \frac{1}{2\sigma^2}\left(\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon} - d\right) \ .\end{aligned} \tag{3.3.43}$$

By (3.3.6), the gradient of the VO objective with an isotropic Gaussian search distribution is

$$\begin{aligned}\nabla_{\boldsymbol{\mu}}U(\boldsymbol{\mu}, \sigma^2) &= \frac{1}{\sigma}\mathrm{E}[f(\boldsymbol{\mu}+\sigma\boldsymbol{\epsilon})\boldsymbol{\epsilon}] \approx \frac{1}{N\sigma}\sum_{n=1}^{N} f(\boldsymbol{\mu}+\sigma\boldsymbol{\epsilon}_n)\boldsymbol{\epsilon}_n \\ \nabla_{\boldsymbol{\sigma}^2}U(\boldsymbol{\mu}, \sigma^2) &= \frac{1}{2\sigma^2}\mathrm{E}\big[f(\boldsymbol{\mu}+\sigma\boldsymbol{\epsilon})\left(\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon} - d\right)\big] \approx \frac{1}{2N\sigma^2}\sum_{n=1}^{N} f(\boldsymbol{\mu}+\sigma\boldsymbol{\epsilon}_n)\left({\boldsymbol{\epsilon}_n}^{\mathrm{T}}\boldsymbol{\epsilon}_n - d\right) \ .\end{aligned} \tag{3.3.44}$$

This is evidently a simple generalization of univariate case in (3.3.18) to the d-dimensional case while keeping the variance scalar. Compared to the general Gaussian search distribution, the isotropic Gaussian has only a single variance parameter to be optimized. As a consequence, it is much less memory intensive and sampling can be done in $O(d)$ time. However, the distribution captures no covariance between the dimensions being searched and the smoothing of the objective is the same in all dimensions.

3.3.3.6 Separable Gaussian search distribution

A more flexible version of the Gaussian search distribution compared to the isotropic special case is the separable Gaussian. Similarly to the isotropic Gaussian, the separable Gaussian captures no covariance information between the dimensions. Instead, it parameterizes the covariance matrix by d individual variance parameters along its diagonal. The PDF is defined as for the multivariate Gaussian in (3.3.31) but with $\boldsymbol{\Sigma} = \operatorname{diag}\big(\sigma_1^2, \sigma_2^2, \ldots, \sigma_d^2\big)$. Let

$$\boldsymbol{\sigma}^2 = \begin{bmatrix}\sigma_1^2 & \sigma_2^2 & \cdots & \sigma_d^2\end{bmatrix}^{\mathrm{T}} \tag{3.3.45}$$

be the parameter vector holding the variances with squaring applied elementwise. Then

$$\boldsymbol{\Sigma} = \operatorname{diag}\big(\boldsymbol{\sigma}^2\big) \tag{3.3.46}$$

where $\mathbf{e} = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}^{\mathrm{T}}$ and the $\odot$ operator denotes elementwise multiplication. Inserting this expression for $\boldsymbol{\Sigma}$ into the log-PDF of the Gaussian in (3.3.32) gives the logarithm of the separable Gaussian PDF which can be simplified by noting,

$$\begin{aligned}\log|\boldsymbol{\Sigma}| &= \log\left|\operatorname{diag}\big(\boldsymbol{\sigma}^2\big)\right| \\ &= \log\prod_{i=1}^{d}\sigma_i^2 \\ &= \sum_{i=1}^{d}\log\sigma_i^2 \\ &= \mathbf{e}^{\mathrm{T}}\log\left(\boldsymbol{\sigma}^2\right) . \end{aligned} \tag{3.3.47}$$

Likewise for the quadratic form,

$$(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}) = (\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\operatorname{diag}\big(\boldsymbol{\sigma}^{-2}\big)(\mathbf{x}-\boldsymbol{\mu}) \tag{3.3.48}$$

$$\begin{aligned} &= \sum_{i=1}^{d}\frac{(x_i-\mu_i)^2}{\sigma_i^2} \\ &= \big(\boldsymbol{\sigma}^{-2}\big)^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu})^2 . \end{aligned} \tag{3.3.49}$$

Here, the inverse squaring of $\boldsymbol{\sigma}$ is applied elementwise, $\boldsymbol{\sigma}^{-2} = \begin{bmatrix}\sigma_1^{-2} & \sigma_2^{-2} & \ldots & \sigma_d^{-2}\end{bmatrix}^{\mathrm{T}}$. From (3.3.47) and (3.3.49) it follows that

$$\log\mathcal{N}\big(\mathbf{x}|\boldsymbol{\mu},(\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}})\odot\mathbf{I}\big) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\mathbf{e}^{\mathrm{T}}\log\left(\boldsymbol{\sigma}^2\right) - \frac{1}{2}\big(\boldsymbol{\sigma}^{-2}\big)^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu})^2 . \tag{3.3.50}$$

As before, reparameterize $\mathbf{x} = \boldsymbol{\mu} + \boldsymbol{\sigma}\odot\boldsymbol{\epsilon}$ so

$$U(\boldsymbol{\mu},\sigma^2) = \mathrm{E}[f(\mathbf{x})]_{\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},(\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}})\odot\mathbf{I})} = \mathrm{E}[f(\boldsymbol{\mu}+\boldsymbol{\sigma}\odot\boldsymbol{\epsilon})]_{\mathcal{N}(\boldsymbol{\epsilon}|\mathbf{0},\mathbf{I})} . \tag{3.3.51}$$

Using the quadratic form in (3.3.48), the gradient w.r.t. $\boldsymbol{\mu}$ is

$$\begin{aligned}\nabla_{\boldsymbol{\mu}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, (\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}}) \odot \mathbf{I}) &= \frac{1}{2}\nabla_{\boldsymbol{\mu}}\big(\boldsymbol{\sigma}^{-2}\big)^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu})^2 \\ &= \boldsymbol{\sigma}^{-2} \odot (\mathbf{x}-\boldsymbol{\mu}) \\ &= \boldsymbol{\sigma}^{-1} \odot \boldsymbol{\epsilon} \ . \end{aligned} \tag{3.3.52}$$

Using the quadratic form (3.3.49), the gradient w.r.t. $\boldsymbol{\sigma}^2$ is

$$\begin{aligned}\nabla_{\boldsymbol{\sigma}^2} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, (\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}}) \odot \mathbf{I}) &= -\frac{1}{2}\nabla_{\boldsymbol{\sigma}^2}\mathbf{e}^{\mathrm{T}}\log\boldsymbol{\sigma}^2 - \frac{1}{2}\nabla_{\boldsymbol{\sigma}^2}\big(\boldsymbol{\sigma}^{-2}\big)^{\mathrm{T}}(\mathbf{x}-\boldsymbol{\mu})^2 \\ &= -\frac{1}{2}\boldsymbol{\sigma}^{-2} + \frac{1}{2}\mathrm{diag}\big(\boldsymbol{\sigma}^{-4}\big)(\mathbf{x}-\boldsymbol{\mu})^2 \\ &= -\frac{1}{2}\boldsymbol{\sigma}^{-2} + \frac{1}{2}\boldsymbol{\sigma}^{-4} \odot (\mathbf{x}-\boldsymbol{\mu})^2 \\ &= -\frac{1}{2}\boldsymbol{\sigma}^{-2} \odot \big(\boldsymbol{\epsilon}^2 - 1\big) \ . \end{aligned} \tag{3.3.53}$$

By (3.3.6), the gradient of the VO objective becomes

$$\nabla_{\boldsymbol{\mu}} U(\boldsymbol{\mu}, \sigma^2) = \boldsymbol{\sigma}^{-1} \odot \mathrm{E}[f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon})\boldsymbol{\epsilon}] \approx \boldsymbol{\sigma}^{-1} \odot \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}_n)\boldsymbol{\epsilon}_n \tag{3.3.54}$$

$$\nabla_{\sigma}^2 U(\boldsymbol{\mu}, \sigma^2) = \frac{1}{2}\boldsymbol{\sigma}^{-2} \odot \mathrm{E}\big[f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon})\big(\boldsymbol{\epsilon}^2 - 1\big)\big] \approx \boldsymbol{\sigma}^{-2} \odot \frac{1}{2N}\sum_{n=1}^{N} f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}_n)\big(\boldsymbol{\epsilon}_n^2 - 1\big) \ .$$

Note that these gradients are in fact the gradients of the univariate Gaussian in each dimension of the separable Gaussian. That this must be the case can also be seen by rewriting the d dimensional separable Gaussian as a product of univariate Gaussians as follows.

$$\begin{aligned}\mathcal{N}\big(\mathbf{x}|\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2)\big) &= \frac{1}{(2\pi)^{d/2}|\mathrm{diag}(\boldsymbol{\sigma}^2)|^{1/2}}\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\mathrm{diag}(\boldsymbol{\sigma}^2)^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \\ &= \frac{1}{(2\pi)^{d/2}\prod_{i=1}^{d}\sigma_i}\exp\left(-\frac{1}{2}\begin{bmatrix} x_1-\mu_1 \\ \vdots \\ x_d-\mu_d\end{bmatrix}^{\mathrm{T}}\begin{bmatrix}\sigma_1^{-2} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_d^{-2}\end{bmatrix}\begin{bmatrix} x_1-\mu_1 \\ \vdots \\ x_d-\mu_d\end{bmatrix}\right) \\ &= \frac{1}{\prod_{i=1}^{d}(2\pi)^{1/2}\sigma_i}\exp\left(-\sum_{i=1}^{d}\frac{(x_i-\mu_i)^2}{2\sigma_i^2}\right) \\ &= \frac{1}{\prod_{i=1}^{d}\sqrt{2\pi}\sigma_i}\prod_{i=1}^{d}\exp\left(-\frac{(x_i-\mu_i)^2}{2\sigma_i^2}\right) \\ &= \prod_{i=1}^{d}\frac{1}{\sqrt{2\pi}\sigma_i}\exp\left(-\frac{(x_i-\mu_i)^2}{2\sigma_i^2}\right) \\ &= \prod_{i=1}^{d}\mathcal{N}(x_i|\mu_i,\sigma_i^2) \ . \end{aligned} \tag{3.3.55}$$

Then

$$\log \mathcal{N}\big(\mathbf{x}|\boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}^2)\big) = \log \prod_{i=1}^{d} \mathcal{N}(x_i|\mu_i, \sigma_i^2) = \sum_{i=1}^{d} \log \mathcal{N}(x_i|\mu_i, \sigma_i^2) \tag{3.3.56}$$

which is simply a sum of univariate Gaussians. Since each dimension is independent from the others, taking the derivative of $\log \mathcal{N}(\mathbf{x}|\mu, \boldsymbol{\Sigma})$ w.r.t. $\boldsymbol{\sigma}$ simply returns the derivative of a univariate Gaussian in each dimension as in (3.3.54).

Compared to the isotropic Gaussian, the separable Gaussian has d variances rather than one which allows greater flexibility during the search and potentially faster convergence. Compared to the general multivariate Gaussian, the number of parameters is linear in the dimension rather than quadratic.

Using VO with a separable Gaussian search distribution to train an NN requires storing the weights as the mean parameter and as many variance parameters as there are weights. This significantly adds to the number of parameters held in memory during training but this is possible for many network models which already store the weights, their gradients and potentially associated momentum buffers. An alternative approach is to consider only a subset of the trained NN weights as independent dimensions. One way to do this is to define each network layer as a dimension and parameterize a unique variance only for every layer. Perturbations are then sampled from the corresponding dimension of the search distribution for each weight of the specific layer. These approaches will be examined in Chapter 4.

3.3.3.7 Low rank covariance matrix approximation

Low rank approximations of the covariance matrix allows for a significant reduction of the number of parameters from being quadratic in dimension to being linear in dimension. Such approximations have been applied several places such as for training Gaussian Mixture Models [63]. However, much of the previous work is concerned with optimization of the low rank approximation by maximum likelihood and the likes of it with the goal of fitting a model [4, 20, 114]. This is somewhat different from the case of variational optimization where the covariance matrix is not fitted per say but rather updated continuously to give the best samples for making progress in some optimization problem. This section introduces the truncated SVD approximation which is one way of forming a theoretical background for low rank matrix approximations.

Any general, real valued matrix $\mathbf{A}$ can be decomposed as

$$\underbrace{\mathbf{A}}_{N\times D} = \underbrace{\mathbf{U}}_{N\times N} \underbrace{\mathbf{S}}_{N\times D} \underbrace{\mathbf{V}^{\mathrm{T}}}_{D\times D} \tag{3.3.57}$$

where $\mathbf{U}$ has orthonormal columns known as the left singular vectors, $\mathbf{S}$ is a diagonal matrix holding the D (assuming $N > D$) singular values and $N-D$ zeros and $\mathbf{V}$ holds the orthonormal right singular vectors in its columns, similarly to $\mathbf{U}$. This is the singular value decomposition (SVD) which can be seen as a generalized eigendecomposition and has close ties to principal components analysis (PCA) [70].

For non-square matrices, the so-called economy-sized SVD exploits the fact that for $N > D$ the $N - D$ last columns of $\mathbf{U}$ are multiplied by zeros in $\mathbf{S}$ and thus omits these giving

$$\underbrace{\mathbf{A}}_{N\times D} = \underbrace{\hat{\mathbf{U}}}_{N\times D} \underbrace{\hat{\mathbf{S}}}_{D\times D} \underbrace{\hat{\mathbf{V}}^{\mathrm{T}}}_{D\times D} . \tag{3.3.58}$$

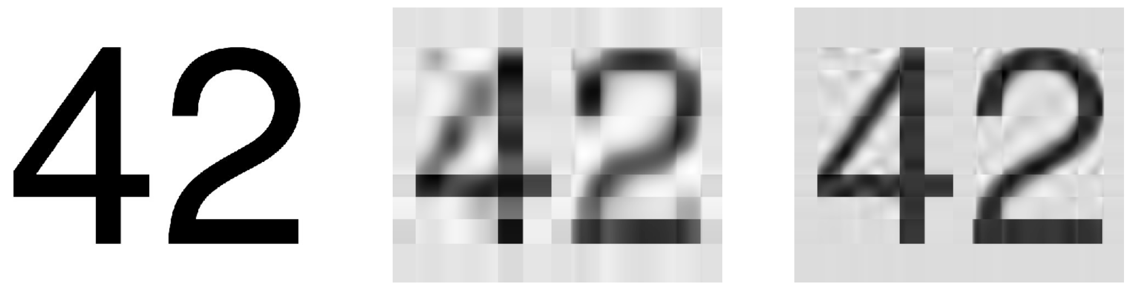


Figure 3.4: **(a)** An example image of $500 \times 600 = 300.000$ pixels. **(b)** and **(c)** Respectively, rank-5 and 10 approximations of the image. The low rank approximations are constructed from 5.505 and 11.010 parameters, respectively, reduced by a factor of about 56 and 28 compared to the original. The high level of structure on the original image results in few large singular values which in turn gives high quality low rank approximations. Note how the axis aligned parts of the "4" and "2" are more easily reconstructed.

When the singular values are sorted in descending order in $\mathbf{S}$, the best possible low-rank approximation of the matrix $\mathbf{A}$ can be shown to be given by

$$\mathbf{A} \approx \mathbf{A}_L = \mathbf{U}_{:,1:L}\mathbf{S}_{1:L,1:L}\mathbf{V}^{\mathrm{T}}{}_{:,1:L}, \tag{3.3.59}$$

for some chosen rank L. This can also be written as a sum of outer products of the L left and right singular vectors weighted by the corresponding L largest singular values

$$\mathbf{A} \approx \mathbf{A}_L = \sum_{i=1}^{L} s_i \mathbf{u}_i \mathbf{v}^{\mathrm{T}}{}_i \ . \tag{3.3.60}$$

If the singular values quickly reduce in value, this truncated SVD can be a good approximation for small values of L. Furthermore, the original matrix requires storing ND elements while the approximation requires $NL + L + DL = L(N + D + 1)$ which is significantly smaller than ND for small values of L [70].

The low-rank approximation is illustrated in Figures 3.4 and 3.5. In Figure 3.4, an image and its rank-5 and rank-10 approximations are shown while the the singular values of the image matrix are plotted in Figure 3.5a. Figure 3.5b shows the MSE of the rank L approximation matrix $\mathbf{A}_L$ compared to the original matrix.

In the context of Gaussian search distributions, the matrix of interest is the covariance matrix which has the additional properties of being symmetric and square and is required to be positive definite. Inspired by the low-rank approximation presented above, a rank-L approximation to the covariance matrix $\boldsymbol{\Sigma}$ can be written as [34]

$$\boldsymbol{\Sigma} \approx \mathbf{D} + \sum_{i=1}^{L} \mathbf{u}_i \mathbf{u}_i^{\mathrm{T}} = \mathbf{D} + \mathbf{U}^{\mathrm{T}}\mathbf{U} \tag{3.3.61}$$

where $\mathbf{u}_i$ are d dimensional column vectors and $\mathbf{U} = \begin{bmatrix} u_1{}^{\mathrm{T}} & \cdots & u_L{}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}$ has the $\mathbf{u}_i$ vectors in its rows. $\mathbf{D}$ is a diagonal matrix which is chosen large enough that $\boldsymbol{\Sigma}$ is positive definite.

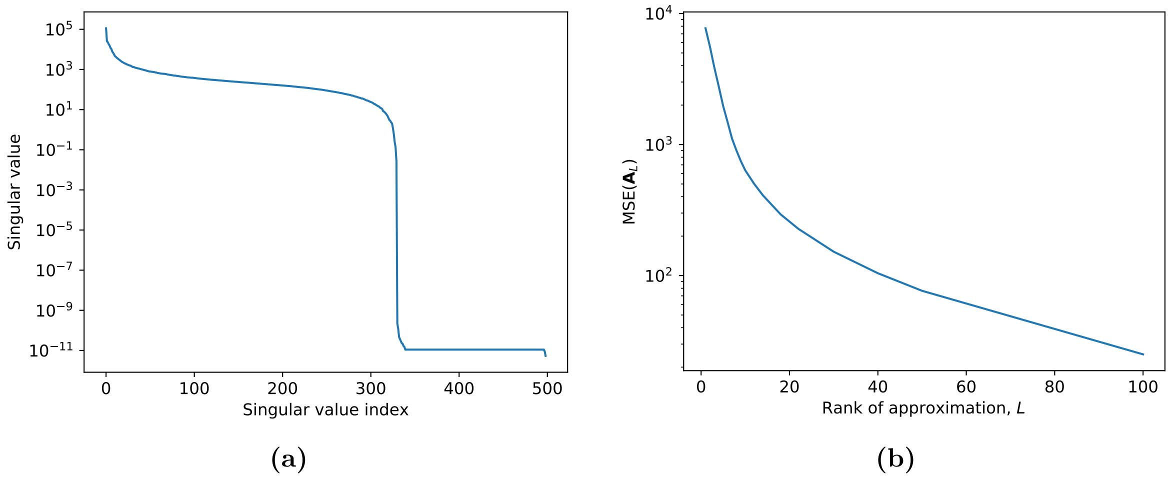


Figure 3.5: **(a)** Singular values of the image in Figure 3.4a as a function of index. **(b)** MSE of different rank approximations of that image. Note the rapid initial decrease in the singular values driving a steep decrease in the MSE over the first 10 singular values.

This approximation has dL parameters rather than $d(d+1)/2$ which is considerably fewer for $L \ll d$.

It seems fair to conjecture that a good approximation might be achievable in this manner for a choice of $L \ll d$ in the case where the search space is the parameter space of an NN with d up to hundreds of millions. This thesis has not studied this avenue further.

3.3.3.8 Cauchy distribution

For completeness, this section briefly considers the Cauchy distribution for use in VO. Since the Cauchy is a heavy-tailed distribution it can be used for a so-called hill-climber version of VO where the population size is 1 [94].

The PDF of the Cauchy is

$$p(x|\mu,\gamma) = \frac{1}{\pi\gamma}\left[1+\left(\frac{x-\mu}{\gamma}\right)^2\right]^{-1} = \frac{1}{\pi\gamma}\left[\frac{\gamma^2}{(x-\mu)^2+\gamma^2}\right] \tag{3.3.62}$$

with log-PDF

$$\log p(x|\mu,\gamma) = -\log(\pi\gamma) - \log\left[1+\left(\frac{x-\mu}{\gamma}\right)^2\right] . \tag{3.3.63}$$

At each iteration, a single perturbation is made using the Cauchy and its performance compared to the unperturbed model. If the perturbed model is better it is used as the unperturbed model in the next iteration, otherwise it is disregarded. Hill-climber VO can be better at avoiding getting stuck in local minima. However, due to the rarity of these in the loss surfaces of NNs this method is not considered further in this thesis.

3.4 Natural gradient

This section first discusses the problems related to using the regular search gradient in VO in Section 3.4.1. It then discusses how to measure distance or similarity between probability distributions in Section 3.4.2 and and introduces the Kullback-Leibler (KL) divergence in Section 3.4.3. The FIM is derived in Section 3.4.4 as a second order approximation to the KL divergence. Following this, the natural gradient is derived in Section 3.4.5 as the optimal search direction when optimizing the variational upper bound subject to a dissimilarity constraint based on the KL divergence between the updated and previous search distributions at every iteration of gradient descent. The natural gradient is then computed for the case of a Gaussian search distribution and finally, the benefits of it are demonstrated for a simple problem.

3.4.1 Problems of regular search gradients

A central problem of regular search gradients is their inability to precisely locate any optima, even a convex quadratic one. Take for instance a univariate Gaussian to be the search distribution. In order to locate an approximately quadratic optima, σ must go to zero in order to place all probability mass at the optimal point, x^*. However, the search gradients in (3.3.17) and the upper bound gradients in (3.3.18) are not numerically stable for very small values of σ: As $\sigma \rightarrow 0$ clearly $1/\sigma^2 \rightarrow \infty$ and so does the estimators. Although $\mathrm{E}\left[\epsilon f(x+\epsilon)\right] \rightarrow 0$ for $\sigma \rightarrow 0$, the variance of the gradient estimator, derived from the Taylor series in (3.2.13), can be seen to go to infinity.

Thus, the behaviour of VO with regular search gradients will be to first adjust σ to find a local attractor. When a local attractor has been found, the algorithm steps towards it, gradually decreasing the value of σ as the objective function becomes approximately quadratic around the attractor. As $\sigma \rightarrow 0$ the gradients become too large and the computed update shoots away from the attractor, effectively restarting the search. Even though not quite as dramatic, the opposite case of $\sigma \gg 1$, which would arise at a large plateau, results in insignificant updates to the mean parameter and can completely halt progress [118].

This behaviour is completely opposite of what is intuitively wanted of a VO estimator. When in a flat region, σ should be increased and large steps made in any promising direction. When closing in on an attractor, the step sizes should ideally go to zero. This highlights the need for adjusting the gradient to have better numerics for small values of the variance.

3.4.2 The distance between successive search distributions

The regular gradient is a vector that points in the direction of the steepest increase of the objective function. The steepest direction is understood in terms of Euclidean distance such that the gradient gives the direction where the smallest total change in the parameter gives the largest change in the function value with changes measured by Euclidean distance. It is evident that if Euclidean distance between optimized parameters is not a suitable measure of the resulting variation in the objective function, then the regular gradient gives a suboptimal direction in the search space.

In the VO setting, a search distribution is maintained over the network weights and it is updated iteratively using gradient descent. Evaluating the closeness of successive search distributions by computing the Euclidean distance between their parameter vectors can be shown to be give counter intuitive results. Take for instance the two pairs of univariate Gaussians

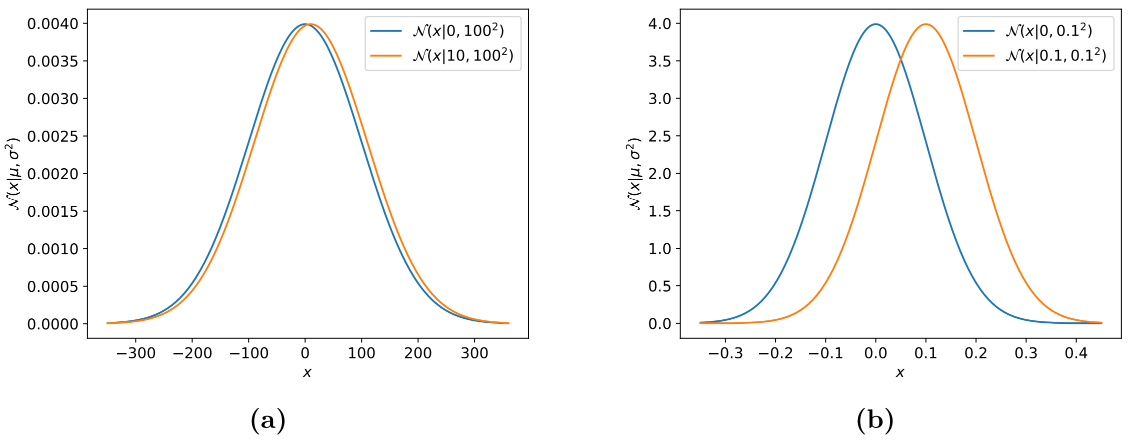


Figure 3.6: Two pairs of univariate Gaussians, P_1 in **(a)** and P_2 in **(b)**. The P_1 pair is obviously very similar while the P_2 pair are somewhat dissimilar. A suitable measure of the dissimilarity of two probability distributions is expected to reflect this.

shown in Figure 3.6,

$$P_1 = \{\mathcal{N}(0, 100^2), \mathcal{N}(10, 100^2)\}$$

and

$$P_2 = \{\mathcal{N}(0, 0.1^2), \mathcal{N}(0.1, 0.1^2)\} .$$

Due to their large variance, the first two distributions are almost identical while the other pair of distributions share limited support. As a consequence of their parameterizations however, the Euclidean distance, D_2, between the first pair is

$$D_2\big(\mathcal{N}(0, 100^2), \mathcal{N}(10, 100^2)\big) = \sqrt{(0-10)^2 + (100^2 - 100^2)^2} = 10$$

whereas the distance between the second pair is

$$D_2\big(\mathcal{N}(0, 0.1^2), \mathcal{N}(0.1, 0.1^2)\big) = \sqrt{(0-0.1)^2 + (0.1^2 - 0.1^2)^2} = 0.1,$$

a factor of 1000 lower.

That Euclidean distance is an unsuited measure of closeness between probability distributions can be theoretically attributed to distributions not residing in a Euclidean space but rather on a Riemannian manifold (or statistical manifold) [113]. Although this thesis will not delve further into the details of these manifolds, measuring distance within them must be done with respect to a form of distance metric. This is akin to relating a line segment ds in Euclidean space to changes dx and dy in the x and y directions by the relation $ds = \sqrt{dx^2 + dy^2}$ only here, ds is a line segment on the statistical manifold and the directions represent parameters of a family of probability distributions, e.g. μ and σ of the univariate Gaussian [113].

3.4.3 The Kullback-Leibler divergence

A better and more natural measure of distance between two probability distributions is the Kullback-Leibler (KL) divergence. It is defined as [8]

$$\begin{aligned} D_{\mathrm{KL}}(p||q) &= -\int p(\mathbf{x}) \log q(\mathbf{x})\, \mathrm{d}\mathbf{x} - \left(-\int p(\mathbf{x}) \log p(\mathbf{x})\, \mathrm{d}\mathbf{x}\right) \\ &= -\int p(\mathbf{x}) \log\left(\frac{q(\mathbf{x})}{p(\mathbf{x})}\right) \mathrm{d}\mathbf{x}\ . \end{aligned} \tag{3.4.1}$$

The KL divergence satisfies $D_{\mathrm{KL}}(p||q) \geq 0$ with equality only when $p(\mathbf{x}) = q(\mathbf{x})$ and it is asymmetrical such that $D_{\mathrm{KL}}(p||q) \neq D_{\mathrm{KL}}(q||p)$. As such is it strictly speaking not a measure of distance but should rather be interpreted as a measure of information loss by using q rather than p or, inversely, a measure of required additional information by using q rather than p, to encode some amount of information. For example, in variational Bayesian methods, the KL divergence is often used as the measure of dissimilarity between a distribution $q(\mathbf{z})$ that approximates a posterior distribution $p(\mathbf{z}|\mathbf{x})$ where $\mathbf{x}$ and $\mathbf{z}$ are respectively observed and unobserved (latent) variables. In fact, $q(\mathbf{z})$ approximates Bayes' theorem

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})} \tag{3.4.2}$$

in which the marginalization over $\mathbf{z}$ to compute $p(\mathbf{x})$ is often intractable. The approximation is computed by minimizing the KL divergence of the approximate posterior between the true posterior, $D_{\mathrm{KL}}(q||p)$. The KL divergence can then be seen as the average minimal additional amount of information required to specify the value of $\mathbf{z}$ as a result of using $q(\mathbf{z})$ instead of $p(\mathbf{z}|\mathbf{x})$[6] [8].

A symmetrized version of the KL divergence which satisfies $D_{\mathrm{KL}}(p, q) = D_{\mathrm{KL}}(q, p)$ can be defined as

$$\begin{aligned} D_{\mathrm{KL}}(p, q) &= D_{\mathrm{KL}}(p||q) + D_{\mathrm{KL}}(q||p) \\ &= -\int p(\mathbf{x}) \log\left(\frac{q(\mathbf{x})}{p(\mathbf{x})}\right) \mathrm{d}\mathbf{x} - \int q(\mathbf{x}) \log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) \mathrm{d}\mathbf{x} \\ &= \int p(\mathbf{x}) \log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) \mathrm{d}\mathbf{x} - \int q(\mathbf{x}) \log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) \mathrm{d}\mathbf{x} \\ &= \int (p(\mathbf{x}) - q(\mathbf{x})) \log\left(\frac{p(\mathbf{x})}{q(\mathbf{x})}\right) \mathrm{d}\mathbf{x}\ . \end{aligned} \tag{3.4.3}$$

The KL divergence and the symmetrized KL divergence between two univariate Gaussians, $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$, are[7]

$$D_{\mathrm{KL}}\big(\mathcal{N}(\mu_1, \sigma_1^2)||\mathcal{N}(\mu_2, \sigma_2^2)\big) = \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2 + (\mu_1 + \mu_2)^2}{2\sigma_2^2} - \frac{1}{2} \tag{3.4.4}$$

$$D_{\mathrm{KL}}\big(\mathcal{N}(\mu_1, \sigma_1^2), \mathcal{N}(\mu_2, \sigma_2^2)\big) = \big((\mu_1 - \mu_2)^2 + (\sigma_1^2 + \sigma_2^2)\big)\left(\frac{1}{2\sigma_1^2} + \frac{1}{2\sigma_2^2}\right) - 2\ . \tag{3.4.5}$$

[6]The information content is measured in bits if the base 2 logarithm is used and in nats if the natural logarithm is used. The relation between the two is $1\,\mathrm{nat} = \frac{1}{\ln 2}\mathrm{bit} \approx 1.44\,\mathrm{bit}$.

[7]For derivations, see Appendix E

For the pairs, P_1 and P_2 considered above, the KL divergence and symmetrized KL divergence become

$$D_{\mathrm{KL}}\big(\mathcal{N}(0,100^2)||\mathcal{N}(10,100^2)\big) = 0.005$$
$$D_{\mathrm{KL}}\big(\mathcal{N}(0,0.1^2)||\mathcal{N}(0.1,0.1^2)\big) = 0.5$$

and

$$D_{\mathrm{KL}}\big(\mathcal{N}(0,100^2),\mathcal{N}(10,100^2)\big) = 0.01$$
$$D_{\mathrm{KL}}\big(\mathcal{N}(0,0.1^2),\mathcal{N}(0.1,0.1^2)\big) = 1 \ ,$$

respectively. The ordering of the distributions in the regular KL divergence is unimportant in this specific case since the asymmetry is in the variances and the variances are equal here. In both cases, this measure of dissimilarity is a factor of 100 lower for the P_1 pair compared to the P_2 pair. Evidently, both the regular and symmetrized KL divergence much better represent similarity between distributions than the Euclidean measure which, contrary to intuition, considered the P_1 pair to be much further apart than P_2.

3.4.4 The Fisher information matrix

For the purposes of the following section, this section will consider how to compute an approximation to the KL divergence. For two distributions, $p(\mathbf{x}|\boldsymbol{\theta})$ and $p(\mathbf{x}|\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$, that differ by some small vector $\Delta\boldsymbol{\theta}$ in their parameter vectors, a second order Taylor approximation to their symmetric KL divergence can be computed as follows[8]. Let

$$\Delta p(\mathbf{x}|\boldsymbol{\theta}) = p(\mathbf{x}|\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) - p(\mathbf{x}|\boldsymbol{\theta})$$

and rewrite the symmetric KL divergence as follows

$$\begin{aligned} D_{\mathrm{KL}}(p(\mathbf{x}|\boldsymbol{\theta}+\Delta\boldsymbol{\theta}),p(\mathbf{x}|\boldsymbol{\theta})) &= \int (p(\mathbf{x}|\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) - p(\mathbf{x}|\boldsymbol{\theta}))\log\left(\frac{p(\mathbf{x}|\boldsymbol{\theta}+\Delta\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right)\mathrm{d}\mathbf{x} \\ &= \int \Delta p(\mathbf{x}|\boldsymbol{\theta})\log\left(1+\frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right)\mathrm{d}\mathbf{x} \\ &= \int p(\mathbf{x}|\boldsymbol{\theta})\frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\log\left(1+\frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right)\mathrm{d}\mathbf{x} \ . \end{aligned} \tag{3.4.6}$$

Now, the Taylor expansion of the logarithm is

$$\log(1+x) = \log(1) + x + \mathcal{O}(x^2) \tag{3.4.7}$$

which implies that

$$\log\left(1+\frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\right) \approx \frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})} \ . \tag{3.4.8}$$

Applying this to the symmetric KL divergence above yields

$$D_{\mathrm{KL}}p(\mathbf{x}|\boldsymbol{\theta})(p(\mathbf{x}|\boldsymbol{\theta}+\Delta\boldsymbol{\theta}),p(\mathbf{x}|\boldsymbol{\theta})) \approx \int p(\mathbf{x}|\boldsymbol{\theta})\frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\frac{\Delta p(\mathbf{x}|\boldsymbol{\theta})}{p(\mathbf{x}|\boldsymbol{\theta})}\,\mathrm{d}\mathbf{x} \ . \tag{3.4.9}$$

[8]This is derivation is intended to sketch the proof but is not a completely rigorous proof in itself.

Given that the parameter change $\Delta\theta$ is small enough, $\nabla_\theta p(\mathbf{x}|\theta)^{\mathrm{T}}\Delta\theta$ will be a finite difference approximation to $\Delta p(\mathbf{x}|\theta)$, i.e.

$$\Delta p(\mathbf{x}|\theta) \approx \nabla_\theta p(\mathbf{x}|\theta)^{\mathrm{T}}\Delta\theta \ . \tag{3.4.10}$$

Substituting this approximation and using the log-derivative trick (3.3.4), the symmetric KL divergence becomes

$$\begin{aligned} D_{\mathrm{KL}}(p(\mathbf{x}|\theta+\Delta\theta), p(\mathbf{x}|\theta)) &\approx \int p(\mathbf{x}|\theta)\Delta\theta^{\mathrm{T}}\frac{\nabla_\theta p(\mathbf{x}|\theta)}{p(\mathbf{x}|\theta)}\frac{\nabla_\theta p(\mathbf{x}|\theta)^{\mathrm{T}}}{p(\mathbf{x}|\theta)}\Delta\theta \, \mathrm{d}\mathbf{x} \\ &= \int p(\mathbf{x}|\theta)\Delta\theta^{\mathrm{T}}\nabla_\theta \log p(\mathbf{x}|\theta)\nabla_\theta \log p(\mathbf{x}|\theta)^{\mathrm{T}}\Delta\theta \, \mathrm{d}\mathbf{x} \\ &= \Delta\theta^{\mathrm{T}}\int p(\mathbf{x}|\theta)\nabla_\theta \log p(\mathbf{x}|\theta)\nabla_\theta \log p(\mathbf{x}|\theta)^{\mathrm{T}} \, \mathrm{d}\mathbf{x} \, \Delta\theta \ . \end{aligned} \tag{3.4.11}$$

Note that the integral is an expectation with respect to $p(\mathbf{x}|\theta)$ of the outer product of the gradient of the log-PDF of the search distribution. This can be seen to be the Fisher information matrix (FIM) which is defined as [8]

$$\mathbf{F}_\theta = \mathrm{E}\big[\nabla_\theta \log p(\mathbf{x}|\theta)\nabla_\theta \log p(\mathbf{x}|\theta)^{\mathrm{T}}\big]_{p(\mathbf{x}|\theta)} \ . \tag{3.4.12}$$

Using the FIM, the approximate symmetric KL divergence can thus be written as

$$D_{\mathrm{KL}}(p(\mathbf{x}|\theta+\Delta\theta), p(\mathbf{x}|\theta)) \approx \Delta\theta^{\mathrm{T}}\mathbf{F}_\theta\Delta\theta \ . \tag{3.4.13}$$

Under certain regularity conditions which will not be discussed here, the FIM can also be written as [62]

$$\mathbf{F}_\theta = -\mathrm{E}[\mathbf{H}_\theta \log p(\mathbf{x}|\theta)] = \mathrm{Cov}[\nabla_\theta \log p(\mathbf{x}|\theta)] \tag{3.4.14}$$

which is useful for computation.

3.4.5 Steepest descent w.r.t. a distance metric

Section 3.4 set out to understand what makes the regular gradient suboptimal for gradient descent in variational optimization (VO). Now that it is clear why Euclidean distance is unsuited and the KL divergence has been introduced and substantiated as a more fitting measure of similarity between distributions, this section will derive the natural gradient by imposing a constraint on the dissimilarity between the updated and previous search distributions. To demonstrate the relation between the regular and natural gradients, the regular gradient will be derived first in an unconstrained manner and by imposing a Euclidean dissimilarity constraint.

At each iteration of VO, a direction $\Delta\theta$ is sought that minimizes the variational upper bound. With no constraints on the update this can be expressed as the following unconstrained minimization problem.

$$\min_{\Delta\theta} U(\theta+\Delta\theta) \approx U(\theta) + \nabla_\theta U(\theta)^{\mathrm{T}}\Delta\theta + \frac{1}{2}\Delta\theta^{\mathrm{T}}\mathbf{H}_\theta U(\theta)^{\mathrm{T}}\Delta\theta \ , \tag{3.4.15}$$

which uses a second order Taylor expansion of the variational upper bound around the current parameter θ. The solution to this problem is found by taking the gradient w.r.t. $\Delta\theta$ of the objective,

$$\nabla_{\Delta\theta}\left[U(\theta) + \nabla_\theta U(\theta)^{\mathrm{T}}\Delta\theta + \frac{1}{2}\Delta\theta^{\mathrm{T}}\nabla_\theta U(\theta)^{\mathrm{T}}\Delta\theta\right] = \nabla_\theta U(\theta) + \mathbf{H}_\theta U(\theta)\Delta\theta, \tag{3.4.16}$$

setting it to zero and solving for $\Delta\boldsymbol{\theta}$. This yields

$$\Delta\boldsymbol{\theta} = -\mathbf{H}_{\boldsymbol{\theta}} U(\boldsymbol{\theta})^{-1} \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})$$

which is the well-known Newton search direction. In cases where Hessian information is not available or computationally intractable, it can be ignored and a fixed learning rate used on the gradient instead giving the search direction

$$\Delta\boldsymbol{\theta} = -\eta \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) \ . \tag{3.4.17}$$

Alternatively, a constraint can be imposed on the optimization problem in (3.4.15). For instance, constraining the Euclidean distance $D_2(\cdot)$ between parameters $\boldsymbol{\theta}$ and $\boldsymbol{\theta} + \Delta\boldsymbol{\theta}$ to be some small constant κ^2, the problem becomes,

$$\begin{aligned} &\min_{\Delta\boldsymbol{\theta}} U(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \approx U(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})^{\mathrm{T}} \Delta\boldsymbol{\theta} \\ &\text{subject to } D_2(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}, \boldsymbol{\theta}) = \sqrt{\Delta\boldsymbol{\theta}^{\mathrm{T}} \Delta\boldsymbol{\theta}} = \kappa^2 \ , \end{aligned} \tag{3.4.18}$$

where only a first order Taylor approximation has been used since second order dependencies are introduced in the constraint. Such a problem is solved using the methods of constrained optimization [74]. Writing the Euclidean constraint as $\Delta\boldsymbol{\theta}^{\mathrm{T}}\Delta\boldsymbol{\theta} = \kappa$, the Lagrangian for the problem becomes

$$\mathcal{L}(\boldsymbol{\theta}, \lambda) = U(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})^{\mathrm{T}} \Delta\boldsymbol{\theta} - \lambda\big(\Delta\boldsymbol{\theta}^{\mathrm{T}} \Delta\boldsymbol{\theta} - \kappa\big) \tag{3.4.19}$$

which is minimized w.r.t. $\Delta\boldsymbol{\theta}$

$$0 = \nabla_{\Delta\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \lambda) = \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) - \lambda \Delta\boldsymbol{\theta}$$

yielding

$$\Delta\boldsymbol{\theta} = \lambda^{-1} \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) \ . \tag{3.4.20}$$

The Lagrange multiplier λ can be found by first forming the dual of the Lagrangian by insertion of the result into the primal Lagrangian and then minimizing the dual w.r.t. the multiplier. The dual Lagrangian becomes

$$\begin{aligned} \mathcal{L}_d(\lambda) &= U(\boldsymbol{\theta}) + \lambda^{-1} \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})^{\mathrm{T}} \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) - \lambda\kappa \\ &= U(\boldsymbol{\theta}) + \lambda^{-1} \nabla^2_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) - \lambda\kappa \end{aligned}$$

where ∇^2 denotes the vector Laplacian. Differentiate the Lagrangian w.r.t. λ and set it to zero,

$$0 = \frac{d}{d\lambda} \mathcal{L}_d(\lambda) = -\lambda^{-2} \nabla^2_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) - \kappa \ .$$

Finally solve for λ to obtain

$$\lambda = -\sqrt{\frac{\nabla^2_{\boldsymbol{\theta}} U(\boldsymbol{\theta})}{\kappa}} \ . \tag{3.4.21}$$

The search direction then becomes

$$\Delta\boldsymbol{\theta} = -\sqrt{\frac{\kappa}{\nabla^2_{\boldsymbol{\theta}} U(\boldsymbol{\theta})}} \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) = -\eta \nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta}) \tag{3.4.22}$$

where $\eta = \sqrt{\frac{\kappa}{\nabla^2_\theta U(\theta)}}$ is some potentially undesirable learning rate that can be replaced by a constant without changing the search direction. Note that this constrained approach has resulted in the emergence of the negative regular gradient as the optimal descent direction for minimization while constraining step sizes to be constant in the sense of Euclidean distance.

Rather than using the Euclidean distance between distribution parameters as the measure of dissimilarity between the distributions, now the approximate symmetrized KL divergence defined in (3.4.3) is used to impose the dissimilarity constraint in (3.4.18). The minimization problem is then given by,

$$\begin{aligned} &\min_{\Delta\theta} U(\theta + \Delta\theta) \approx U(\theta) + \nabla_\theta U(\theta)^{\mathrm{T}} \Delta\theta \\ &\text{subject to } D_{\mathrm{KL}}(p(\mathbf{x}|\theta + \Delta\theta), p(\mathbf{x}|\theta)) \approx \frac{1}{2} \Delta\theta^{\mathrm{T}} \mathbf{F}_\theta \Delta\theta = \kappa \ , \end{aligned} \tag{3.4.23}$$

where $p(\mathbf{x}|\theta)$ is the used search distribution. As discussed earlier, the dissimilarity constraint now takes the form of a distance w.r.t. a distance metric, here, the FIM. In fact, the FIM measures closeness of the shape of two distributions and is proportional to the amount of information that the distribution function contains about the parameter. As such, it provides a distance metric on the statistical manifold [113]. Again, to obtain the optimal search direction, the Lagrangian is formed,

$$\mathcal{L}(\theta, \lambda) = U(\theta) + \nabla_\theta U(\theta)^{\mathrm{T}} \Delta\theta - \lambda \left(\frac{1}{2} \Delta\theta^{\mathrm{T}} \mathbf{F}_\theta \Delta\theta - \kappa \right) \ , \tag{3.4.24}$$

differentiated w.r.t. $\Delta\theta$ and set equal to zero,

$$0 = \nabla_\theta \mathcal{L}(\theta, \lambda) = \nabla_{\Delta\theta} U(\theta) - \lambda \mathbf{F}_\theta \Delta\theta \ , \tag{3.4.25}$$

and finally solved for $\Delta\theta$ to yield

$$\Delta\theta = \lambda^{-1} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta) \ . \tag{3.4.26}$$

Again, the Lagrange multiplier λ can be found by minimizing the dual Lagrangian,

$$\mathcal{L}_d(\lambda) = U(\theta) + \lambda^{-1} \nabla_\theta U(\theta)^{\mathrm{T}} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta) - \lambda\kappa \ , \tag{3.4.27}$$

w.r.t the multiplier,

$$0 = \frac{d}{d\lambda} \mathcal{L}_d(\lambda) = -\lambda^{-2} \nabla_\theta U(\theta)^{\mathrm{T}} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta) - \kappa \ , \tag{3.4.28}$$

and solving for λ,

$$\lambda = -\sqrt{\frac{\nabla_\theta U(\theta)^{\mathrm{T}} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta)}{\kappa}} \ . \tag{3.4.29}$$

The search direction is then

$$\Delta\theta = -\sqrt{\frac{\kappa}{\nabla_\theta U(\theta)^{\mathrm{T}} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta)}} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta) = -\eta \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta) \tag{3.4.30}$$

where $\eta = \sqrt{\frac{\kappa}{\nabla_\theta U(\theta)^{\mathrm{T}} \mathbf{F}_\theta^{-1} \nabla_\theta U(\theta)}}$ is an arbitrary learning rate as before. This search direction is the optimal direction of descent for minimization of the objective while satisfying the constraint that the updated distribution is dissimilar from the previous distribution only by a constant amount as measured by the symmetric KL divergence. This search direction is precisely the natural gradient originally introduced in [2].

The natural gradient is thus computed by rescaling the regular gradient with the inverse FIM. Since the FIM is a positive semi-definite matrix it has $n(n+1)/2$ unique entries for a distribution with n parameters. This can in some cases be intractable to store and compute. For example, for an NN optimized with VO using a d-dimensional multivariate Gaussian with full covariance matrix $n = (d + d(d+1)/2) = \mathcal{O}(d^2)$ and the number of elements in the FIM is $\mathcal{O}(d^4)$. However, in some interesting special cases, the FIM has much fewer unique parameters. Below, such distributions are considered.

In [118], an additional abstraction is made for a general class of rotationally symmetric distributions (which includes the Gaussian) by introduction of "exponential local natural coordinates". This constitutes a reparameterization of the search distribution that results in the FIM becoming the identity and the gradient thus automatically being the natural one. This has not been considered further here but could reduce computation when using the natural gradient with distributions with e.g. covariances.

3.4.6 Natural gradient for different search distributions

This section derives the FIM and the associated natural gradient for the univariate, isotropic and separable Gaussian search distributions.

3.4.6.1 Univariate Gaussian

The FIM for a univariate Gaussian can be computed as follows. Since the univariate Gaussian has two parameters, the FIM will be a 2×2 matrix. The derivatives of the Gaussian log-PDF were computed in (3.3.17) with which the gradient of the univariate Gaussian w.r.t. to its parameter vector $\boldsymbol{\theta} = \begin{bmatrix}\mu & \sigma^2\end{bmatrix}^{\mathrm{T}}$ can be constructed as

$$\nabla_{\boldsymbol{\theta}} \log \mathcal{N}(x|\mu,\sigma^2) = \begin{bmatrix} \frac{1}{\sigma^2}(x-\mu) \\ -\frac{1}{2\sigma^2} + \frac{1}{2\sigma^4(x-\mu)^2} \end{bmatrix} . \tag{3.4.31}$$

Its Hessian is then

$$\mathbf{H}_{\boldsymbol{\theta}} \log \mathcal{N}(x|\mu,\sigma^2) = \begin{bmatrix} -\frac{1}{\sigma^2} & -\frac{1}{\sigma^4}(x-\mu) \\ -\frac{1}{\sigma^4}(x-\mu) & \frac{1}{2\sigma^4} - \frac{1}{\sigma^6}(x-\mu)^2 \end{bmatrix} . \tag{3.4.32}$$

With $\mathrm{E}[x-\mu] = 0$ and $\mathrm{E}\big[(x-\mu)^2\big] = \sigma^2$ and using (3.4.14), the FIM then becomes

$$\mathbf{F}_{\boldsymbol{\theta}} = -\mathrm{E}\big[\mathbf{H}_{\boldsymbol{\theta}} \log \mathcal{N}(x|\mu,\sigma^2)\big] = \begin{bmatrix} \frac{1}{\sigma^2} & 0 \\ 0 & \frac{1}{2\sigma^4} \end{bmatrix} \tag{3.4.33}$$

and the inverse FIM required for the natural gradient is easily obtained,

$$\mathbf{F}_{\boldsymbol{\theta}}^{-1} = \begin{bmatrix} \sigma^2 & 0 \\ 0 & 2\sigma^4 \end{bmatrix} . \tag{3.4.34}$$

The natural gradient version of the VO gradient estimator in (3.3.18) therefore becomes

$$\begin{aligned}\sigma^2 \frac{\partial}{\partial \mu} U(\mu, \sigma^2) &= \sigma \mathrm{E}[f(\mu + \sigma\epsilon)\epsilon] \approx \frac{\sigma}{N} \sum_{n=1}^{N} f(\mu + \sigma\epsilon_n)\epsilon_n \\ 2\sigma^4 \frac{\partial}{\partial \sigma^2} U(\mu, \sigma^2) &= \sigma^2 \mathrm{E}\left[f(\mu + \sigma\epsilon)\left(\epsilon^2 - 1\right)\right] \approx \frac{\sigma^2}{N} \sum_{n=1}^{N} f(\mu + \sigma\epsilon_n)\left(\epsilon_n^2 - 1\right)\end{aligned} \tag{3.4.35}$$

where $\epsilon \sim \mathcal{N}(0, 1)$. These evidently posses the desired property that the gradient goes to zero as the variance goes to zero and the opposite.

3.4.6.2 Isotropic Gaussian

A d-dimensional isotropic Gaussian, $\mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{I})$, has $d + 1$ parameters which implies that the FIM will be a $(d+1) \times (d+1)$ matrix. The gradients of the log-PDF of the isotropic Gaussian were derived in (3.3.43). The gradient of the isotropic Gaussian w.r.t. to its parameter vector $\boldsymbol{\theta} = \begin{bmatrix}\boldsymbol{\mu} & \sigma^2\end{bmatrix}^{\mathrm{T}}$ can then be constructed as

$$\nabla_{\boldsymbol{\theta}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I}) = \begin{bmatrix} \frac{1}{\sigma^2}(\mathbf{x} - \boldsymbol{\mu}) \\ -\frac{d}{2\sigma^2} + \frac{1}{2\sigma^4}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu}) \end{bmatrix} . \tag{3.4.36}$$

It follows that the Hessian is

$$\mathbf{H}_{\boldsymbol{\theta}} \log \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \sigma^2\mathbf{I}) = \begin{bmatrix} -\frac{1}{\sigma^2}\mathbf{I} & -\frac{1}{\sigma^4}(\mathbf{x} - \boldsymbol{\mu}) \\ -\frac{1}{\sigma^4}(\mathbf{x} - \boldsymbol{\mu}) & \frac{d}{2\sigma^4} - \frac{1}{\sigma^6}(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu}) \end{bmatrix} . \tag{3.4.37}$$

Again, use (3.4.14) to compute the FIM and exploit that $\mathrm{E}[\mathbf{x} - \boldsymbol{\mu}] = \mathbf{0}$ and $\mathrm{E}\left[(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu})\right] = d\sigma^2$ to obtain[9]

$$\mathbf{F}_{\boldsymbol{\theta}} = \begin{bmatrix} \frac{1}{\sigma^2}\mathbf{I} & 0 \\ 0 & \frac{1}{2\sigma^4} \end{bmatrix} \Longleftrightarrow \mathbf{F}_{\boldsymbol{\theta}}^{-1} = \begin{bmatrix} \sigma^2\mathbf{I} & 0 \\ 0 & 2\sigma^4 \end{bmatrix} . \tag{3.4.38}$$

This is simply the univariate result in each dimension of the mean while the result for the scalar variance parameter remains unchanged. The natural gradient version of the VO gradient estimator in (3.3.44) therefore becomes

$$\begin{aligned}\sigma^2 \nabla_{\boldsymbol{\mu}} U(\boldsymbol{\mu}, \sigma^2) &= \sigma \mathrm{E}[f(\boldsymbol{\mu} + \sigma\boldsymbol{\epsilon})\boldsymbol{\epsilon}] \approx \frac{\sigma}{N} \sum_{n=1}^{N} f(\boldsymbol{\mu} + \sigma\boldsymbol{\epsilon}_n)\boldsymbol{\epsilon}_n \\ 2\sigma^4 \nabla_{\sigma^2} U(\boldsymbol{\mu}, \sigma^2) &= \sigma^2 \mathrm{E}\left[f(\boldsymbol{\mu} + \sigma\boldsymbol{\epsilon})\left(\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon} - d\right)\right] \approx \frac{\sigma^2}{N} \sum_{n=1}^{N} f(\boldsymbol{\mu} + \sigma\boldsymbol{\epsilon}_n)\left(\boldsymbol{\epsilon}_n^2 - d\right)\end{aligned} \tag{3.4.39}$$

[9]This follows from the following argument.

$$\mathrm{E}\left[(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu})\right] = \mathrm{E}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x}\right] - \mathrm{E}\left[\mathbf{x}^{\mathrm{T}}\boldsymbol{\mu}\right] - \mathrm{E}\left[\boldsymbol{\mu}^{\mathrm{T}}\mathbf{x}\right] + \mathrm{E}\left[\boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\mu}\right] = \mathrm{E}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x}\right] - \boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\mu} .$$

Now, when $\mathbf{x} \sim \mathcal{N}(\mu, \sigma^2\mathbf{I})$

$$\mathrm{E}\left[\mathbf{x}^{\mathrm{T}}\mathbf{x}\right] = \mathrm{E}\left[\sum_{i=1}^{d} x_i^2\right] = \sum_{i=1}^{d} \mathrm{E}[x_i^2] = \sum_{i=1}^{d} \mathrm{Var}[x_i] + \mathrm{E}[x_i]^2 = \sum_{i=1}^{d} \sigma^2 + \mu_i^2 = d\sigma^2 + \boldsymbol{\mu}^{\mathrm{T}}\boldsymbol{\mu} ,$$

such that

$$\mathrm{E}\left[(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}(\mathbf{x} - \boldsymbol{\mu})\right] = d\sigma^2 .$$

where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The transformation applied to the VO gradient using the isotropic Gaussian is clearly identical to the one applied to the gradient using a univariate Gaussian; in both cases the mean and variance gradients are multiplied by the scalars σ^2 and $2\sigma^4$, respectively.

3.4.6.3 Separable Gaussian

A d-dimensional separable Gaussian, $\mathcal{N}\big(\mathbf{x}|\boldsymbol{\mu}, (\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}}) \odot \mathbf{I}\big)$, has $2d$ parameters which implies that the FIM will be a $2d \times 2d$ matrix. The gradients of the log-PDF of the separable Gaussian were derived in (3.3.52) and (3.3.53). The gradient of the separable Gaussian w.r.t. its parameter vector $\boldsymbol{\theta} = \begin{bmatrix}\boldsymbol{\mu} & \boldsymbol{\sigma}^2\end{bmatrix}^{\mathrm{T}}$ can then be constructed as

$$\nabla_{\boldsymbol{\theta}} \log \mathcal{N}\big(\mathbf{x}|\boldsymbol{\mu}, (\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}}) \odot \mathbf{I}\big) = \begin{bmatrix} \boldsymbol{\sigma}^{-2} \odot (\mathbf{x} - \boldsymbol{\mu}) \\ -\frac{1}{2}\boldsymbol{\sigma}^{-2} + \frac{1}{2}\boldsymbol{\sigma}^{-4} \odot (\mathbf{x} - \boldsymbol{\mu})^2 \end{bmatrix}. \tag{3.4.40}$$

It follows that the Hessian is

$$\mathbf{H}_{\boldsymbol{\theta}} \log \mathcal{N}\big(\mathbf{x}|\boldsymbol{\mu}, (\boldsymbol{\sigma}^2\mathbf{e}^{\mathrm{T}}) \odot \mathbf{I}\big) = \begin{bmatrix} \boldsymbol{\sigma}^{-2} \odot \mathbf{I} & -\boldsymbol{\sigma}^{-4} \odot (\mathbf{x} - \boldsymbol{\mu}) \\ -\boldsymbol{\sigma}^{-4} \odot (\mathbf{x} - \boldsymbol{\mu}) & \frac{1}{2}\boldsymbol{\sigma}^{-4} - \boldsymbol{\sigma}^{-6} \odot (\mathbf{x} - \boldsymbol{\mu})^2 \end{bmatrix}. \tag{3.4.41}$$

Again, use (3.4.14) to compute the FIM and exploit that $\mathrm{E}[\mathbf{x} - \boldsymbol{\mu}] = \mathbf{0}$ and $\mathrm{E}\big[(\mathbf{x} - \boldsymbol{\mu})^2\big] = \boldsymbol{\sigma}^2$ to obtain

$$\mathbf{F}_{\boldsymbol{\theta}} = \begin{bmatrix} \boldsymbol{\sigma}^{-2} \odot \mathbf{I} & 0 \\ 0 & \frac{1}{2}\boldsymbol{\sigma}^{-4} \odot \mathbf{I} \end{bmatrix} \Longleftrightarrow \mathbf{F}_{\boldsymbol{\theta}}^{-1} = \begin{bmatrix} \boldsymbol{\sigma}^{2} \odot \mathbf{I} & 0 \\ 0 & 2\boldsymbol{\sigma}^{4} \odot \mathbf{I} \end{bmatrix}. \tag{3.4.42}$$

Once again, this is the univariate result in each dimension of the parameter vector of the separable Gaussian. The natural gradient version of the VO gradient estimator in (3.3.54) therefore becomes

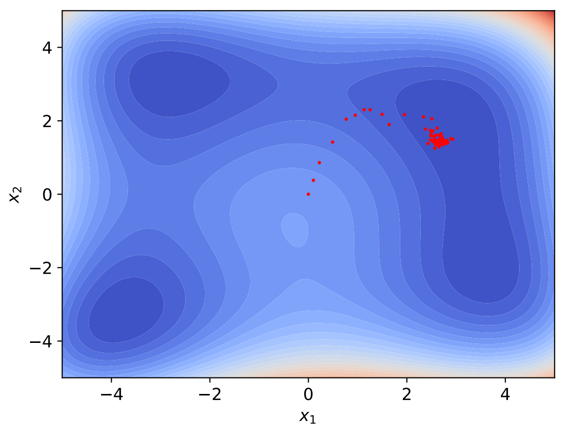
$$\begin{aligned} \boldsymbol{\sigma}^2 \odot \nabla_{\boldsymbol{\mu}} U(\boldsymbol{\mu}, \sigma^2) &= \boldsymbol{\sigma} \odot \mathrm{E}[f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon})\boldsymbol{\epsilon}] \\ &\approx \boldsymbol{\sigma} \odot \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}_n)\boldsymbol{\epsilon}_n \\ 2\boldsymbol{\sigma}^4 \odot \nabla_{\sigma}^2 U(\boldsymbol{\mu}, \sigma^2) &= \boldsymbol{\sigma}^2 \odot \mathrm{E}\big[f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon})\big(\boldsymbol{\epsilon}^2 - 1\big)\big] \\ &\approx \boldsymbol{\sigma}^2 \odot \frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}_n)\big(\boldsymbol{\epsilon}_n^2 - 1\big) \end{aligned} \tag{3.4.43}$$

where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The transformation applied here is again similar to the one applied to the gradient using a univariate Gaussian.

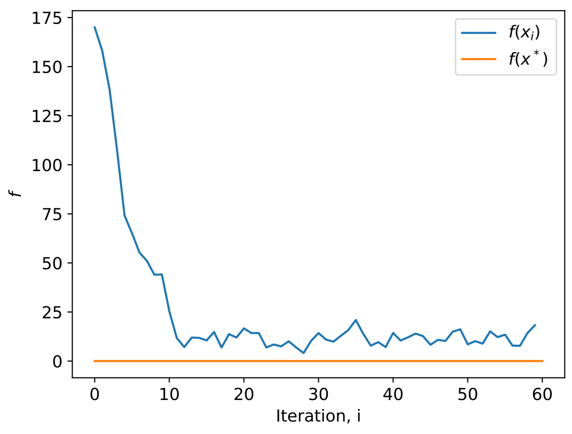
3.4.7 Comparison of regular and natural gradients

In Figures 3.7, 3.8 and 3.9, different versions of the variational optimization algorithm listed in Algorithm 1 are compared on Himmelblau's function. Himmelblau's function is two-dimensional objective function with four global minima and a local maximum. In these figures as before, red denotes high values, blue denotes low values and the darker the colour, the higher the absolute value. The actual values are omitted for simplicity.

In Figure 3.7, the convergence of the algorithm used in [90] and derived in (3.2.24) is shown. This algorithm estimates the regular gradient and does not adapt the value of the Gaussian search distribution variance. In **(a)**, the iteration sequence is plotted on the contours of the



(a)



(b)

Figure 3.7: **(a)** Convergence on Himmelblau's function of the algorithm used by [90] with no adaptation of the variance. As in earlier contours, red denotes high values, blue denotes low values and the darker the colour, the higher the absolute value. The actual values are omitted for simplicity. **(b)** Objective function value at each iteration of the algorithm. The algorithm finds a minimum but struggles to converge due to the fixed search distribution variance.

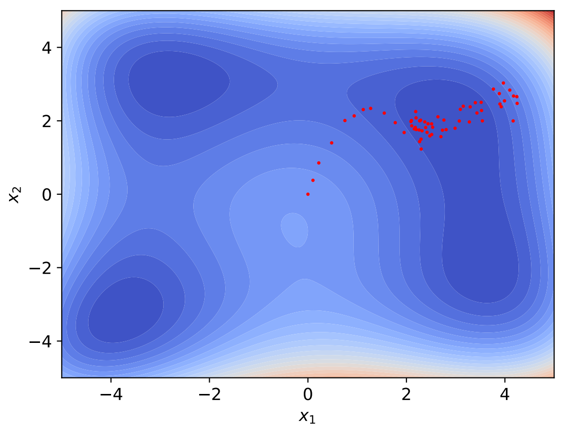
objective function while in **(b)**, the corresponding objective function value at each iteration is shown along with the global minimum value. Evidently, this algorithm quickly homes in on one of the global minima and stays in its vicinity for succeeding iterations. The objective function value reaches low values, but the fixed size of the variance results in it fluctuating above the global minimum never converging on it.

Figure 3.8 presents the convergence of the VO algorithm using an isotropic Gaussian search distribution. The used estimators for the regular gradients of the mean and variance are those in (3.3.44). Similarly to the fixed variance version, this algorithm also quickly homes in on a minimum. However, the optimization of the variance results in increasingly smaller variances for each iteration. As previously discussed and and as expected, this increasingly smaller variance can be seen to give rise to increasingly larger gradient estimates. In the limit of zero variance (close to the minimum), the gradients explode and the function value starts increasing.

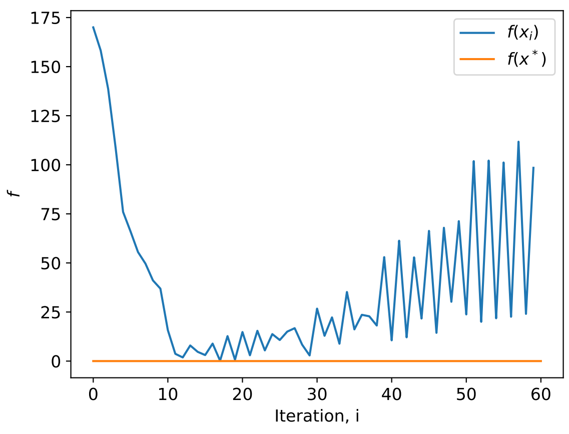
Finally, Figure 3.9 presents the convergence of the same VO algorithm using instead the natural gradient. The VO natural gradient estimators for the univariate Gaussian used here are those in (3.4.35). The minimum is found as before but this time the optimization of the variance drives the gradients to zero at the minimum resulting in much better convergence. Although the direction of the natural gradient is not different from the regular gradient for this specific type of search distribution, the gradient is much more well-behaved as σ decreases.

These examples clearly demonstrate the differences between the methods and highlights VO with natural gradients as the best choice for this problem. Replacing the regular gradient with the natural gradient results in better numerics for the estimators and convergence to the minimum. It should however be noted that Himmelblau's function is a low-dimensional objective function and the number of perturbations is much larger than the dimensionality for this example. This is fundamentally different from the optimization of NNs by search in the extremely high-dimensional parameter space.

It can be noted that Gaussian VO with fixed variance as in Figure 3.7 in fact optimizes

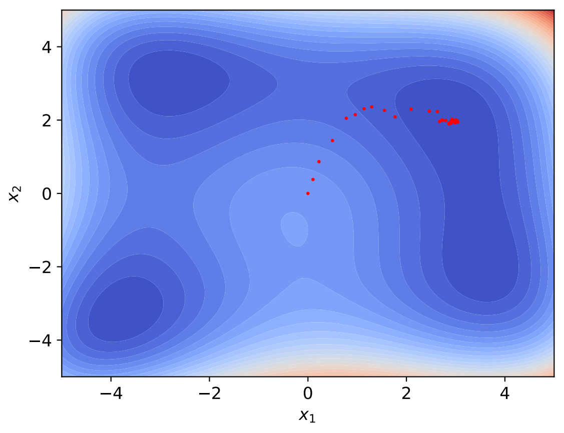


(a)

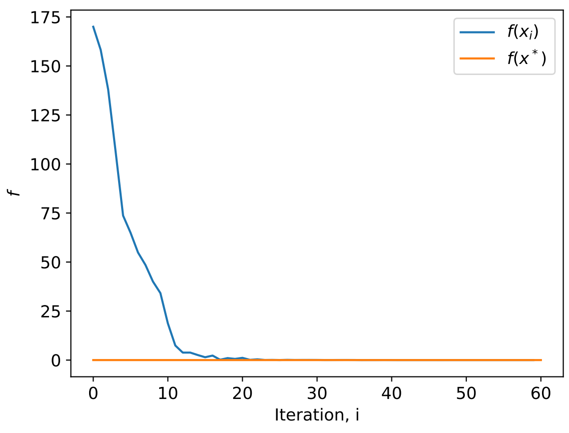


(b)

Figure 3.8: **(a)** Convergence of the VO algorithm with isotropic Gaussian search distribution using regular gradients. **(b)** Objective function value at each iteration. Similarly to the fixed variance version, a minimum is found, but the optimization of the variance drives it towards zero, resulting in larger gradients and instability.



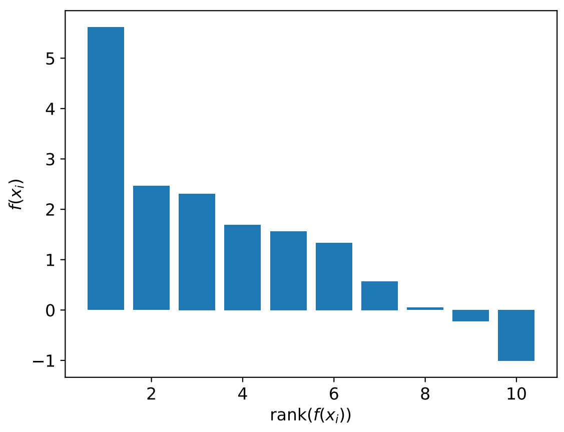
(a)



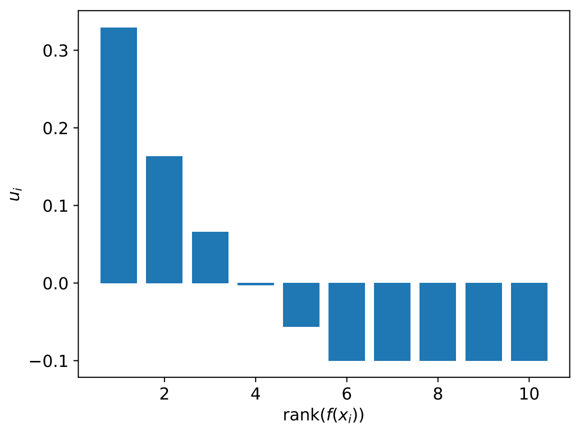
(b)

Figure 3.9: **(a)** Convergence of the VO algorithm with isotropic Gaussian search distribution using natural gradients. **(b)** Objective function value at each iteration. A minimum is found and the optimization of the variance drives the gradients toward zero resulting in convergence to the optimum.

for the expected fitness of the entire population of perturbations rather than the fitness of the search distribution center [53]. As such, the fitness of the center of the search distribution (the unperturbed individual) may not be optimal at all although it often is close to the optimum, at least in terms of search space distance. This also results in solutions that exhibit robustness to small perturbations to the parameters [53].



(a)



(b)

Figure 3.10: Illustration of the fitness rank transform applied to evaluated fitnesses. **(a)** Ten randomly sampled fitnesses. **(b)** The ten corresponding utilities according to (3.5.1).

3.5 Performance and robustness improving techniques

3.5.1 Fitness rank transform

In general, the values returned by the function being optimized, $f(\mathbf{x}_i)$, may vary wildly in size depending on the function and the points at which it is sampled. Directly using the returned function value makes hyperparameters, such as the learning rate, highly dependent on the objective function. Additionally, the returned function values may increase or decrease as the algorithm converges to an optimum which can result in numerical instability.

One way to avoid this is to transform the function values before they are used to compute the gradient. One possible transform type is the so-called rank transform. This transform ranks the sampled function values $f(\mathbf{x}_i)$ and assigns to each one a value that depends only the relative value of the sample compared to the other samples. These new values are called utilities. Utilities do not change in size during optimization and have no dependency on the actual value of the objective function but do encode information on the relative fitness of the samples. Using utilities makes VO invariant to any order-preserving transformation of the objective function such as addition or multiplication by a scalar [118]. To decouple the learning rate of the algorithm from the utilities it can be required that $\sum_{i=1}^{N} |u_i| = C$ for a constant C.

Several possible choices of rank transform exist and it should be viewed as a free parameter of the problem. In this thesis, the same fitness rank transform as used in [118] has been applied throughout. This is also closely related to the one used in the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [31]. This transformation requires sorting the fitnesses $f(\mathbf{x}_i)$ descending order such that $\mathbf{x}_1$ denotes the best sample and $\mathbf{x}_N$ the worst. The associated fitnesses are then replaced by the utilities given by

$$u_i = \frac{\max\left\{0, \log\left(\frac{N}{2}+1\right) - \log(k)\right\}}{\sum_{j=1}^{N} \max\left\{0, \log\left(\frac{N}{2}+1\right) - \log(j)\right\}} - \frac{1}{N} \ , \quad i = 1, \dots, N \ , \tag{3.5.1}$$

such that u_i replaces $f(\mathbf{x}_i)$. Figure 3.10 shows the the transformation of ten randomly sampled fitnesses in **(a)** to their corresponding utilities in **(b)** using (3.5.1).

3.5.2 Importance mixing

First introduced in [110] and explored further in [122] and [93], importance mixing is a probabilistic approach to decrease the number of new samples necessary to estimate the search distribution gradient at each iteration and is closely related to importance sampling [29]. Importance mixing exploits the fact that the VO algorithm maintains a search distribution of candidate solutions to include previous perturbations into the next iteration based on how likely they are under the new search distribution.

3.5.2.1 Procedure

Specifically, since the search distribution is only every updated in small steps, the KL divergence between the current distribution, $p(\boldsymbol{\epsilon}|\boldsymbol{\theta})$, and the previous one, $p(\mathbf{x}|\boldsymbol{\theta}')$, will always be relatively small[10]. This results in many perturbations sampled from $p(\mathbf{x}|\boldsymbol{\theta})$ being located in high density areas of $p(\boldsymbol{\epsilon}|\boldsymbol{\theta}')$. In turn, this leads to fitness evaluations being unnecessarily repeated [110].

Importance mixing seeks to solve this problem by the following algorithm. First, perform rejection sampling on the previous search distribution, $p(\mathbf{x}|\boldsymbol{\theta}')$, such that perturbation $\boldsymbol{\epsilon}_i$ is accepted for reuse with probability

$$\min\left\{1, (1-\alpha)\frac{p(\boldsymbol{\epsilon}_i)|\boldsymbol{\theta})}{p(\boldsymbol{\epsilon}_i)|\boldsymbol{\theta}')}\right\} \tag{3.5.2}$$

where $\alpha \in [0, 1]$ is the forced minimal refresh rate. The more likely the perturbation is in the new distribution compared to the old, the higher the acceptance probability. This leads to the acceptance of $N_a \geq 0$ perturbations. Second, perform reverse rejection sampling of perturbations from the current search distribution, $\boldsymbol{\epsilon}_j \sim p(\boldsymbol{\epsilon}|\boldsymbol{\theta})$ accepting the perturbations with probability

$$\max\left\{\alpha, 1-\frac{p(\boldsymbol{\epsilon}_j)|\boldsymbol{\theta}')}{p(\boldsymbol{\epsilon}_j)|\boldsymbol{\theta})}\right\} \tag{3.5.3}$$

until a $N-N_a$ perturbations have been accepted resulting in a total sample size of N. This procedure is listed in Algorithm 2. In the extreme, setting $\alpha = 1$ results in no perturbations being accepted in the first step and all perturbations being accepted in step two. This is effectively the same as not using importance mixing. It can be noted that since the reused samples are likely in the new search distribution, they must also be some of the better performing samples of the previous iteration, assuming that the search distribution was updated in a promising direction.

In [110] this algorithm is shown to yield a new set of samples that conform to the current search distribution, $p(\boldsymbol{\epsilon}|\boldsymbol{\theta})$. Furthermore, [122] shows empirically that the number of new fitness evaluations in an iteration was reduced by a factor of 5 for a range of black box optimization benchmarks.

3.5.2.2 Previous work and dimensionality

To the author's best knowledge, importance mixing has been applied along with VO mostly to ordinary black box function optimization and only to very small NNs, such as an RNN with 21 weights optimized to solve the pole balancing task [122]. The recent application of a VO

[10]This is even more true when using the natural gradient as discussed in Section 3.4

Algorithm 2 Importance mixing [110].

```
Require: Search distributions; current, $p(\boldsymbol{\epsilon}|\boldsymbol{\theta})$, and previous, $p(\boldsymbol{\epsilon}|\boldsymbol{\theta}')$. $N$ previous perturbations, $\boldsymbol{\epsilon}_i \sim p(\boldsymbol{\epsilon}|\boldsymbol{\theta}')$. Forced minimal refresh rate, $\alpha$.
 1: for each previous perturbation $i = 1, \ldots, N$ do
 2:     Compute importance weight
            $$\frac{p(\boldsymbol{\epsilon}_i|\boldsymbol{\theta})}{p(\boldsymbol{\epsilon}_i|\boldsymbol{\theta}')}$$
 3:     Accept $\boldsymbol{\epsilon}_i$ for reuse with probability
            $$\min\left\{1, (1-\alpha)\frac{p(\boldsymbol{\epsilon}_i|\boldsymbol{\theta})}{p(\boldsymbol{\epsilon}_i|\boldsymbol{\theta}')}\right\}$$
 4: end for
 5: Let $N_a$ be the number of perturbations accepted for reuse.
 6: repeat with $j = 1, \ldots$
 7:     Sample new perturbation $\boldsymbol{\epsilon}_j \sim p(\boldsymbol{\epsilon}|\boldsymbol{\theta})$
 8:     Accept $\boldsymbol{\epsilon}_j$ with probability
            $$\max\left\{\alpha, 1 - \frac{p(\boldsymbol{\epsilon}_j|\boldsymbol{\theta}')}{p(\boldsymbol{\epsilon}_j|\boldsymbol{\theta})}\right\}$$
 9: until $N - N_a$ new perturbations have been accepted
10: return the total set of perturbations $\{\boldsymbol{\epsilon}_i, \boldsymbol{\epsilon}_j\}$ for all accepted $i$ and $j$.
```

variant to neural networks in [90] did not reuse any previous samples, always creating new ones at every iteration.

Considering search space dimensionality, the previous applications of importance mixing correspond to the case where the number of samples is larger than the number of dimensions, $N > d$. In this case, the problem solved by importance mixing is the redundant sampling of the objective function. Importance mixing then works by sampling from the new distribution while reusing samples from the distribution of the previous iteration. The resulting total set of samples then conforms to the new distribution while having a reduced number of new samples to evaluate. Using importance mixing thus decreases the number of required function evaluations. The number of search distribution parameter updates (gradient estimates) is not lowered by use of importance mixing.

However, when the search space dimensionality is much higher than the number of samples, $N \ll d$, importance mixing can be interpreted from another perspective as well. Recall the discussion on dimensionality of Section 3.3.2.4. Here it was noted that when $N \ll d$, the N samples span an N dimensional subspace of the d dimensional search space. Reusing samples from the previous iteration(s) then corresponds to reusing the most promising subspace dimensions from the previous iteration for the new search.

3.5.2.3 High-dimensional importance weight collapse

Notwithstanding the promises of importance mixing, as many other techniques and algorithms, it suffers from the curse of dimensionality. The issue arises as a consequence of the high dimension of the samples and their associated search distributions. Specifically, the importance weights collapse for high dimensional density functions since they have practically disjoint support[11] [6]. In practice this results in the importance weights being zero or in rare cases practically infinite in turn removing the random element of the rejection sampling. This makes importance mixing unsuited for high-dimensional spaces; at least without modifications. Figure 3.11 illustrates the high-dimensional importance weight collapse.

Obvious methods for addressing the problem of high-dimensionality are those of dimensionality reduction. Dimensionality reduction in the weight space of NNs naturally hints towards indirect encondings. Examples are those developed for neuroevolution applications in [107] and [48]. Another approach to the dimensionality reduction is that developed in [109] which specifically aims at improving density ratio estimation by performing estimation in the so-called hetero-distributional subspace which can be low-dimensional. Yet another way of reducing the dimensionality of the search space of NNs is to consider the activations rather than the weights and then map activation gradients to weight gradients. The number of activations is typically much smaller than the number of weights. In a regular FNN, the transformation between two layers with respectively n_1 and n_2 activations has a total of n_1+n_2 activations and $n_1 n_2$ weights potentially with n_2 biases. For large enough values of n_1 and n_2, $n_1 + n_2 \ll n_1 n_2$, although the activation space dimension is still rather high for most NN models. Circumventing the importance weight collapse in the context of VO could give potentially significant speedups due to reuse of information and fewer required function evaluations.

Importance sampling has seen other applications within deep learning. In [125] and [1] it was used to form an alternative to uniform sampling of training data during SGD that focuses on the most informative parts of the training data in order to reduce gradient variance. In [43] and [44] these ideas were further developed and [44] finds relative test error reductions of between 5 and 17% using this technique for image classification. In [43], a surrogate NN was trained to predict approximate importance weights during training of a primary task NN.

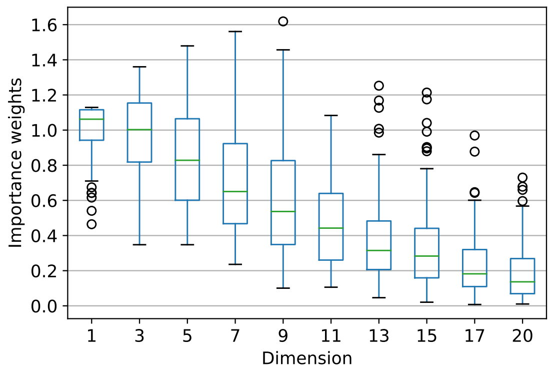
3.5.3 Adaptation sampling

Adaptation sampling is another probabilistic approach for improving VO. As importance mixing, it relies on computation of the importance weights and as such suffers from the curse of dimensionality. Here, it is introduced nevertheless since it can potentially be used to adapt hyperparameters of the optimization problem online including the all-important learning rate. Adaptation sampling has not been used experimentally in this thesis.

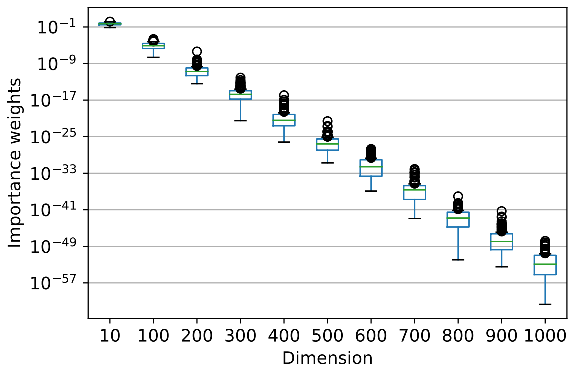
First introduced in [93] and further discussed and experimentally validated in [92], adaptation sampling overall works by considering if a fractional change to any certain hyperparameter would have yielded a search distribution more likely to produce the best performing perturbations of the current search distribution.

This presentation will consider the learning rate applied to the search gradient, η. Adaptation sampling is then used to determine if a hypothetical search distribution $p(\mathbf{x}|\boldsymbol{\theta}')$ would have been more likely to generate the best performing samples from the current distribution $p(\mathbf{x}|\boldsymbol{\theta})$. The hypothetical distribution is obtained as was the current but by using a slightly larger learning rate $(1+c)\eta$ where $c > 0$ is some small number. Then, importance weights are

[11] The support of a real-valued function is the subset of the domain outside of which the function is equal to zero[15]. Two sets are said to be disjoint if they have no elements in common. Disjoint support of two densities then means that whenever one density produces a nonzero output, the other produces a zero.



(a)



(b)

Figure 3.11: Importance weights, such as those used in Algorithm 2, collapse for high-dimensional density functions. The figures show boxplots for the distribution of 1000 importance weights computed from samples drawn from a separable Gaussian of different dimensionalities. The importance weights are computed as $\frac{\mathcal{N}(\boldsymbol{\mu}_1+\mathbf{r},\mathrm{diag}(\boldsymbol{\sigma}_1+\mathbf{r})}{\mathcal{N}(\boldsymbol{\mu}_2+\mathbf{r},\mathrm{diag}(\boldsymbol{\sigma}_2+\mathbf{r})}$ with $\boldsymbol{\mu}_1 = 1.1\mathbf{e}, \boldsymbol{\sigma}_1 = 0.9\mathbf{e}, \boldsymbol{\mu}_2 = 1.0\mathbf{e}, \boldsymbol{\sigma}_2 = 1.0\mathbf{e}$ and $\mathbf{r} \sim \mathcal{N}(\mathbf{0}, 0.01\mathbf{e})$. **(a)** shows dimensions one to twenty while **(b)** shows dimensions up to 1000. (For a reference on interpretation of these boxplots, see Appendix F.

computed in the same way as for importance mixing,

$$w_i = \frac{p(\mathbf{x}_i|\boldsymbol{\theta}')}{p(\mathbf{x}_i|\boldsymbol{\theta})} , \tag{3.5.4}$$

for the samples $\mathbf{x}_i \sim p(\mathbf{x}|\boldsymbol{\theta})$ for $i = 1, \dots, N$. The samples are then ranked based on their fitnesses and being assigned weights with one set of ranks assigned the importance weights while another set gets unit weights,

$$\begin{aligned} S' &\leftarrow \{(w_i, \mathrm{rank}(\mathbf{x}_i))\} \\ S &\leftarrow \{(1, \mathrm{rank}(\mathbf{x}_i))\} . \end{aligned} \tag{3.5.5}$$

The goal is then to decide if S' is superior to S. In [93], a weighted version of the Mann-Whitney test is developed for this purpose. The regular Mann-Whitney test is a non-parametric test of the null hypothesis that it is equally likely that a random sample from the set S will be less than or greater than a random sample from a second set S' [64]. Thus, a rejection of the null hypothesis at significance level ρ is equivalent to concluding that one sample is "larger" than the other at that significance level. If S' is tested larger than S, the learning rate is increased by a fractional amount up to some maximal value $\eta_{\max}$,

$$\eta \leftarrow \min\{(1+c)\eta, \eta_{\max}\} \tag{3.5.6}$$

where c is some small number. If S' is tested smaller than S, the learning rate is decayed towards its initial value,

$$\eta \leftarrow (1-c)\eta + c\eta_0 \tag{3.5.7}$$

where η_0 is the initial value. The Mann-Whitney test and its weighted generalization are described in Appendix D for convenience. The adaptation sampling procedure is summarized in Algorithm 3.

Algorithm 3 Adaptation sampling [92].

```
Require: Search distribution parameters from current and previous iteration, $\theta_t$ and $\theta_{t-1}$.
    Learning rate $\eta$, initial learning rate $\eta_0$ and maximum learning rate $\eta_{\max}$. Set of samples
    and associated fitnesses $\{\mathbf{x}_i, f(\mathbf{x}_i)\}$. Increase factor/decay rate $c$ and significance level $\rho$.
 1: Compute the hypothetical parameters $\theta'$ using $(1+c)\eta$.
 2: for each sample $i = 1, \dots, N$ do
 3:     Compute the importance weight
            $$w_i = \frac{p(\mathbf{x}_i)|\theta')}{p(\mathbf{x}_i)|\theta)}$$
 4: end for
 5: Compute the rank of each sample from its fitness $\{\text{rank}(\mathbf{x}_i)\}$
 6: Let $S$ and $S'$ be the respectively unit weighted and importance weighted sequences of ranks
            $$S \leftarrow \{1, \text{rank}(\mathbf{x}_i)\}$$
            $$S' \leftarrow \{w_i, \text{rank}(\mathbf{x}_i)\}$$
 7: Perform a statistical test to determine if the sequence $S'$ is superior to its $S$.
 8: if $S'$ superior then return $\min\{(1+c)\eta, \eta_{\max}\}$        ▷ Increase learning rate
 9: else return $(1-c)\eta + c\eta_0$        ▷ Decay learning rate towards $\eta_0$
10: end if
```

3.5.4 Safe mutation

When perturbing network weights in order to obtain a potentially better performing network there is a significant risk that the learned transformations are broken. This can result in a model that no longer performs anywhere near the performance of the unperturbed model, at worst completely erasing all learned transformations from input to output.

In [54], a method for coping with this challenge was presented. The publication was part of a series of articles [53, 16, 124, 108] that explore optimization of NNs in the parameter space by a variant of VO based on [90]. In [54], backpropagation is used to compute sensitivities of model output units to changes in weights. This sensitivity information can then be used to scale random perturbations of network weights so as to avoid perturbing the model in a manner that changes its output too much.

This section introduces the background of the so-called "safe mutation" and how to appropriately compute the sensitivities and scale the sampled perturbations based on [54].

3.5.4.1 Output divergence from weight perturbation

Let $\mathbf{y}(\mathbf{x}, \mathbf{w}) = NN(\mathbf{x}|\mathbf{w})$ denote the output $\mathbf{y}$ of a neural network NN parameterized by $\mathbf{w}$ and evaluated on some input $\mathbf{x}$. For a mini-batch of I inputs $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \dots & \mathbf{x}_I\end{bmatrix}$ this gives $\mathbf{Y}(\mathbf{X}, \mathbf{w}) = NN(\mathbf{X}|\mathbf{w})$ with $Y_{ki} = NN(X_{:,i}|\mathbf{w})_k$ equal to the k'th output unit's value for the i'th batch example. Now, the divergence of the output of the model that results from an additive perturbation $\boldsymbol{\epsilon}$ to its weights can be computed by

$$D(\boldsymbol{\epsilon}|\mathbf{w}) = \frac{1}{I}\sum_{k=1}^{K}\sum_{i=1}^{I}\left(NN(\mathbf{x}_i|\mathbf{w})_k - NN(\mathbf{x}_i|\mathbf{w}+\boldsymbol{\epsilon})_k\right)^2 \tag{3.5.8}$$

$$= \frac{1}{I}\sum_{k=1}^{K}\sum_{i=1}^{I}\left(NN(\mathbf{X}|\mathbf{w})_{ki} - NN(\mathbf{X}|\mathbf{w}+\boldsymbol{\epsilon})_{ki}\right)^2 \ . \tag{3.5.9}$$

The divergence can thus be computed by forward propagating a mini-batch through the unperturbed and perturbed networks and summing the element-wise squared differences of their outputs, finally normalizing by the mini-batch size.

3.5.4.2 Safe mutation through rescaling (SM-R)

The simplest way to constrain the divergence of network outputs is to decompose the perturbation into a direction and a magnitude. The direction is sampled from the search distribution but is normalized to have $\|\boldsymbol{\epsilon}\| = 1$ and is then rescaled to match some predefined amount of divergence. The safe perturbation is then given by

$$\boldsymbol{\epsilon}_{\text{safe}} = \alpha\boldsymbol{\epsilon} \ , \tag{3.5.10}$$

where $\alpha > 0$ is the magnitude of the rescaling. This magnitude can be found using a simple line search algorithm, requiring a forward pass of the mini-batch in each of the networks and an evaluation of the divergence at every iteration. The mini-batch need not be resampled for this purpose, i.e. in reinforcement learning, no additional rollouts are required if the experiences are stored.

This rescaling approach is appealing due to its well-defined metric of "safety" that is, it specifically chooses perturbations to give a certain amount of divergence in the network outputs. However, if a few parameters have very high sensitivities, perturbation of these may dominate the divergence and result in a rescaling factor that is very small. In turn, this leads to safe perturbations that are close to zero effectively resulting in the perturbed and unperturbed models being almost identical, $NN(\cdot|\mathbf{w}) \sim NN(\cdot|\mathbf{w}+\boldsymbol{\epsilon}_{\text{safe}})$. Geometrically, this is due to the direction of the perturbation being fixed while only the magnitude is rescaled.

While this rescaling approach can be applied to non-differentiable networks, gradient information of the network outputs w.r.t. the weights can be used to also change the direction of the perturbation.

3.5.4.3 Safe mutation through gradients (SM-G)

In order to scale the perturbation direction as well as the magnitude to achieve some reduction in the divergence of the network outputs, the gradient of the network outputs w.r.t. the weights can be computed. As such, safe mutation through gradients can be seen as stopping one step short in the application of the chain rule to the computational graph of the model. Instead of computing the gradient of some objective w.r.t. the network weights, the gradient is computed for the outputs themselves. For instance, in

$$\frac{\partial E}{\partial \mathbf{w}} = \frac{\partial E}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}} \tag{3.5.11}$$

$\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$ is computed rather than $\frac{\partial E}{\partial \mathbf{w}}$. In fact, $\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$ can be seen as being the Jacobian matrix describing the sensitivities of the output units to changes in the weights.

By a first order Taylor expansion, the network output at a single unit k for a single example $\mathbf{x}_i$, $Y_{ki} = NN(\mathbf{x}_i|\mathbf{w})_k$, can be seen as a function of the weight perturbation,

$$Y_{ki}(\mathbf{x}_i, \boldsymbol{\epsilon}|\mathbf{w}) \approx NN(\mathbf{x}_i|\mathbf{w})_k + \boldsymbol{\epsilon}\nabla_{\mathbf{w}} NN(\mathbf{x}_i|\mathbf{w})_k \ . \tag{3.5.12}$$

The gradient of the output w.r.t. any weight, $\nabla_{\mathbf{w}} NN(\mathbf{x}_i|\mathbf{w})_k$, can thus be seen as a single point estimate of the sensitivity of that output unit to that weight. For a mini-batch of inputs, these estimates can be summed and averaged to give a better sensitivity estimate. The sensitivity of the k'th output unit can then be written as

$$\frac{1}{I}\sum_{i=1}^{I} |\nabla_{\mathbf{w}} NN(\mathbf{X}|\mathbf{w})_{ki}| \ .$$

Here it is important to note that to compute this single unit sensitivity, each batch example must be forward propagated, have its loss evaluated, be backpropagated to compute gradients and then the gradients must be replaced by their absolute value. Although the forward pass can be batched, the *absolute* gradients must be accumulated over the mini-batch when backpropagating. All modern numerical libraries for deep learning are designed to efficiently propagate batches of inputs and accumulate *signed* gradients. Accumulating the absolute gradients therefore requires backpropagating each example by itself and computing the absolute gradient before backpropagating the next example. This is rather inefficient.

A much more efficient way to obtain a sensitivity estimate, is to compute an approximate per unit sensitivity by instead simply summing the signed gradients

$$\sum_{i=1}^{I} \nabla_{\mathbf{w}} NN(\mathbf{X}|\mathbf{w})_{ki} \ .$$

According to [54], it empirically improves performance to not average the summed signed gradient over the batch, i.e. not divide by I. Summing the signed gradient introduces gradient washout while not dividing by I helps keep the gradient magnitude larger, counteracting the washout.

Since the output layer generally has more than a single unit, i.e. it is a vector contrary to the usual scalar valued error function, both of these two sensitivity estimates result in a gradient for each weight, for each output unit. That is, K gradients are obtained for each weight. To handle this, the gradient from each output unit is interpreted as a component of a K-dimensional gradient vector and the total gradient, i.e. the sensitivity, is computed as the Euclidean length of this vector. This results in two variants of the safe mutation,

$$\mathbf{s}_{\text{abs}} = \sqrt{\sum_{k=1}^{K}\left(\frac{1}{I}\sum_{i=1}^{I} |\nabla_{\mathbf{w}} NN(\mathbf{X}|\mathbf{w})_{ki}|\right)^2} \tag{3.5.13}$$

$$\mathbf{s}_{\text{sum}} = \sqrt{\sum_{k=1}^{K}\left(\sum_{i=1}^{I} \nabla_{\mathbf{w}} NN(\mathbf{X}|\mathbf{w})_{ki}\right)^2} \ . \tag{3.5.14}$$

The absolute gradient variant is computationally much more expensive than the signed gradient variant due to the reasons discussed above. Finally, the sampled perturbation is adjusted to be safe according to the computed sensitivities,

$$\boldsymbol{\epsilon}_{\text{safe}} = \frac{\boldsymbol{\epsilon}}{\mathbf{s}} \ . \tag{3.5.15}$$

3.5.4.4 Safe mutation through Hessian of divergence

A slightly different approach is to consider the gradient of the divergence defined in (3.5.9). This allows perturbing the weights in a way that results in a certain amount of divergence, as in the rescaling approach, but here based on gradients which introduces changes to the direction as well as scaling the magnitude.

However, there is a small catch. The gradient of the divergence can be written as

$$\begin{aligned}\nabla_{\mathbf{w}} D(\boldsymbol{\epsilon}|\mathbf{w}) &= \frac{1}{I}\nabla_{\mathbf{w}} \sum_{k=1}^{K}\sum_{i=1}^{I} (NN(\mathbf{X}|\mathbf{w})_{ki} - NN(\mathbf{X}|\mathbf{w}+\boldsymbol{\epsilon})_{ki})^2 \\ &= \frac{1}{I}\sum_{k=1}^{K}\sum_{i=1}^{I} \nabla_{\mathbf{w}}(NN(\mathbf{X}|\mathbf{w})_{ki} - NN(\mathbf{X}|\mathbf{w}+\boldsymbol{\epsilon})_{ki})^2 \ .\end{aligned}$$

By the chain rule, the term in the sum becomes

$$2(NN(\mathbf{X}|\mathbf{w})_{ki} - NN(\mathbf{X}|\mathbf{w}+\boldsymbol{\epsilon})_{ki})\nabla_{\mathbf{w}}(NN(\mathbf{X}|\mathbf{w})_{ki} - NN(\mathbf{X}|\mathbf{w}+\boldsymbol{\epsilon})_{ki}) \ . \tag{3.5.16}$$

When the gradient of the divergence is evaluated at the network's current weights $\boldsymbol{\epsilon} = 0$ this term is zero and as a result, the gradient is zero. Therefore, a second order approximation must be used.

A Taylor expansion of the gradient of the divergence about the current network's weights gives a second order approximation by

$$\begin{aligned}\nabla_{\mathbf{w}} D(0+\boldsymbol{\epsilon}_0|\mathbf{w}) &\approx \nabla_{\mathbf{w}} D(0|\mathbf{w}) + \nabla_{\mathbf{w}}\left[\nabla_{\mathbf{w}} D(0|\mathbf{w})^{\mathrm{T}}\boldsymbol{\epsilon}_0\right] \\ &= 0 + \nabla_{\mathbf{w}}\left[\nabla_{\mathbf{w}} D(0|\mathbf{w})^{\mathrm{T}}\right]\boldsymbol{\epsilon}_0 \\ &= \mathbf{H}_{\mathbf{w}} D(0|\mathbf{w})\boldsymbol{\epsilon}_0 \end{aligned} \tag{3.5.17}$$

since the gradient of the divergence at $\boldsymbol{\epsilon} = 0$ is zero everywhere[12]. Note that the Hessian only needs to be computed as part of a matrix-vector product, $\mathbf{H}_{\mathbf{w}} D(0|\mathbf{w})\boldsymbol{\epsilon}_0$, effectively giving the curvature of the divergence in the direction of the perturbation.

The resulting sensitivities are then computed similarly to before,

$$\mathbf{s}_{\text{SO}} = \sqrt{|\mathbf{H}_{\mathbf{w}} D(0|\mathbf{w})|\boldsymbol{\epsilon}} \tag{3.5.18}$$

for an unscaled perturbation $\boldsymbol{\epsilon}$ which is then adjusted by $\mathbf{s}_{\text{SO}}$ according to (3.5.15).

[12]There can be some confusion about the Hessian and the Laplacian. The Hessian operator can be defined as the *outer* product of the gradient operator (nabla)

$$\mathbf{H}_{\mathbf{w}} = \nabla_{\mathbf{w}} \nabla_{\mathbf{w}}{}^{\mathrm{T}}$$

while the Laplacian can be defined as the *inner* product of the gradient operator

$$\nabla^2_{\mathbf{w}} = \nabla_{\mathbf{w}}{}^{\mathrm{T}} \nabla_{\mathbf{w}} .$$

While the Hessian contains all possible second order partial derivatives, the Laplacian is the trace of the Hessian; a scalar sum of unmixed second order partial derivatives. Here, the Hessian is the operator that arises.

3.5.4.5 Numerical considerations

The computation of the sensitivities derived above requires some consideration to be put into the numerical properties. Both $\mathbf{s}_{\text{abs}}$ and $\mathbf{s}_{\text{sum}}$ are computed using at least one non-normalized sum which runs the risk of numerical overflow. The implementation of the sensitivity computation used here sets any sensitivity which has overflowed to 1. This results in that sensitivity not influencing the perturbation and the perturbation being applied as sampled. Another perhaps more sensible approach would be to set the perturbation to zero and not change the highly sensitive parameter at all. This however risks freezing that parameter indefinitely if the sensitivity does not change considerably. The sensitivity overflow problem occurs extremely rarely.

Another issue is that the numerical value of the sensitivities is somewhat arbitrary. This is most the case with the signed gradient variant which approximates the absolute gradient variant, but the problem also exists for the absolute gradient variant since the K sensitivity gradients are summed at every weight. A normalization can be applied to scale the sensitivities into the range $[0, 1]$

$$\hat{\mathbf{s}} = \frac{\mathbf{s} - \min\{\mathbf{s}\}}{\max\{\mathbf{s}\} - \min\{\mathbf{s}\}} \,. \tag{3.5.19}$$

which is a natural range for a scaling applied by division. It does however, not remove the risk of the sensitivities being zero or very close to zero which results in very large perturbations. This problem has been handled by clamping the sensitivities below by a small constant,

$$\hat{\mathbf{s}} \leftarrow \min\{\xi, \hat{\mathbf{s}}\} \tag{3.5.20}$$

Throughout this thesis, the smallest allowed sensitivity has been $\xi = 10^{-2}$ which has been found to yield stable rescaling.

Obtaining $\boldsymbol{\epsilon}_{\text{safe}}$ by rescaling with sensitivities in the range $[10^{-2}, 1]$ does not change the mean of the perturbation since it is zero but it does change the variance. In order to decouple safe mutation from the variance of the perturbation, the perturbation is finally divided by its new variance and multiplied by the wanted variance to get back the original variance in a way that corresponds to the representation and dimensionality of the search distribution. None of these considerations were described in [54].

3.6 Methods for variance reduction

Monte Carlo estimators, such as the VO gradient, are well-known to have high variance. However, many methods exist to alleviate this problem. This section considers such methods for reducing the variance of the VO gradient estimator. First, Section 3.6.1 derives the variance of the estimator. Section 3.6.2 derives and discusses antithetic sampling, Section 3.6.3 introduces the method of common random numbers (CRN) while Section 3.6.4 applies a reparameterization and exploits the structure of FNNs to obtain a lower variance estimator in a novel way.

3.6.1 Variance of VO estimators

Consider the variance of the VO gradient estimator for a general search distribution $p(\mathbf{x}|\boldsymbol{\theta})$ which was derived in (3.3.7). Consider the gradient for $\boldsymbol{\theta}$ and compute its variance as

$$\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})] = \mathrm{Var}\left[\frac{1}{N}\sum_{n=1}^{N} f(\boldsymbol{\epsilon}_n)\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\epsilon}_n|\boldsymbol{\theta})\right] \tag{3.6.1}$$

where $\boldsymbol{\epsilon}$ is used in place of $\mathbf{x}$ to stay in the perturbation terminology. The variance can be expanded in variance and covariance terms. Let $g(\boldsymbol{\epsilon}_n) = f(\boldsymbol{\epsilon}_n)\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\epsilon}_n|\boldsymbol{\theta})$ for compactness. Then,

$$\begin{aligned}
\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})] &= \mathrm{Var}\left[\frac{1}{N}\sum_{n=1}^{N} g(\boldsymbol{\epsilon}_n)\right] \\
&= \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} \mathrm{Cov}[g(\boldsymbol{\epsilon}_i), g(\boldsymbol{\epsilon}_j)] \\
&= \underbrace{\frac{1}{N^2}\sum_{i=1}^{N} \mathrm{Var}[g(\boldsymbol{\epsilon}_i)]}_{\text{diagonal}} + \underbrace{\frac{2}{N^2}\sum_{i=1}^{N}\sum_{j=i+1}^{N} \mathrm{Cov}[g(\boldsymbol{\epsilon}_i), g(\boldsymbol{\epsilon}_j)]}_{\text{off-diagonal}} && (3.6.2) \\
&= \frac{1}{N}\mathrm{Var}[g(\boldsymbol{\epsilon})] + \frac{2}{N^2}\sum_{i=1}^{N}\sum_{j=i+1}^{N} \mathrm{Cov}[g(\boldsymbol{\epsilon}_i), g(\boldsymbol{\epsilon}_j)] && (3.6.3)
\end{aligned}$$

when using regular sampling, $\boldsymbol{\epsilon}_i$ and $\boldsymbol{\epsilon}_j$ are IID random variables and $\mathrm{Cov}[\boldsymbol{\epsilon}_i, \boldsymbol{\epsilon}_j] = 0$ for all i and j where $i \neq j$. For the independence it holds more generally that $g(\boldsymbol{\epsilon}_i)$ and $h(\boldsymbol{\epsilon}_j)$ are independent for any functions g and h. A simple argument for this is as follows: Since $\boldsymbol{\epsilon}_i$ contains no information about $\boldsymbol{\epsilon}_j$, neither will $g(\boldsymbol{\epsilon}_i)$ contain any information about $h(\boldsymbol{\epsilon}_j)$, and vice versa, regardless the functions. With $g(\boldsymbol{\epsilon}_i) = f(\boldsymbol{\epsilon}_i)\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\epsilon}_i|\boldsymbol{\theta})$ and $h(\boldsymbol{\epsilon}_j) = g(\boldsymbol{\epsilon}_j)$,

$$\mathrm{Cov}[g(\boldsymbol{\epsilon}_i), g(\boldsymbol{\epsilon}_j)] = 0 \ , \quad \forall\, i, j \text{ with } i \neq j \ . \tag{3.6.4}$$

The variance of the gradient estimate then reduces to include only the diagonal of the covariance matrix, i.e. the variances

$$\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})] = \frac{1}{N}\mathrm{Var}[g(\boldsymbol{\epsilon})] \ . \tag{3.6.5}$$

Thus, the variance of the VO gradient estimator is $\mathcal{O}(1/N)$ where N is the number of perturbations. This result and the following results for antithetic sampling hold when considering a single batch example. The case of mini-batches with additive objective functions are considered in Section 3.6.4. This result holds regardless of the choice of search distribution as long as the samples are IID.

3.6.2 Antithetic sampling

A simple but powerful method for variance reduction in stochastic estimation is that of antithetic sampling. This method was successfully applied for stochastic estimation of gradients using an isotropic Gaussian search distribution with fixed variance in [90] although no theoretical argument for its efficiency was provided.

For every sampled path $\{\epsilon_1, \epsilon_2, \dots, \epsilon_n\}$, antithetic sampling simply consists of also taking the so-called antithetic or mirrored path. This effectively doubles the number of samples obtained by sampling which may be beneficial in it self in the case where sampling is expensive.

Considering stochastic gradient estimation using a search distribution, the antithetic samples form a vector in the search space that points in the exact opposite direction of the original samples. Intuitively speaking, in the case that the original samples proved to give lower function values, the antithetic sampling then probes the opposite path and if it gives higher function values, the variance is reduced due to negative covariance.

For the IID sampling, the covariance was shown to be zero. However, a negative covariance will result in a reduction of the variance compared to regular IID sampling. If $\boldsymbol{\epsilon}_i$ and $\boldsymbol{\epsilon}_j$ are chosen to not all be independent, the covariance becomes nonzero. One way to correlate the perturbations is to choose the mirrored perturbation given by $\tilde{\boldsymbol{\epsilon}}_i = c - \boldsymbol{\epsilon}_i$, for every $\boldsymbol{\epsilon}_i$, where c is the center of symmetry of the search distribution. This is the method of antithetic sampling. When the search distribution is symmetric around zero, as can for instance be made the case for any Gaussian perturbation by the reparameterization trick, $c = 0$ and $\tilde{\boldsymbol{\epsilon}}_i = -\boldsymbol{\epsilon}_i$ to give

$$\mathrm{Cov}[\boldsymbol{\epsilon}_i, \tilde{\boldsymbol{\epsilon}}_i] = \mathrm{Cov}[\boldsymbol{\epsilon}_i, -\boldsymbol{\epsilon}_i] = -\mathrm{Var}[\boldsymbol{\epsilon}_i] \ . \tag{3.6.6}$$

To compute the total covariance between perturbations, $\mathrm{Cov}[g(\boldsymbol{\epsilon}_i), g(\boldsymbol{\epsilon}_j)]\ \forall i, j$, consider first the covariance between any antithetic pair, $\mathrm{Cov}[g(\boldsymbol{\epsilon}), g(\tilde{\boldsymbol{\epsilon}})]$. This is most easily done by decomposing $g(\cdot)$ in its odd and even components as follows. An even function e satisfies $e(x) = e(-x)$ and an odd function o satisfies $o(x) = -o(-x)$. It follows that any function, here g, can be decomposed into even and odd parts by writing

$$g_e(\boldsymbol{\epsilon}) = \frac{g(\boldsymbol{\epsilon}) + g(-\boldsymbol{\epsilon})}{2} \tag{3.6.7a}$$

$$g_o(\boldsymbol{\epsilon}) = \frac{g(\boldsymbol{\epsilon}) - g(-\boldsymbol{\epsilon})}{2} \tag{3.6.7b}$$

$$g(\boldsymbol{\epsilon}) = g_e(\boldsymbol{\epsilon}) + g_o(\boldsymbol{\epsilon}) \tag{3.6.7c}$$

where g_e denotes the even component and g_o denotes the odd. The even and odd components are orthogonal in the sense that $\int g_{\mathrm{e}}(\boldsymbol{\epsilon}) g_o(\boldsymbol{\epsilon})\, \mathrm{d}\boldsymbol{\epsilon} = 0$. The covariance can then be calculated by the following manipulations.

$$\begin{aligned} \mathrm{Cov}[g(\boldsymbol{\epsilon}), g(-\boldsymbol{\epsilon})] &= \mathrm{Cov}[g_e(\boldsymbol{\epsilon}) + g_o(\boldsymbol{\epsilon}), g_e(-\boldsymbol{\epsilon}) + g_o(-\boldsymbol{\epsilon})] \\ &= \mathrm{Var}[g_e(\boldsymbol{\epsilon})] - \mathrm{Var}[g_o(\boldsymbol{\epsilon})] \ . \end{aligned} \tag{3.6.8}$$

Evidently, the covariance is negative if the odd component has larger variance than the even component which results in lower total variance compared to IID sampling. Since the covariances are only nonzero for the antithetic pairs, the covariance matrix will be block diagonal with 2×2 blocks, assuming that the mirrored sample is introduced as $\boldsymbol{\epsilon}_{i+1} = -\boldsymbol{\epsilon}_i$ where $i \in [0, 2, 4, \dots, N/2]$ and N is even. There will then be $N/2$ nonzero covariances in the lower triangular matrix such that (3.6.3) reduces to

$$\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] = \frac{1}{N}\mathrm{Var}[g(\boldsymbol{\epsilon})] + \frac{1}{N}\mathrm{Cov}[g(\boldsymbol{\epsilon}), g(-\boldsymbol{\epsilon})] \tag{3.6.9}$$

where $\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a$ denotes the antithetic VO gradient estimate. In fact, if $g_e(\boldsymbol{\epsilon}) = 0$ such that $g(\boldsymbol{\epsilon}) = g_o(\boldsymbol{\epsilon})$ is purely odd, then $\mathrm{Cov}[g_o(\boldsymbol{\epsilon}), g_o(-\boldsymbol{\epsilon})] = -\mathrm{Var}[g_o(\boldsymbol{\epsilon})]$ and $\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] = 0$. If instead $g(\boldsymbol{\epsilon}) = g_e(\boldsymbol{\epsilon})$, then $\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] = 2\mathrm{Var}[g_e(\boldsymbol{\epsilon})]$.

$$\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] = \begin{cases} 0 & \text{if } g(\boldsymbol{\epsilon}) = g_o(\boldsymbol{\epsilon}) \\ 2\mathrm{Var}[g_e(\boldsymbol{\epsilon})] & \text{if } g(\boldsymbol{\epsilon}) = g_e(\boldsymbol{\epsilon}) \end{cases} \tag{3.6.10}$$

These two extremes are respectively the best and worst case scenarios for antithetic sampling. At best, the variance is reduced to zero and the gradient is exact after evaluating only at one pair of perturbations. At worst, the variance of the gradient is doubled.

Instead of writing the regular estimator and taking $\boldsymbol{\epsilon}_{i+1} = -\boldsymbol{\epsilon}_i$, alternatively one can write the antithetic VO gradient estimate directly

$$\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a = \frac{1}{N} \sum_{n=1}^{N/2} g(\boldsymbol{\epsilon}_n) + g(-\boldsymbol{\epsilon}_n) \tag{3.6.11}$$

where N again must be an even integer. This can be written in terms of the even component of the function by (3.6.7a)

$$\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a = \frac{2}{N} \sum_{n=1}^{N/2} g_e(\boldsymbol{\epsilon}_n)\ . \tag{3.6.12}$$

Now, the variance of the antithetic estimator becomes

$$\begin{aligned} \mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] &= \frac{4}{N^2} \mathrm{Var}\left[\sum_{n=1}^{N/2} g_e(\boldsymbol{\epsilon}_n)\right] \\ &= \frac{4}{N^2} \sum_{n=1}^{N/2} \mathrm{Var}[g_e(\boldsymbol{\epsilon}_n)] \\ &= \frac{2}{N} \mathrm{Var}[g_e(\boldsymbol{\epsilon})] \end{aligned} \tag{3.6.13}$$

since the covariance is zero in this formulation. From (3.6.5), the variance of the ordinary VO gradient estimator can similarly be split into even and odd function components which gives

$$\mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] = \frac{1}{N}(\mathrm{Var}[g_e(\boldsymbol{\epsilon})] + \mathrm{Var}[g_o(\boldsymbol{\epsilon})]) \tag{3.6.14}$$

since $\mathrm{Cov}[g_e(\boldsymbol{\epsilon}_n), g_o(\boldsymbol{\epsilon}_n)] = 0$ due to orthogonality. Clearly then, antithetic sampling trades in the variance of the odd component for the variance of the even component. In summary one can write

$$\begin{bmatrix} \mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})] \\ \mathrm{Var}[\nabla_{\boldsymbol{\theta}} U(\boldsymbol{\theta})_a] \end{bmatrix} = \frac{1}{N} \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} \mathrm{Var}[g_e(\boldsymbol{\epsilon})] \\ \mathrm{Var}[g_o(\boldsymbol{\epsilon})] \end{bmatrix} \tag{3.6.15}$$

which is a generalization of (3.6.10). Evidently, there is a potentially large variance reduction to gain from using antithetic sampling on a function that is primarily odd or at least has most of its variance in the odd component.

By expanding $g(\boldsymbol{\epsilon})$ in a Taylor series and plugging it into (3.6.11) there is another perspective to antithetic sampling to be noted. Consider the univariate case where ϵ is scalar. The argument holds for the multivariate case as well. The Taylor series can be written as

$$g(\epsilon) = \sum_{i=0}^{\infty} \frac{g^{(i)}(0)}{i!} \epsilon^i\ . \tag{3.6.16}$$

With $\tilde{\epsilon} = -\epsilon$ then

$$\begin{aligned} g(\epsilon) + g(\tilde{\epsilon}) &= g(\epsilon) + g(-\epsilon) && (3.6.17) \\ &= \sum_{i=0}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i + \sum_{i=0}^{\infty} \frac{g^{(i)}(0)}{i!}(-\epsilon)^i && (3.6.18) \\ &= \sum_{i=0}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i + \left(\sum_{i=0,2,\ldots}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i - \sum_{i=1,3,\ldots}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i \right) && (3.6.19) \\ &= \sum_{i=0,2,\ldots}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i + \sum_{i=0,2,\ldots}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i && (3.6.20) \\ &= 2 \sum_{i=0,2,\ldots}^{\infty} \frac{g^{(i)}(0)}{i!}\epsilon^i \ . && (3.6.21) \end{aligned}$$

Clearly then, the antithetic sampling serves to cancel out odd derivatives of the Taylor series of g while doubling the contribution of the even derivatives.

In VO, the function g is composed by the objective function f multiplied by the perturbation $\boldsymbol{\epsilon}$. In the reinforcement learning setting, the objective is expected to exhibit some degree of anti-symmetry while it is unlikely to be purely odd. For instance, in highly symmetric mazes walking in the x direction can be equivalent to walking in the $-x$ direction but generally it is not. This thesis will not attempt to analyze how anti-symmetry holds through the multiplication by $\boldsymbol{\epsilon}$ and perturbation of the NN parameters by $\boldsymbol{\epsilon}$, however, it seems there is reason to conjecture that antithetic sampling may reduce the gradient estimate variance.

An additional note can be made about the subspace spanned by the VO perturbations when using antithetic sampling. Without antithetic sampling, N random perturbations are likely to span an N dimensional subspace of the d dimensional search space. When using antithetic sampling, half of these perturbations are replaced by the negative of an already existing perturbation. As such, the antithetic perturbations are linearly dependent on the existing perturbations. Consequently, the subspace will be maximally $N/2$ dimensional for a population of N perturbations.

3.6.3 Common random numbers

The method of CRN is usually applied when comparing two or more stochastic simulations of some system for a some different sets of hyperparameters. The simulations are then made such that any stochastic elements of the systems share the same seed for all the simulations. That is, the stochastic elements in each simulation are identical, at least initially. For instance, if random noise is added in an otherwise deterministic simulation of a system, the sequence of random numbers that represent the noise is the same for all the simulations. The simulations may still differ due to the different hyperparameter settings. The reduced amount of randomness in turn reduces variance [26].

In the application of VO to supervised learning, the method of CRN translates to evaluating each of the perturbations on the same mini-batch of training examples. Specifically, in classification, if the batch over-represents some class which the model is particularly bad at correctly classifying, any perturbation that outperforms the others has a good chance of improving on classifying that class. If the batches were different, some models may be evaluated too harshly due to being handed a difficult batch compared to the batches given to other perturbations.

In the RL setting, it corresponds to evaluating the perturbations on environments which have been seeded such that they have the same initial condition. In RL, the episodes that follow may be quite different despite the shared initial state due to the different policies represented by the perturbed networks.

3.6.4 Local reparameterization

The local reparameterization method [46] was presented as a means of reducing the variance of the stochastic gradient variational Bayes (SGVB) method [47] which seeks to perform efficient approximate inference and learning in directed graphical models in the fully Bayesian setting. Here, the approach is applied to reduce the variance of the VO estimator. This section describes an untried idea that should be subject to further analysis and experimental validation.

3.6.4.1 An alternative Monte Carlo estimator for the upper bound

The local reparameterization is useful in situations where the objective $f(\mathbf{x})$ is composed of a sum of individual terms, typically averaged. Then

$$f(\mathbf{x}) = \frac{1}{|\mathcal{B}|}\sum_{b\in\mathcal{B}} f_b(\mathbf{x}) \tag{3.6.22}$$

where $f_i(\mathbf{x})$ denotes the objective evaluated at the network with parameters $\mathbf{x}$ on the b'th batch example. This is the case for error functions in supervised learning (2.3.1). In such a case, the VO upper bound can be rewritten as

$$U(\boldsymbol{\theta}) = \mathrm{E}[f(\mathbf{x})]_{p(\mathbf{x}|\boldsymbol{\theta})} = \frac{1}{|\mathcal{B}|}\sum_{b\in\mathcal{B}} \mathrm{E}[f_b(\mathbf{x})]_{p(\mathbf{x}|\boldsymbol{\theta})} \ . \tag{3.6.23}$$

An unbiased Monte Carlo estimator for $U(\boldsymbol{\theta})$ is then

$$U(\boldsymbol{\theta}) \approx \frac{1}{N|\mathcal{B}|}\sum_{n=1}^{N}\sum_{b\in\mathcal{B}} f_b(\mathbf{x}_n) \tag{3.6.24}$$

which is effectively the same result as previously seen for the gradient. However, rather than using the same weights for all examples in a batch, one can now sample new weights for each example. This results in the following alternative unbiased Monte Carlo estimator.

$$\tilde{U}(\boldsymbol{\theta}) \approx \frac{1}{N|\mathcal{B}|}\sum_{n=1}^{N}\sum_{b\in\mathcal{B}} f_b(\mathbf{x}_{bn}) \ . \tag{3.6.25}$$

3.6.4.2 Variance of the upper bound estimators

The variance of the estimator in (3.6.24) is

$$\begin{aligned}
\mathrm{Var}[U(\theta)] &= \frac{1}{N^2|\mathcal{B}|^2}\sum_{n,n'=1}^{N}\sum_{b,b'\in\mathcal{B}}\mathrm{Cov}[f_b(\mathbf{x}_n), f_{b'}(\mathbf{x}_{n'})] \\
&= \frac{1}{N^2|\mathcal{B}|^2}\sum_{n=1}^{N}\sum_{b\in\mathcal{B}}\mathrm{Var}[f_b(\mathbf{x}_n)] + \frac{1}{N^2|\mathcal{B}|^2}\sum_{\substack{n,n'=1\\ n\neq n'}}^{N}\sum_{\substack{b,b'\in\mathcal{B}\\ b\neq b'}}\mathrm{Cov}[f_b(\mathbf{x}_n), f_{b'}(\mathbf{x}_{n'})] \\
&= \frac{1}{N|\mathcal{B}|^2}\sum_{b\in\mathcal{B}}\mathrm{Var}[f_b(\mathbf{x})] + \frac{N-1}{N|\mathcal{B}|^2}\sum_{\substack{b,b'\in\mathcal{B}\\ b\neq b'}}\mathrm{Cov}[f_b(\mathbf{x}), f_{b'}(\mathbf{x})] \\
&= \frac{1}{N|\mathcal{B}|}\mathrm{Var}[f_b(\mathbf{x})] + \frac{N-1}{N}\frac{|\mathcal{B}|-1}{|\mathcal{B}|}\mathrm{Cov}[f_b(\mathbf{x}), f_{b'}(\mathbf{x})] \qquad (3.6.26)
\end{aligned}$$

where it is used that the within-batch covariance is the same on average, as for the perturbations. This result may at first seem to contradict (3.6.5) but in fact does not. In (3.6.5), only a single batch-example was considered while here, an entire mini-batch is studied including the potentially non-zero between-batch covariance. It should be noted that the variance scales as $\mathcal{O}(1/N|\mathcal{B}|)$ while the covariance scales as $\mathcal{O}(1)$ for large enough N and $|\mathcal{B}|$ and as such effectively lower bounds $\mathrm{Var}[U(\theta)]$.

By the same manipulations, the variance of the estimator in (3.6.25) that samples new parameters for each batch example is

$$\mathrm{Var}\left[\tilde{U}(\theta)\right] = \frac{1}{N|\mathcal{B}|}\mathrm{Var}[f_b(\mathbf{x}_b)] + \frac{N-1}{N}\frac{|\mathcal{B}|-1}{|\mathcal{B}|}\mathrm{Cov}[f_b(\mathbf{x}_b), f_{b'}(\mathbf{x}_{b'})]$$

but here it is noted that the covariance term can be rewritten

$$\begin{aligned}
\mathrm{Var}\left[\tilde{U}(\theta)\right] &= \frac{1}{N|\mathcal{B}|}\mathrm{Var}[f_b(\mathbf{x}_b)] + \frac{N-1}{N}\frac{|\mathcal{B}|-1}{|\mathcal{B}|}\mathrm{E}[(f_b(\mathbf{x}_b)-U_b)(f_{b'}(\mathbf{x}_{b'})-U_{b'})]_{p(\mathbf{x}_b|\theta),p(\mathbf{x}_{b'}|\theta)} \\
&= \frac{1}{N|\mathcal{B}|}\mathrm{Var}[f_b(\mathbf{x}_b)] + \frac{N-1}{N}\frac{|\mathcal{B}|-1}{|\mathcal{B}|}\left(\mathrm{E}[f_b(\mathbf{x}_b)]_{p(\mathbf{x}_b|\theta)}-U_b\right)\left(\mathrm{E}[f_{b'}(\mathbf{x}_{b'})]_{p(\mathbf{x}_{b'}|\theta)}-U_{b'}\right) \\
&= \frac{1}{N|\mathcal{B}|}\mathrm{Var}[f_b(\mathbf{x}_b)] \qquad (3.6.27)
\end{aligned}$$

where $U_b = \mathrm{E}[f_b(\mathbf{x}_b)]_{p(\mathbf{x}_b|\theta)}$. The covariance term is zero in the alternative estimator that samples new parameters for the model for each batch example.

3.6.4.3 Propagation of a distribution over activations

The derivation made to give (3.3.7) is followed again to obtain the Monte Carlo estimator for the gradient of the alternative estimator,

$$\nabla_\theta\tilde{U}(\theta) \approx \frac{1}{N|\mathcal{B}|}\sum_{n=1}^{N}\sum_{b\in\mathcal{B}} f_b(\mathbf{x}_{bn})\nabla_\theta \log p(\mathbf{x}_{bn}|\theta)\ , \qquad (3.6.28)$$

where $x_{b,n} \sim p(\mathbf{x}_{b,n}|\theta)$. From a computational point of view this estimator is not very efficient and especially not for modern deep learning software which is optimized to evaluate a single network on $|\mathcal{B}|$ training examples. The advantage of the formulation is that the variance of the estimator is reduced from $\mathcal{O}(1)$ to $\mathcal{O}(1/N|\mathcal{B}|)$ because each term is evaluated on an independent realization of the parameters [46].

To achieve computational efficiency, it is exploited that FNN models depend upon the weights and biases only through the activations, $\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{A}^{[l-1]}$ where $\mathbf{A}^{[l]} = \varphi\big(\mathbf{Z}^{[l]}\big)$, as discussed in Chapter 2. In elementwise notation this becomes $Z_{ib}^{[l]} = \sum_j W_{ij}^{[l]} A_{jb}^{[l-1]}$, where $Z_{ib}^{[l]}$ is the i'th element in the l'th layer activation of the b'th batch example. The following will consider the isotropic Gaussian search distribution $p(\mathbf{W}|\boldsymbol{\theta}) = \mathcal{N}\big(\mathbf{W}|\boldsymbol{\mu}, \sigma^2\mathbf{I}\big)$ with $\boldsymbol{\theta} = \big\{\boldsymbol{\mu}, \sigma^2\big\}$ and sample by reparameterization, $\mathbf{W} = \boldsymbol{\mu} + \sigma\boldsymbol{\epsilon}, \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The distribution of the activations can be inferred from the distribution of the weights drawn for each training example. Taking $\boldsymbol{\mu}$ and σ to be matrices of same shape as $\mathbf{W}$, this distribution is

$$q\Big(Z_{ib}^{[l]} \;\Big|\; \mathbf{A}^{[l-1]}, \boldsymbol{\theta}\Big) = \mathcal{N}\left(Z_{ib}^{[l]} \;\middle|\; \sum_j \mu_{ij}^{[l]} A_{jb}^{[l-1]}, \sum_j \sigma_{ij}^{[l]^2} A_{jb}^{[l-1]^2}\right) = \mathcal{N}\left(Z_{ib}^{[l]} \;\middle|\; m_{ib}^{[l]}, v_{ib}^{[l]}\right) \tag{3.6.29}$$

where m_{ib} and v_{ib} are respectively the mean and variance of the Gaussian activation distribution. By letting $\mathbf{Z}_{bn}$ denote the stacked vector of activations of all layers of the b'th batch example and n'th perturbation[13] and similarly for $\mathbf{A}_{bn}$ then

$$\nabla_{\boldsymbol{\theta}} \tilde{U}(\theta) \approx \frac{1}{N|\mathcal{B}|} \sum_{n=1}^{N} \sum_{b\in\mathcal{B}} f_b(\mathbf{Z}_{bn}) \nabla_{\boldsymbol{\theta}} \log q(\mathbf{Z}_{bn}|\mathbf{A}_{bn}, \boldsymbol{\theta}) \;, \tag{3.6.30}$$

where $Z_{ibn} = m_{ib} + \sqrt{v_{ib}}\,\xi_{ibn}$, with $\xi_{ibn} \sim \mathcal{N}(0,1)$ and $f(\mathbf{Z})$ is the error function from above now written in terms of the activations. With this, the Monte Carlo estimates for the derivatives are

$$\begin{aligned} \frac{\partial \tilde{U}(\boldsymbol{\theta})}{\partial \mu_{ij}^{[l]}} &\approx \frac{1}{N|\mathcal{B}|} \sum_{n=1}^{N} \sum_{b\in\mathcal{B}} f_b(\mathbf{Z}_{bn}) \frac{\xi_{ibn}}{\sqrt{v_{ib}^{[l]}}} A_{jb}^{[l-1]} \\ \frac{\partial \tilde{U}(\boldsymbol{\theta})}{\partial \sigma_{ij}^{[l]^2}} &\approx \frac{1}{N|\mathcal{B}|} \sum_{n=1}^{N} \sum_{b\in\mathcal{B}} f_b(\mathbf{Z}_{bn}) \frac{\xi_{ibn}^2 - 1}{2v_{ib}^{[l]}} A_{jb}^{[l-1]^2} \;. \end{aligned} \tag{3.6.31}$$

The complexity of a forward pass with this new approach is $\mathcal{O}(N|\mathcal{B}||\mathbf{Z}|)$ where $|\mathbf{Z}|$ is the number of activations/units in the network. The batch can be efficiently forward propagated in a single pass as described in Chapter 2. For comparison, the original approach has a complexity of $\mathcal{O}(N|\mathbf{W}|)$ while a naive implementation of local reparameterization has a complexity of $\mathcal{O}(N|\mathcal{B}||\mathbf{W}|)$ and additionally requires a forward pass for each batch example which is less efficient than the new approach. The local reparameterization thus improves computational efficiency by sampling activations rather than weights.

[13]Strictly speaking, here $\mathbf{Z}$ is a third order tensor with elements Z_{ibn} denoting the i'th element of the b'th batch example activation for the n'th perturbation. Activation vectors of all layers are then stacked into a single dimension, i.

With $\mathbf{A}^{[0]} = \mathbf{X}$, the forward pass in an FNN is

$$\mathbf{m}^{[l]} = \boldsymbol{\mu}^{[l]}\mathbf{A}^{[l-1]} \tag{3.6.32a}$$

$$\mathbf{v}^{[l]} = \left(\boldsymbol{\sigma}^{[l]}\mathbf{A}^{[l-1]}\right)^2 \tag{3.6.32b}$$

$$\mathbf{Z}^{[l]} = \mathbf{m}^{[l]} + \sqrt{\mathbf{v}^{[l]}} \odot \boldsymbol{\xi}^{[l]} \tag{3.6.32c}$$

$$\mathbf{A}^{[l]} = \varphi\left(\mathbf{Z}^{[l]}\right) , \tag{3.6.32d}$$

when vectorized appropriately[14]. This effectively forward propagates a distribution over activations without realizing the perturbed weights explicitly and instead perturbs the activations.

3.6.4.4 Alternative gradient estimator variance

In addition to improving computational efficiency, the local reparameterization also reduces the variance of the gradient estimator. Comparing the estimators in (3.6.31) with the ones derived earlier for the isotropic Gaussian in (3.3.44) it is evident that they are similar but the alternative version derived here perturbs in the activation space rather than the weight space. As was discussed in Section 3.2.4, estimating the gradient has an interpretation as estimating the covariance between the objective and the perturbations. Since the alternative estimator based on the local reparameterization perturbs the activation space its search space has much fewer dimensions. These and their respective perturbations will then individually have much higher correlation with the objective and so the covariance, and hence the gradient, will be easier to accurately estimate [46].

3.6.4.5 Relation to batch normalization

Batch normalization was introduced in Chapter 2. The local reparameterization derived here is not directly affected by the normalization applied to the activations by batch normalization. Likewise, batch normalization is not affected by the forward propagation of a distribution over activations and can simply be applied to these activations as usual.

In fact, it seems that batch normalization can benefit from the local reparameterization by exploiting the already computed mean and variance of the activations. That is, (2.3.39) and (2.3.40) can be replaced with

$$\overline{\mathbf{m}} = \mathrm{E}\left[\mathbf{m}\right]_{p(\mathbf{Z}|\mathbf{A})} = \frac{1}{|\mathcal{B}|}\sum_{b\in\mathcal{B}} \mathbf{m}_b \tag{3.6.33}$$

and

$$\overline{\mathbf{v}} = \mathrm{E}\left[\mathbf{v}\right]_{p(\mathbf{Z}|\mathbf{A})} = \frac{1}{|\mathcal{B}|}\sum_{b\in\mathcal{B}} \left(\mathbf{v}_b + (\mathbf{m}_b - \overline{\mathbf{m}})^2\right) + \mathcal{O}(|\mathcal{B}|^{-1}) \tag{3.6.34}$$

respectively, for each layer. This results in slightly reduced computation required since the running mean variance are no longer required although a mean is still computed over the activations.

[14]The activation mean and variance $\mathbf{m}^{[l]}$ and $\mathbf{v}^{[l]}$ both have dimensions of $\mathbf{A}^{[l]}$. The unperturbed network is parameterized by $\boldsymbol{\mu}^{[l]}$ which has dimensions of $\mathbf{W}^{[l]}$; as has the associated perturbation variance $\boldsymbol{\sigma}^2$.

3.6.4.6 Remarks on application to convolutional neural networks

In a CNN, the same kernel is applied across the entire input which results in the activations of a CNN layer being highly correlated. The covariance between input pixels i and i' will be $\sum_j \sigma_j^2 A_{jb} A_{j'b}$ where j are the pixels in the input corresponding to pixel i in the feature map. A single sample of the feature map activations thus requires diagonalizing a covariance matrix of size (#pixels $\times$ #pixels). This is rather inefficient even if structure in the matrix can be exploited. Therefore, it seems better to stick with the original approach for convolutional layers of the network.

CHAPTER 4
Experimental work

"It is shown that many particular choices among possible neurophysiological assumptions are equivalent, in the sense that for every net behaving under one assumption, there exists another net which behaves under the other and gives the same results, although perhaps not in the same time."

- McCulloch, W.S. and Pitts, W. H. (1943) [65]

4.1 Introduction

In addition to the theoretical work presented in the preceding chapters, this thesis has spent some effort on implementing and experimenting with the different methods, algorithms and augmentations. The algorithms have been implemented in Python using parts of the PyTorch deep learning library. Brief explanations and references to full code repositories are given in Appendix A. The experiments were run on the high performance computing (HPC) cluster at DTU. This section presents the results of the experimental work.

Section 4.2 presents the problems used for the benchmarking and while the different NN architectures used are described in Section 4.3. The general computational complexity and scaling properties of the VO algorithm are examined for both a supervised problem and a reinforcement learning problem in Section 4.4. Section 4.5 examines the influence of the model augmentations of batch normalization and dropout along with safe mutation. In Section 3.6, the different methods for reducing the variance of the gradient are examined- This includes gradient momentum in 4.6.1 and antithetic sampling in 4.6.2. Finally, the effect of adapting the variance using different Gaussian search distributions is evaluated in Section 4.7.

It should be noted that the number of hyperparameters combined with variations on the VO algorithm gives a large number of potential experiments to conduct. This thesis has only performed a small subset of all the possible experiments. The chosen experiments were selected based on what had the highest potential to show improvement to the algorithm.

4.2 Benchmark problems

4.2.1 Supervised learning benchmark

Within the supervised learning setting, the classical problem of handwritten digit recognition based on the Modified National Institute of Standards and Technology database (MNIST) dataset [50] was chosen. This is a small and relatively simple data set that has allowed for

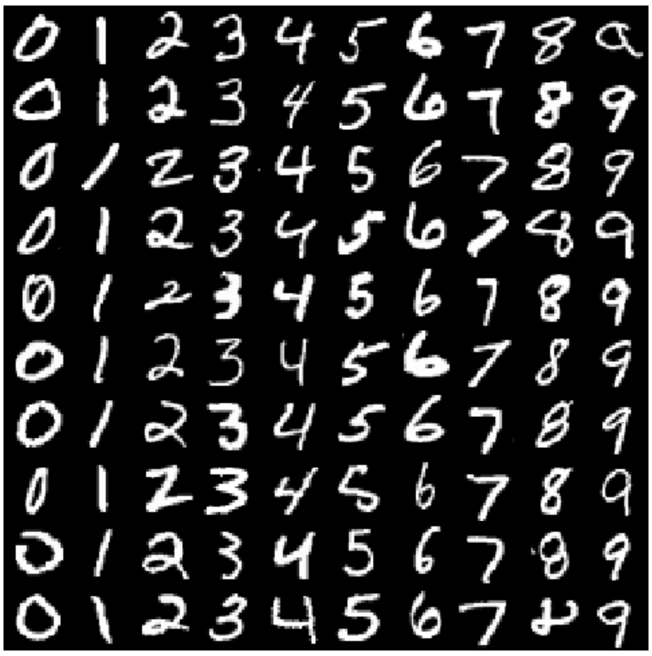


Figure 4.1: Examples from the MNIST dataset sorted by target values from 0 on the left and 9 on the right. As is well-known, a few examples from the dataset are very hard to correctly classify, note for example the bottom most 8 and the top most 9.

rapid prototyping and aided in algorithm development, historically and in this thesis. Even though it has recently received criticism for being too simple, it remains one of the most widely used benchmarks within deep learning [121]. The training data consists of 60.000 greyscale images of handwritten digits between 0 and 9 and the associated label. Each image has 28×28 pixels and represents a single digit. The test data consists of 10.000 images and associated labels. In this thesis, no data augmentation has been applied to the data set. Examples from the MNIST dataset can be seen in Figure 4.1.

To the author's best knowledge, the currently best performing non-ensemble NN model on MNIST is the deep FNN using DropConnect [116, 33] which achieved 99.79% classification accuracy also exploiting extreme data augmentation. An accuracy of 99.76% was achieved in [13] which uses a max-out network in network [59] model with no data augmentation. Using backpropagation and the network to be used for MNIST in this thesis (introduced in Section 4.3.1 and shown in Listing B.1) the best achievable accuracy is about 99.1% using SGD with momentum training for 20 epochs. With the Adam algorithm, accuracy can reach 99.2% for 20 epochs.

4.2.2 Reinforcement learning benchmarks

Within the reinforcement learning setting, focus has been primarily on two environments from the OpenAI Gym toolkit [10]. The first environment is the Atari-2600 game of Freeway[1] in

[1] `http://en.wikipedia.org/wiki/Freeway_(video_game)`, `http://gym.openai.com/envs/Freeway-v0/`

which the agent controls a chicken that must be brought to pass a busy ten lane highway by moving either up or down. If hit by a car, the chicken is pushed either slightly back or moved to the starting position. A reward of 1 is given for each successful passage and the total score is the number of passages within a time slot of 2 minutes and 16 seconds. In [90], using a the fixed variance ES similar to VO, a score of 31 was achieved when averaged over 10 re-runs with the DQN network of [67] encoding a deterministic policy. This network is also used for RL in this thesis

The second RL environment is the Atari game of Seaquest[2]. Here, the agent controls a submarine which must fend off enemies using torpedoes while rescuing divers by moving to their location. The submarine's oxygen supply is limited and the agent must occasionally bring it to the surface to refill and to receive points from rescued divers. The submarine can move in four directions and fire torpedoes. Seaquest is considered harder than Freeway due to the larger action space and more complicated objective. In [90], a score of 1390.0 was achieved in the same way as for Freeway.

In this thesis, VO is used to optimize a policy which maps from the environment's observation space to its action space. The policy is parameterized by a NN model, in this case also called a policy network [102]. The policy is deterministic in that the most probable action, as computed by the policy network, is always taken. Due to computational limitations, the number of experiments made in the reinforcement setting is fewer than in the supervised setting.

In general, hyperparameter optimization has been performed manually and heuristically. Ideally, random hyperparameter search [7] would have been employed to find optimal hyperparameters but was not due to limited computational resources.

4.2.3 Preprocessing of MNIST

Although no data augmentation is applied to the MNIST dataset, the data is normalized before being input to the NN. That is, for each digit image, the average pixel value is computed along with the variance and these are aggregated to a mean and variance for the full dataset. For MNIST, this could be done in a single computation since the entire data set fits in memory. However, since larger datasets were also tried out, the computation was implemented in an online manner according to the algorithm proposed by [117].

The algorithm consists of updating the mean as usual and the variance through online updates to the squared sample distance from the mean. For the mean

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n} \ . \tag{4.2.1}$$

Likewise, the update for the variance is

$$s_n^2 = \frac{(n-2)}{(n-1)} s_{n-1}^2 + \frac{(x_n - \bar{x}_{n-1})^2}{n} \ , \quad n > 1 \ .$$

This online update of the variance is prone to numerical instability which is avoided by updating the squared sample distance from the mean

$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_n)^2$$

[2]`http://en.wikipedia.org/wiki/Seaquest_(video_game)`, `http://gym.openai.com/envs/Seaquest-v0/`

and then computing

$$s_n^2 = \frac{M_{2,n}}{n-1} \ .$$

These formulas have been generalized to the case of multiple examples being added at each online update [12]. Here, this is needed to avoid looping over the $28 \times 28 = 784$ pixels of each image. Let A and B denote two samples with n_A and n_B examples respectively and a mean, variance and squared sample distance from the mean denoted by $\bar{x}_A$, $\bar{x}_B$, s_A^2, s_B^2, $M_{2,A}$, $M_{2,B}$, respectively. The update formulas for the aggregated sample X are then

$$\begin{aligned}
\delta &= \bar{x}_B - \bar{x}_A \\
\bar{x}_X &= \frac{n_A}{n_X}\bar{x}_A + \frac{n_B}{n_X}\bar{x}_B \\
M_{2,X} &= M_{2,A} + M_{2,B} + \delta^2 \frac{n_A n_B}{n_X} \\
s_X^2 &= \frac{M_{2,X}}{n_X - 1}
\end{aligned}$$

where $n_X = n_A + n_B$.

The computed mean and standard deviation for the MNIST dataset were then used to normalize each input pixel x_i as

$$\hat{x}_i = \frac{x_i - \bar{x}}{s} \ .$$

4.2.4 Preprocessing of Atari environments

Atari environments are preprocessed in the same way as in [68]. A frame from an Atari game is a $210 \times 160 \times 3$ pixels RGB image. Due to the limited number of sprites an Atari 2600 was able to display at once, some objects in games occur only in even frames while others occur only in odd frames [68]. To remove this flickering, a single frame is encoded as the maximum value of each pixel of the most recent frame and the previous frame. The frame is then converted to greyscale and resized to 84×84 using a pixel area relation which gives moiré-free results, is fast and yields good results for image downsampling [76]. This reduces the dimensionality of the frame by more than a factor of ten from $210 \times 160 \times 3 = 100800$ pixels to $84 \times 84 = 7056$ pixels. The input given to the NN model is the four most recent frames seen such that the input dimension is $4 \times 84 \times 84$. This gives the model directional information about movement in the images without the use of recurrent units.

An illustration of the results of this preprocessing can be seen for the game of Space Invaders in Figure 4.2. Note that the fired beam is longer in **(b)** compared to **(a)**. This is a side effect of combining successive images by pixel maximum which slightly elongates moving objects.

4.3 Network architectures

4.3.1 Architecture for MNIST

For the MNIST problem, a simple CNN with 22.000 parameters has been used. Overall, the network has two convolutional layers each followed by batch normalization, max pooling and a ReLU nonlinearity. Fully connected layers follow; the first with batch normalization and ReLU nonlinearity; the second with log-softmax nonlinearity.

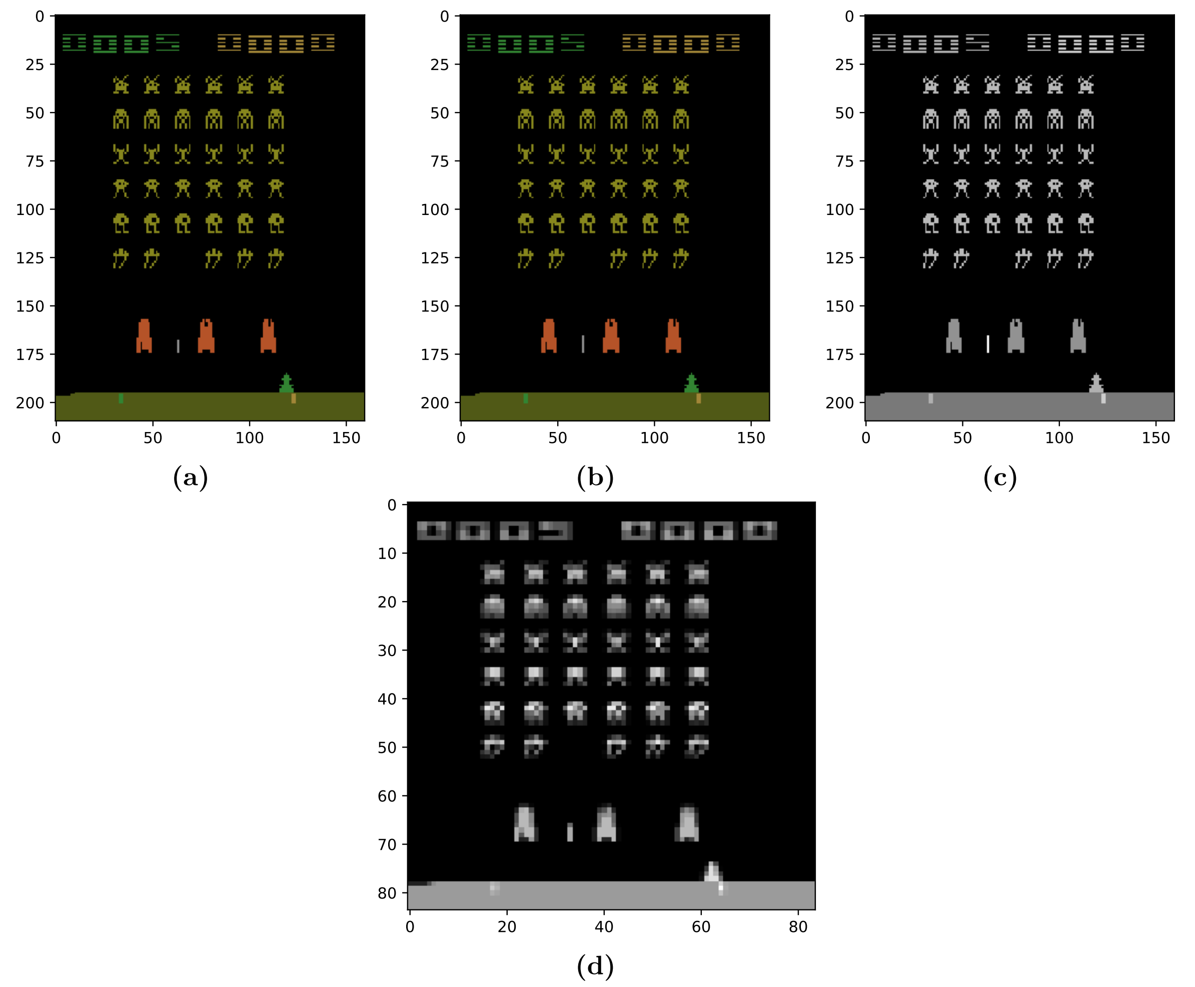


Figure 4.2: (a) Original frame from Space Invaders. **(b)** Flicker removed by setting pixels to their maximal value over previous four frames. Notice the elongated beam. **(c)** Converted to greyscale. **(d)** Resized to 84 × 84.

In more detail, the first layer is a convolution layer with 10 5×5 kernels and 10 biases and is applied to the single greyscale channel of the input images. This layer is followed by a batch normalization layer with a mean and standard deviation for each of the 10 kernels and a learned affine transformation. A max pooling layer with 2×2 kernel and no padding is then applied. Finally, a ReLU nonlinearity is applied. The second convolution layer has 20 kernels but is otherwise identical to the first also followed by batch normalization, max pooling and ReLU nonlinearity. The following fully connected layer has the 320 outputs of the preceding convolution layer as input and has 50 outputs followed by batch normalization and ReLU nonlinearity. The output layer of the model is fully connected with 50 inputs and 10 outputs, one for each digit. A log-softmax nonlinearity is applied to the output units and combined with the NLL loss. Listing B.1 contains a summary of this network.

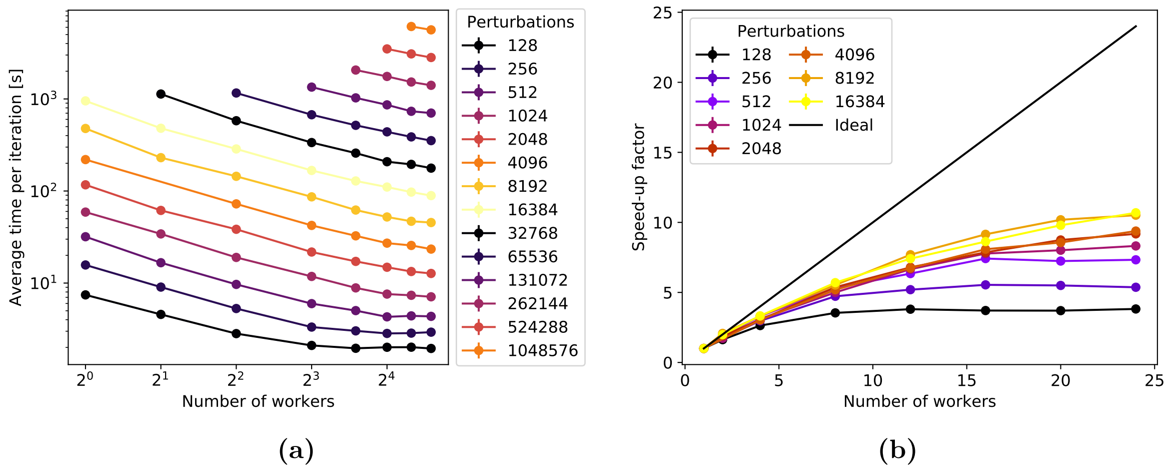


Figure 4.3: Computational scaling in the supervised setting for different numbers of CPUs and perturbations. Each observation is the average over 100 iterations with associated standard deviation (too small to discern). **(a)** The average time spent per iteration in seconds. **(b)** The observed speedup.

4.3.2 Architecture for Atari environments

The network architecture used for learning policies for the Atari environments is the same as the one used by [68, 103]. In short, the network is a CNN with three convolutional layers followed by two fully connected layers totalling 1.685.667 parameters, most of them in the first linear layer.

In more detail, the first convolutional layer has 32 8×8 kernels, 32 biases and is applied to each of the four most recent frames, each of which is treated as a channel. This layer has a stride of 4 and is followed by a ReLU nonlinearity. The second convolutional layer has 64 4×4 kernels, 64 biases and is applied to each of the 32 20×20 outputs of the previous layer. It has a stride of 2 and is also followed by a ReLU nonlinearity. The third and final convolutional has 64 3×3 kernels, 64 biases and is applied to each of the 64 9×9 outputs of the previous layer. It has a stride of 1 and is followed by a ReLU nonlinearity. A fully connected layer with $64 \times 7 \times 7 = 3136$ inputs, 512 outputs and a ReLU nonlinearity connects the flattened output of the final convolutional layer with the output layer. The output layer is fully connected and followed by a log-softmax nonlinearity with its number of outputs defined by the specific Atari environment. For the Freeway environment it is 3 while for Seaquest it is 18. Listing B.4 contains a summary of this network.

4.4 Computational scaling

4.4.1 Scaling in supervised learning

This section presents the computational scaling of the VO method used to train the MNIST architecture described above when using a several CPUs and varying the number of perturbations. Each setting of number of perturbations was run for 100 iterations and evaluated on a mini-batch of 200 images. The mean time and variance were computed from the 100 iterations run for each combination of number of CPUs and perturbations.

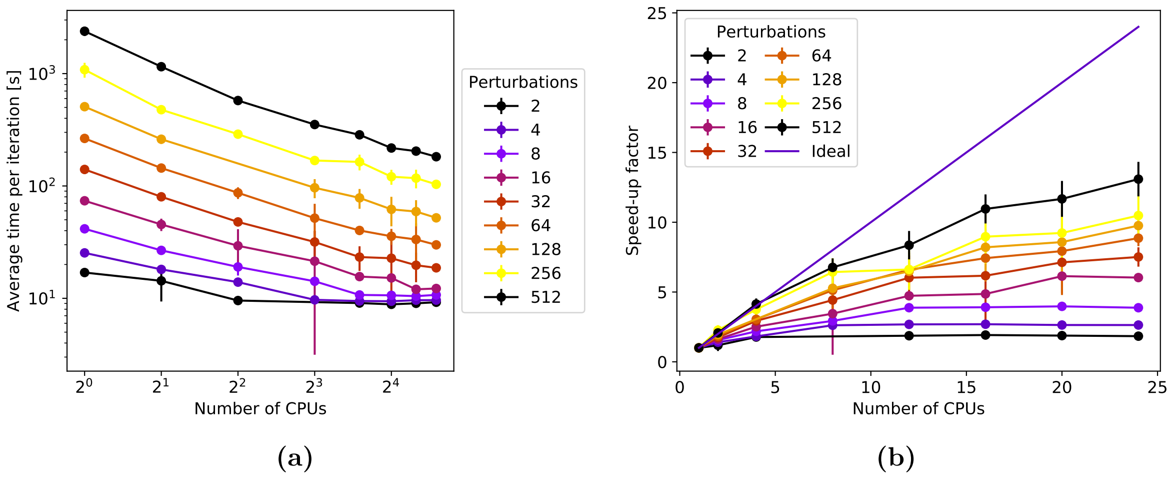


Figure 4.4: Computational scaling in the reinforcement learning setting for different numbers of CPUs and perturbations. Each observation is the average over 100 iterations with associated standard deviation. **(a)** The average time spent per iteration in seconds. **(b)** The observed speedup.

Figure 4.3a shows the average time spent per iteration as a function of the number of CPUs used for various numbers of perturbations. The scaling can be seen to be fairly good for a high enough number of perturbations. Since the evaluation of every perturbation is a single forward pass through the network, the fitness evaluation is quite fast and slower function evaluations would give better scaling. Figure 4.3b shows the measured speedup as a function of the number CPUs for different perturbations. It is evident that speedup quickly drops off from the ideal although an order of magnitude speedup is observed for 24 CPUs.

Obviously, VO is an inefficient choice for optimizing differentiable NNs in the supervised setting. Backpropagation excels at this task while also allowing for efficient batch parallelization on graphics processing units (GPUs).

4.4.2 Scaling on a reinforcement learning problem

This section presents the results of running the same scaling experiment as before but here on the Atari-2600 game Freeway. The used network is the DQN from [68] and preprocessing is as described above. Each episode was limited to 1000 frames (episode horizon) and 100 episodes were simulated for each combination of number of CPUs and perturbations. Freeway was chosen since it has a fixed episode duration allowing the use of untrained policies in the experiment. This removes the dependency of the scaling on the quality of the policy.

The results are presented in the same form as for the supervised case in Figure 4.4. The scaling is somewhat better than for the supervised case and it is achieved for much lower numbers of perturbations as a result of the much more expensive fitness evaluation. For 512 perturbations, increasing the number of CPUs from 1 to 4 provides an almost monomial decrease in the time spent per iteration. This is as observed in [90] although the computational resources expended for experimentation were much greater than here. Conclusively, reinforcement learning is much more well suited for the VO method supervised learning.

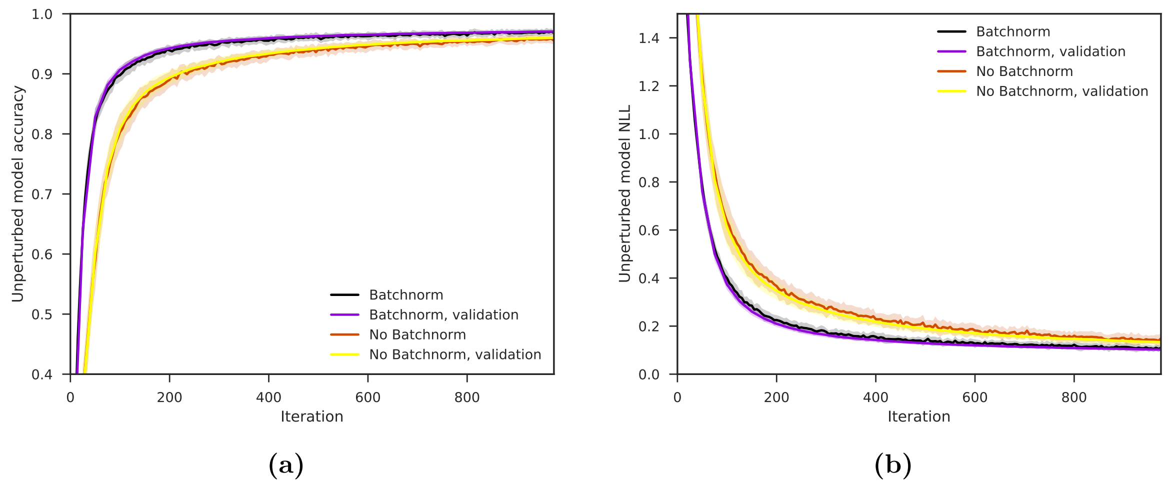


Figure 4.5: Results of experiments with batch normalization for the unperturbed model. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss. Batch normalization is effective at improving the optimization of the NN.

4.5 Effect of model augmentations and safe mutation

This section explores the effects of some common augmentation techniques for NN models along with the effects of safe mutation in the VO algorithm. The experiments investigate whether the VO gradient benefits from the same model improvements as algorithms based on conventional backpropagated gradients do.

The algorithm is isotropic Gaussian VO a with fixed variance of 0.05 similar to [90] but uses safe mutation. The experiments were run with 100 perturbations using SGD with a momentum of 0.9 and L^2 norm regularization of 0.001 on all optimized parameters. A learning rate of 0.05 was used. If nothing else is noted, antithetic sampling and the signed gradient (sum) version of safe mutation were used, batch normalization layers were included in the model and the method of common random numbers was not applied.

The setting is supervised and the data set is MNIST for all experiments. The network used is the one for MNIST (Listing B.1) along with a variant without batch normalization (Listing B.2) and a variant without batch normalization and with dropout (Listing B.3). Each experiment is run for 1000 iterations on mini-batches of 1000 examples for 30 different random seeds. Each experiment shows two plots. The **(a)** plot shows the classification accuracy for the unperturbed model with each series averaged over the 30 runs at every iteration. The **(b)** plot shows the NLL loss. Both metrics are evaluated on the training set and the test (validation[3]) set during training. The shaded bands indicate one standard deviation among the runs.

Some of these experiments were also run using the Adam optimizer. These plots can be seen in Appendix C. The only difference is the choice of optimizer. The hyperparameter settings for Adam are as recommended in the original paper except for the learning rate which is as for SGD. It generally proved to be more difficult to obtain good training characteristics using the Adam optimizer which is hypothesized to be due to the relatively high gradient variance resulting in poor second order moment estimates.

[3]The validation set is in fact the remaining part of the data set leaving no data for testing. However, the validation set is not used for fitting any hyperparameters an can as such be regarded a test set as well.

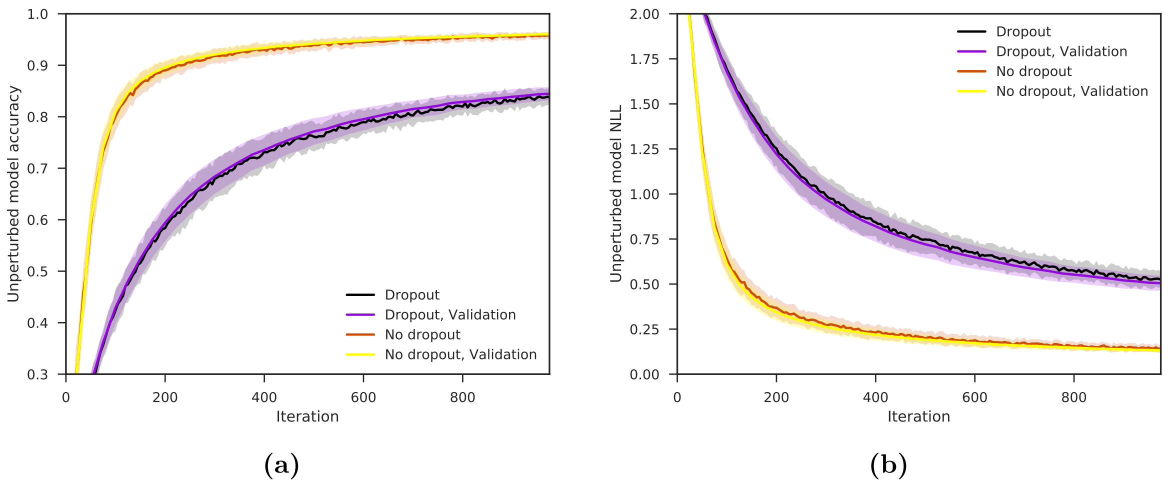


Figure 4.6: Results of experiments with dropout for the unperturbed model. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss. Dropout drastically reduces ease of training probably due to a small network with relatively low capacity and the regularizing effect of optimizing the smoothing VO objective.

4.5.1 Batch normalization

Here, versions of the MNIST network with and without batch normalization layers were compared. The positive effect of batch normalization layers between convolution and fully connected layers in the network is evident from Figure 4.5. The network with batch normalization has a significantly steeper loss curve, reaching low losses and high accuracies faster than the network without batch normalization. It seems the networks asymptote the same final accuracy and loss suggesting that the network without batch normalization is in this case capable of reaching the same level of accuracy as the one with batch normalization, given enough time. This is in line with the expectated effect of batch normalization as presented in Chapter 2. As such, batch normalization can be said to work as expected also with the VO gradient which is estimated without the use of backpropagation.

4.5.2 Dropout

This section examines the effect of applying dropout to the MNIST network. The compared networks are those in Listing B.2 and Listing B.3, i.e. the network with dropout is compared to a network without dropout. Neither of the networks use batch normalization.

The results of the experiment are shown in Figure 4.6. The use of dropout results in slower learning and attainment of significantly inferior loss and classification accuracy withing the 1000 iterations. This effect of using dropout is expected to some degree since for the relatively small network used, randomly zeroing weights can drastically reduce model capacity. Additionally, the MNIST network without dropout showed no signs of overfitting in other experiments using the stochastic gradient estimate. As such, dropout was not expected to be able to radically improve performance.

The fact that no overfitting occurs for this model may be linked to the smoothing of the objective function imposed by using the variational objective. This is most likely a different

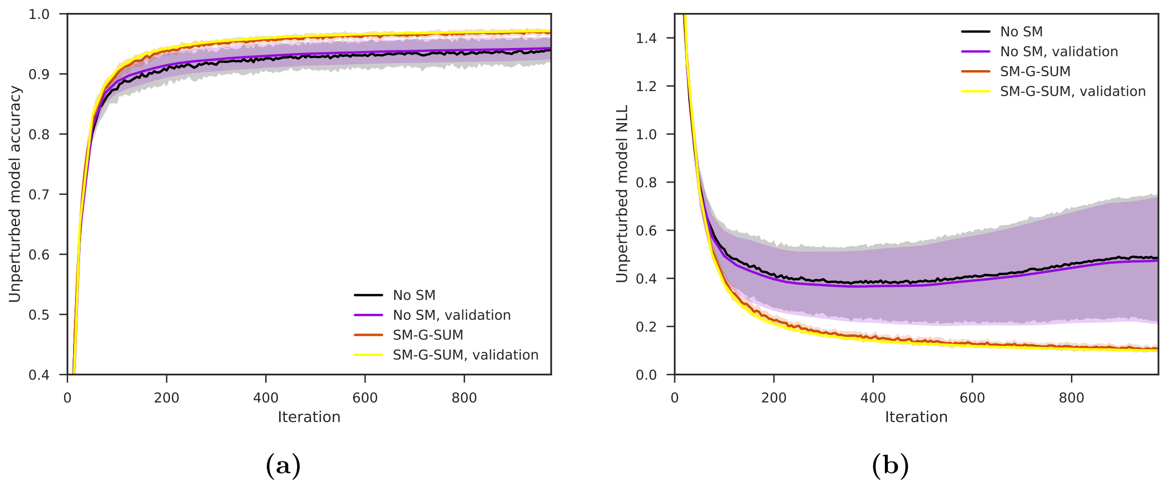


Figure 4.7: Results of experiments with safe mutation (SM-G-SUM) for the unperturbed model. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss. Scaling perturbations according to network parameter sensitivities results in more stable training.

perspective on the fact that VO with a fixed variance optimizes for the average loss of the entire population of perturbations [53]. It seems probable that this assists in avoiding overfitting. This observation is also in line with the examples of Figure 3.7, 3.8 and 3.9 where the algorithm with fixed σ cannot fully converge on the minimum, and in some sense, thus cannot not overfit it.

4.5.3 Safe mutation

This section compares runs with and without the signed gradient (sum) version of safe mutation [54].

The results are shown in Figure 4.7. Considering the loss curves, not using safe mutation results in much higher variance among runs with different seeds and yields an over run average loss that is much higher compared to runs using safe mutation. Considering the classification accuracies, the difference between runs with and without safe mutation is smaller although the same trends are observed: Over run variance is higher without safe mutation than it is with and runs without safe mutation reach lower classification accuracies on average. This, along with the high loss, indicates that the variants without safe mutation end up in relatively poor areas of the loss surfaces.

It should be noted that it cannot be ruled out that a lower learning rate exists that allows better performance without safe mutation than what is observed here. For instance, the results presented in [90] did not use safe mutation. In this case, the use of safe mutation effectively allows using a higher learning rate which must be expected to result in faster convergence.

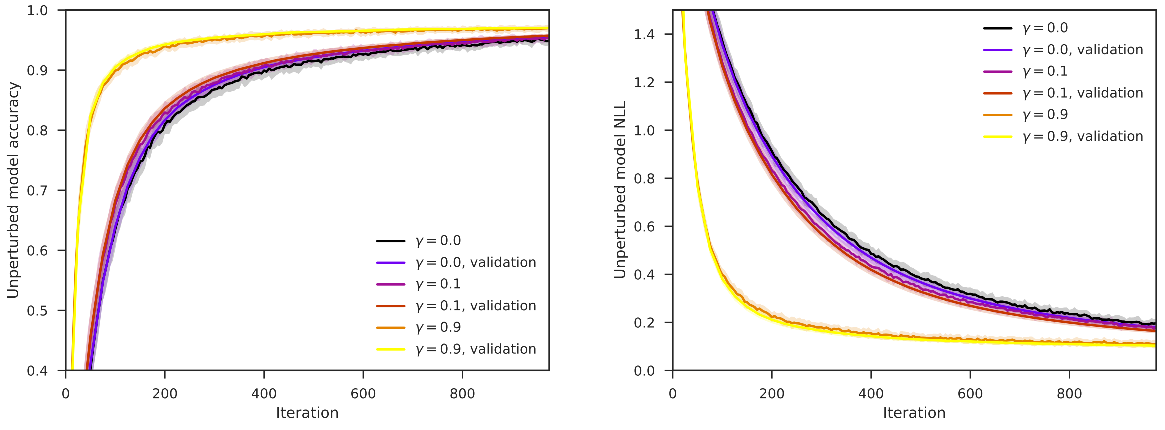


Figure 4.8: Results of experiments using SGD with momentum on the VO gradient for the unperturbed model run on MNIST. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss. Gradient momentum is effective at reducing the gradient variance and results in faster convergence.

4.6 Methods for variance reduction

This section explores the effects of different variance reduction methods. If nothing else is noted, the training details are as described in Section 4.5.

4.6.1 Gradient momentum

Due to the potentially high variance of the VO gradient, the use of momentum (see Section 2.3.4.1) can have a positive effect on the convergence speed and training time of VO. Three levels of momentum were run for thirty random seeds each. The momentums were 0, 0.1 and 0.9. The results can be seen in Figure 4.8. A small benefit can be noted from using a momentum of 0.1 while using 0.9 has a large impact on the convergence speed. It is evident that the VO gradient benefits from including momentum.

A momentum of 0.99 was also included in this experiment but resulted in the algorithm not converging. Instead, the training achieved losses of about 0.6, oscillating up and down presumably due to the gradient being too dependent on past values to adequately adapt to the loss surface. It is plausible that an optimal value of the momentum for this problem resides somewhere between 0.99 and 0.1 but no attempt will be made at finding this value.

When adapting the variance, the effect of momentum on the variance was also examined and found to occasionally be detrimental, especially in the RL setting. At times, the gradient associated with the variance would spike at a single iteration, preceded and followed by much smaller gradients. Momentum would then continue to increase the variance for many iterations following the spike resulting in a large variance giving detrimentally large perturbations. Lowering the variance learning rate results in the variance practically not adapting during training. Using dampened momentum aids in suppressing these spikes and is discussed further in Section 4.7. Alternatively, not using momentum on the variance gradient gives little weight to these spikes and the trend of decreasing variance persists through training as wanted.

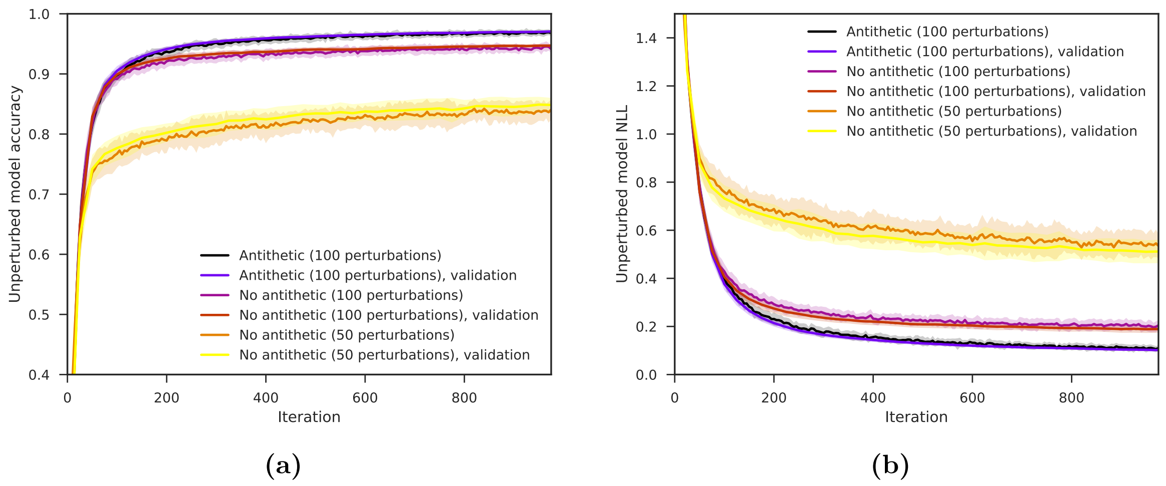


Figure 4.9: Results of experiments with antithetic sampling for the unperturbed model. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss.

4.6.2 Antithetic sampling

In this section, the improvement by using antithetic sampling is experimentally validated. The MNIST network was trained using 100 perturbations with antithetic sampling (50 unique, 50 mirrored), 100 perturbations without antithetic sampling and 50 perturbations without antithetic sampling.

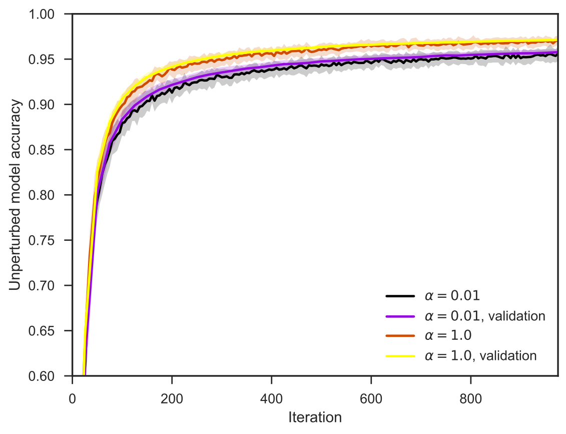
The runs using 100 perturbations with antithetic sampling compares to the runs using 100 perturbations without antithetic sampling in the way that both estimate the gradient from 100 perturbations. From this comparison it can be assessed whether the antithetic samples carry more information than the regular uninformed samples.

The grounds for comparison of runs using 50 perturbations without and 100 perturbations with antithetic sampling is that both of these versions perturb an 50-dimensional subspace of the network parameter space at every iteration of variational optimization (VO). Additionally, the computational complexity of sampling is half as high when using antithetic sampling. Obviously, the time complexity of the sampling operation is negligible compared to the fitness evaluation so comparison in this basis is not entirely fair.

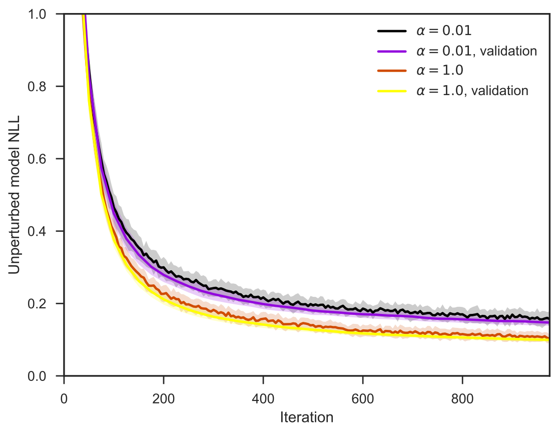
The results are shown in Figure 4.9. It is evident that using antithetic sampling results in a significant improvement when comparing to both runs with 50 and 100 perturbations. As opposed to the effect seen when including batch normalization, the loss curves initially rise with approximately the same slope but do not asymptote the same value. Thus, the benefit from using antithetic sampling is not faster training but rather the location of a better local minimum and thus better final performance. This is almost certainly due to a reduction in the gradient estimate variance.

4.6.3 Importance mixing

This section presents results of using importance mixing. This comparison is made in spite of the importance weight collapse and the numerical problems associated with it. In practice some weights become infinite which leads to some reuse of some perturbations.

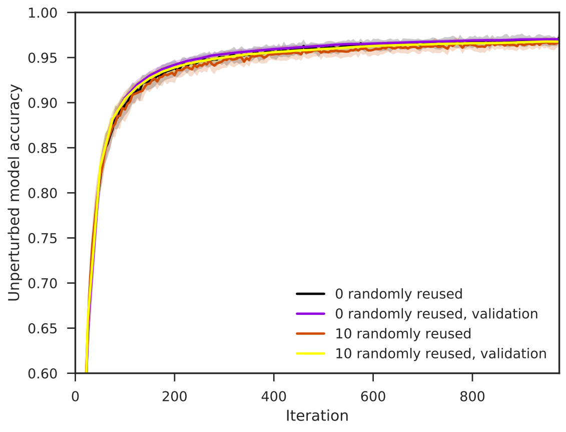


(a)

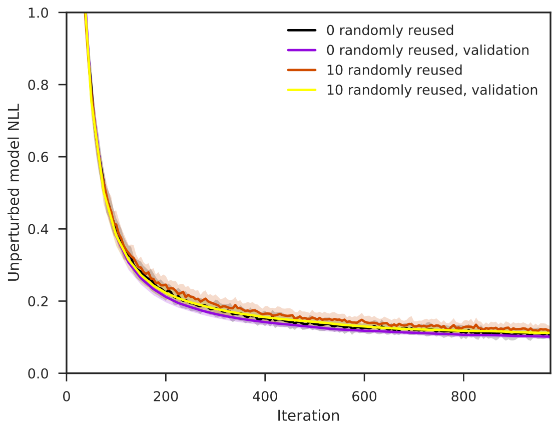


(b)

Figure 4.10: Results of experiments with importance mixing for the unperturbed model. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss.



(a)



(b)

Figure 4.11: Results of experiments with random resampling for the unperturbed model. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss.

Figure 4.10 compares a run with importance mixing ($\alpha = 0.01$) and a run without ($\alpha = 1.0$). The NLL when using importance mixing is slightly higher than when not. This observation is present in the resulting classification accuracy as well. This detrimental effect must be concluded to stem from the collapse of the importance weights as previously discussed.

For comparison, Figure 4.11 presents results runs where importance mixing has been replaced by random reuse of perturbations. In the runs with importance mixing, about 10 of the 100 perturbations where effectively reused at each iteration. Therefore, the experiment with random reuse reused 10 perturbations randomly at each iteration. It can be observed that the effect of randomly reusing perturbations is smaller than that of importance mixing, although it remains detrimental compared to reusing no perturbations.

One can note that randomly reusing perturbations is likely to break to assumption that

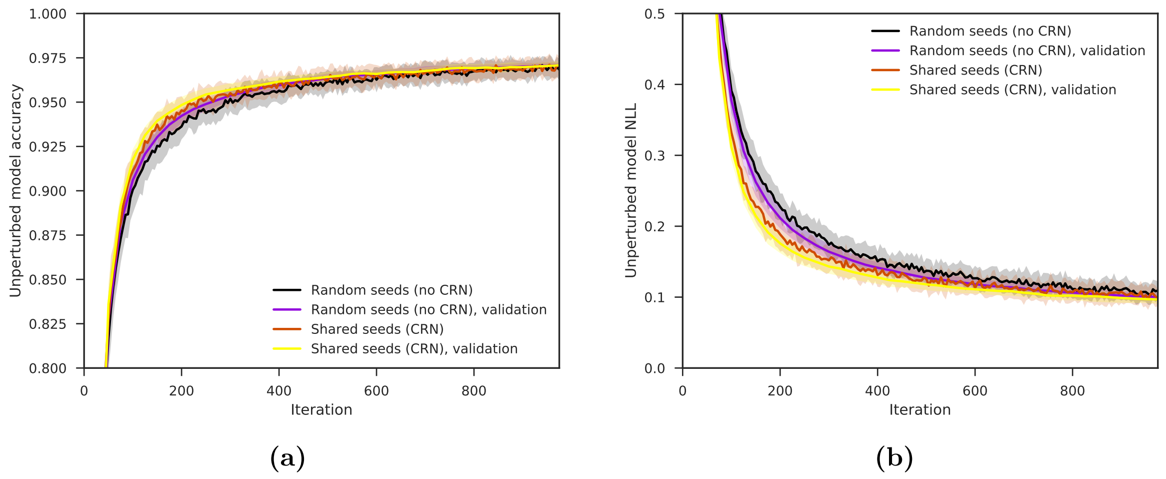


Figure 4.12: Results of experiments with common random numbers (CRN) for the unperturbed model run on MNIST. **(a)** Training and validation set classification accuracy. **(b)** Training and validation set NLL loss.

the set of perturbations follows the search distribution, i.e. the Gaussian in this case. Since the gradient estimators rely on this assumption, the detrimental effect of random reuse of perturbations is expected. Importance mixing is designed to maintain the distribution of the perturbations but due to the collapse of the importance weights this is not the case in practice.

4.6.4 Common random numbers

This section presents results using the method of CRN. The MNIST network was trained using 100 perturbations with and without CRN resulting in the training curves illustrated in Figure 4.12. CRN was also tested in the RL setting with resulting learning curves seen in Figure 4.13. The RL runs were made using 40 perturbations for computational feasibility.

There is a small and almost insignificant benefit to using CRN in the case of supervised learning on MNIST. It can be seen that using CRN, the training and validation accuracy and loss slightly outperform those of runs that did not use CRN on average but within one standard deviation.

The method was also tested in the RL setting with 45 episodes simulated for each of the shared and random seeds versions. The results are seen for Freeway in Figure 4.13a and Seaquest in Figure 4.13b. The average population reward is plotted versus iteration number. The picture is similar to that of supervised learning with no significant improvement to be seen by using shared seeds for the simulations. As such, the method of CRN cannot be concluded to improve on the VO gradient estimate.

4.7 Effect of adapting the variance

Until now, all experiments have been run with an isotropic Gaussian with fixed variance similarly to [90]. Here, different variants of the Gaussian search distribution with adaptive variance will be examined. The examined variants are the isotropic Gaussian, the layer-wise separable Gaussian and the parameter-wise separable Gaussian. For the layer-wise separable

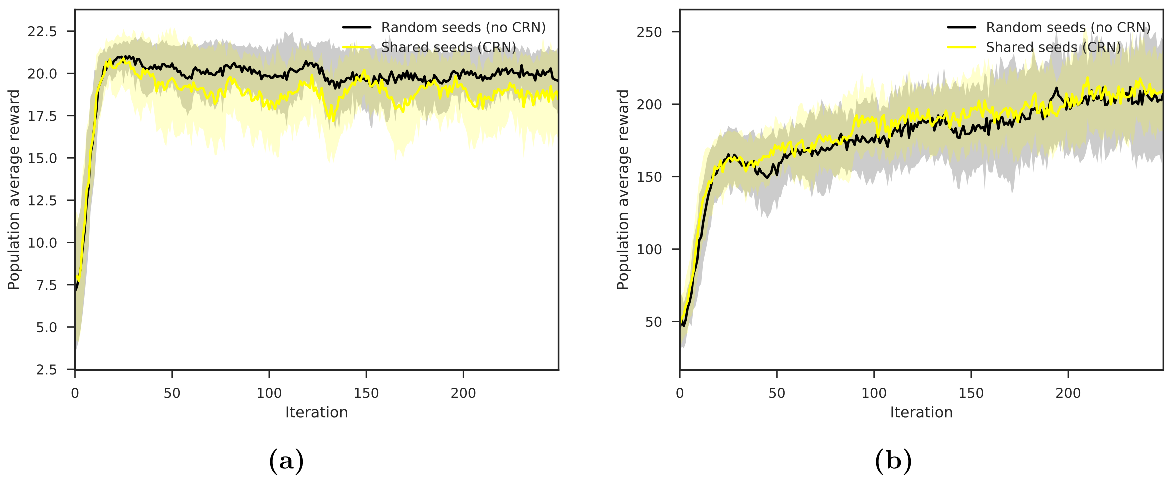


Figure 4.13: Results of experiments with common random numbers (CRN) for the population average on **(a)** Freeway and **(b)** Seaquest. The average reward of the population is plotted as a function of VO iterations.

Gaussian, a single variance parameter is used for each weight/kernel matrix and bias vector in the model. Hyperparameters are otherwise as in Section 4.5.

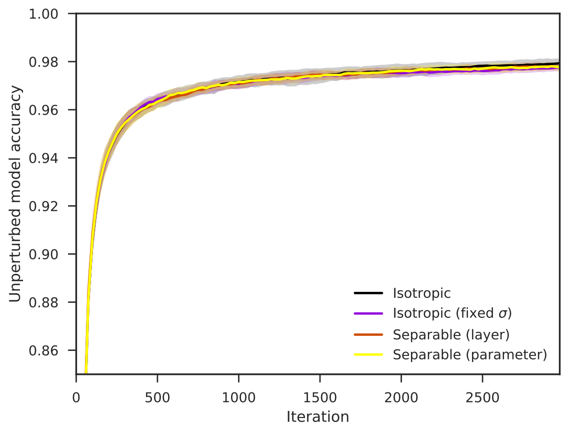
For these experiments, a momentum of 0.9 has been used on the parameter gradient as well as the adapted variance gradient while the variance gradient is also dampened by 0.9. Dampening the variance gradient makes sure it does not spike as discussed in Section 4.6.1. Dampened momentum on the variance gradient results in much smoother updates to the variance than applying no momentum or non-dampened momentum. Learning rates of 2 and 4 have been used for the isotropic and separable Gaussian variances, respectively.

4.7.1 Adapting the variance

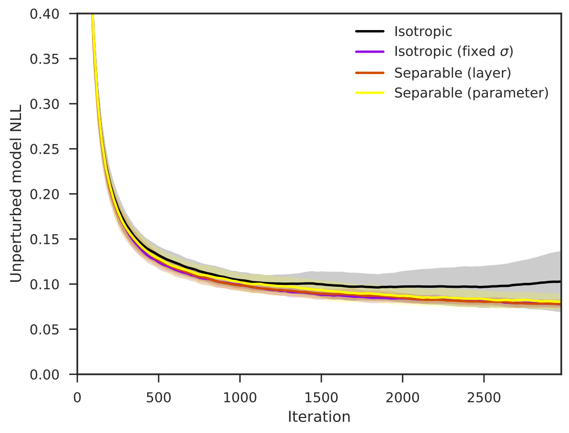
Figure 4.14 shows the results of running the VO algorithm on the MNIST network with the different search distributions. Figure 4.14a and 4.14b respectively show the validation set accuracy and NLL loss for the different search distributions. The most notable difference between the variations is that when the variance is adapted, the isotropic Gaussian gives somewhat unstable learning, in that it gives quite different results solely dependent on random seed. This can be seen not to be the case when adapting a layer- or parameter-wise separable Gaussians.

Despite this, the convergence rate and the median final NLL loss and accuracy are not improved when using these adaptive search distributions compared to using the fixed isotropic Gaussian. Considering the single best and worst performing models on the validation set, it can be noted that these are all obtained by the separable versions. As such, adapting the variance seems to encourage more extensive exploration of the loss landscape but with risk of landing in both a slightly better and slightly worse minimum than would have been reached with a fixed variance.

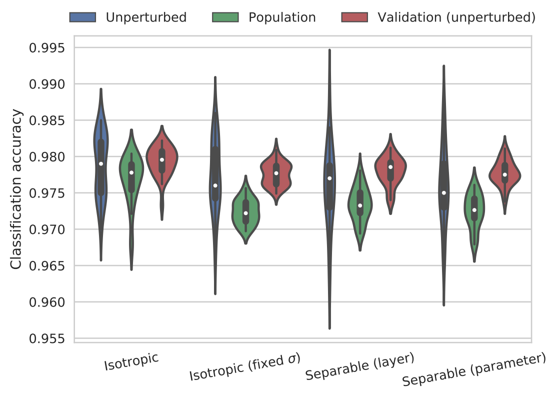
It should be noted that these observations hold for many different learning rates for the variance parameter(s).



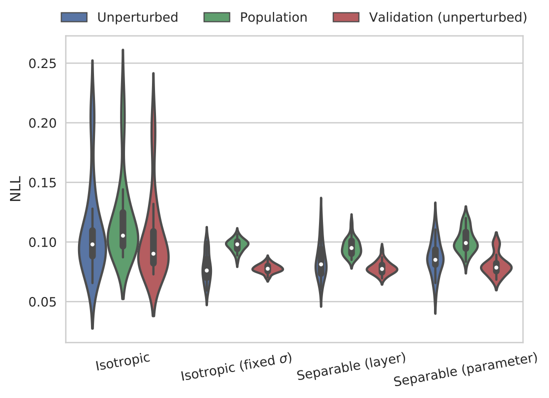
(a)



(b)



(c)



(d)

Figure 4.14: Results of experiments with different VO algorithms. The model variations are the fixed variance strategy of [90] and VO with adjusted variance using an isotropic Gaussian and respectively a layer-wise and per-weight separable Gaussian. Contrary to [90], safe mutation is applied. Plotted versus the iteration number, **(a)** shows the validation set classification accuracy and **(b)** the validation set NLL loss. Difference in performance between the algorithms is very small and within one standard deviation with the adapted isotropic having high between-run variance. In **(c)** and **(d)** the final distribution of the classification accuracy and NLL loss from **(a)** and **(b)** are shown. It can be noted that adapting the variance in the isotropic Gaussian search distribution results a much wider span of the final NLL loss. Additionally, a small group of runs has an even higher final NLL loss than the main body of runs. This is not observed for the other versions.

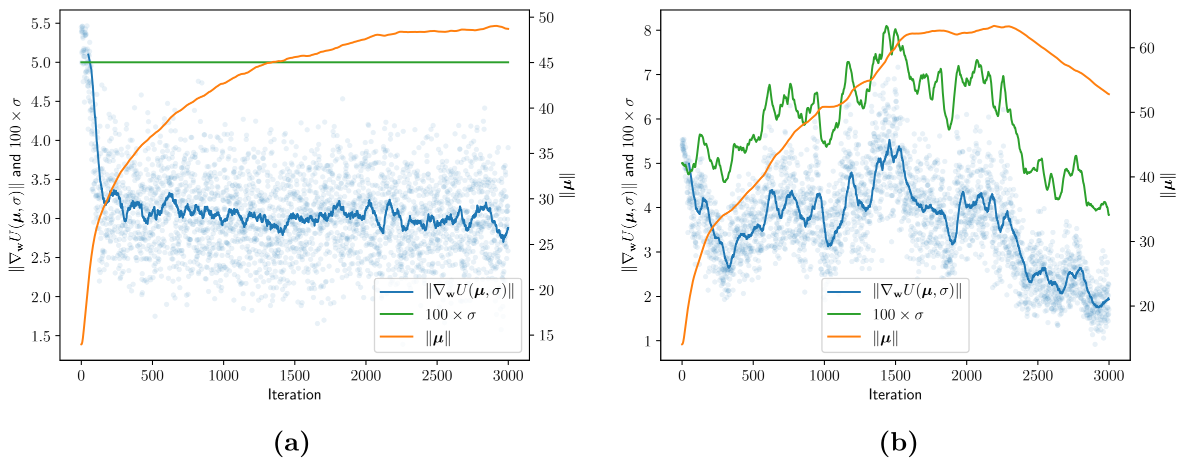


Figure 4.15: 2-norms of the NN parameter vector and VO gradient for an isotropic Gaussian search distribution with the variance overlayed (multiplied by 100 for scale). In **(a)** and **(b)**, the fixed and adapted variance versions are shown, respectively. A centered 50 sample moving average is computed for the gradient. It is clear that adapting the variance directly and significantly influences the norm of the gradient and in turn also the norm of the parameter vector.

4.7.2 The gradient norm

This section examines the interaction between adapting the variance and the norm of the VO gradient and the parameter vector in order to shed more light on the effect of adapting the variance. It does so based on single runs of the training, but it should be noted that the observations hold generally for additional runs with different random seeds.

Consider the gradient and parameter norms when using an isotropic Gaussian search distribution. As can be seen from the derived VO gradient estimators in (3.3.44), the gradient is scaled by the variance. In case the variance is fixed, this is obviously a constant scaling factor. However, when adapting the variance, this effectively scales the term computed by the sum differently at each iteration. It is evident that the variance then has a direct adaptive effect on the gradient norm and thus affects the iteration step size. The effect that the variance has through multiplication on the perturbation and indirectly through the value of the objective function is harder to determine. Since the perturbations are from a standard Gaussian, multiplying them by σ does not change their expectation. Additionally, it is theoretically possible for the objective function to both increase and decrease in value for any size of the perturbation. However, very large perturbations must be expected to result in catastrophic forgetting in the perturbed networks as any learned filters etc. are washed out with noise.

The gradient and parameter norms for fixed and adapted variance isotropic Gaussians are plotted in Figure 4.15 for a single run of the MNIST network (Listing B.1). The variance is overlaid in the plots as well. A fixed variance (4.15a) results in a gradient that rapidly reaches a constant level. The parameter norm increases fairly steadily as a result, but seems to plateau when a minimum is found[4]. Interestingly enough, the gradient does not go to zero as the minimum is reached but remains relatively high. This is in-line with the rarity

[4]The learning curves are similar to the ones in Figure 4.14

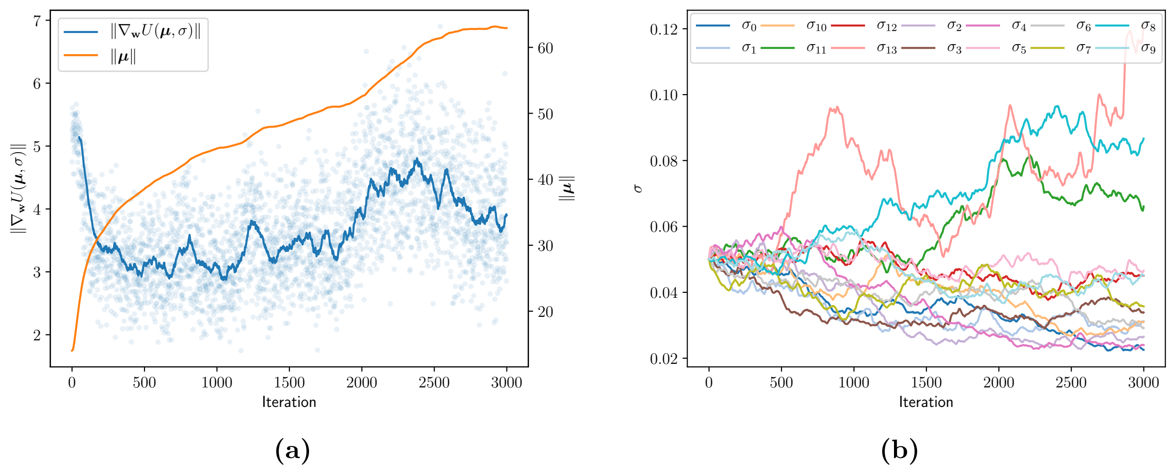


Figure 4.16: **(a)** 2-norms of the NN parameter vector and VO gradient for a layer-wise separable Gaussian search distribution with the variances plotted separately in **(b)**. Most variances tend to zero while a few increase. The gradient norm feels the combined effect but generally increases.

#	Layer	weight/kernel	bias
1	Convolutional	0	1
2	Batch normalization	2	3
3	Convolutional	4	5
4	Batch normalization	6	7
5	Fully connected	8	9
6	Batch normalization	10	11
7	Fully connected	12	13

Table 4.17: Labels of the variances of the layer-wise separable Gaussian in Figure 4.16 used on the MNIST model in Listing B.1. The labels are divided into those for the weight/kernel and those for the bias.

of local minima and the prevalence of saddle points as previously mentioned. It should be noted that the small amount of L^2 regularization adds to decrease the parameter norm. By adapting the variance (4.15b), the gradient is also adapted: As the variance decreases so does the gradient norm and in turn the parameter norm as well. The effect of the variance on the gradient norm is similar to that of adaptively decreasing the learning rate during training in the way that increasingly smaller steps are taken as training progresses. By inspecting the results presented in Figure 4.14, it was found that the small poorly performing group of runs that used the adapted isotropic Gaussian had a variance that generally increased rather than decreased throughout training. This then directly resulted in larger gradients and in turn divergence.

For the layer-wise separable Gaussians, Figure 4.16 shows the results. Here, the variances are plotted separately for clarity. The variances are labelled incrementally by their layer in the model as described by Table 4.17. This search distribution often sees many of the variances go

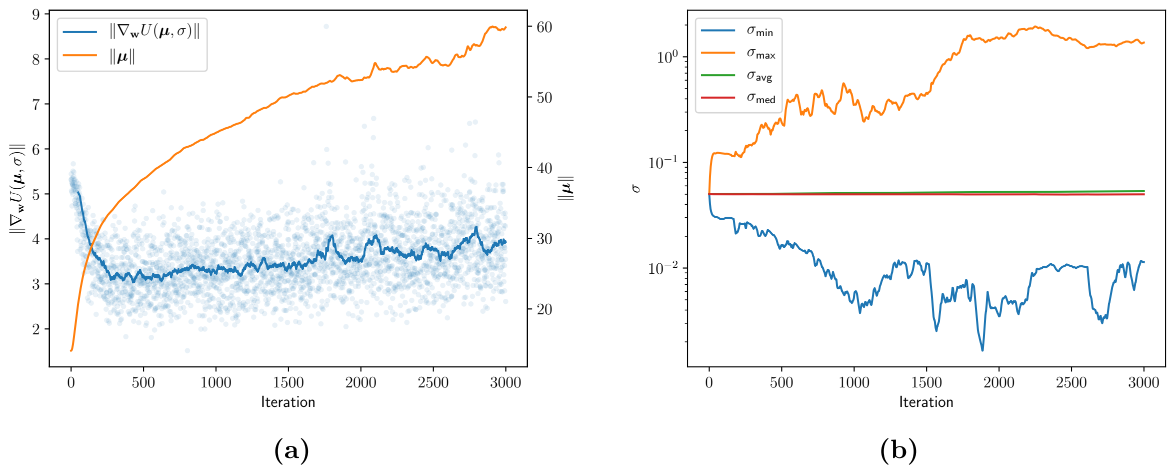


Figure 4.18: (a) 2-norms of the NN parameter vector and VO gradient for a parameter-wise separable Gaussian search distribution with the maximal, minimal, average and median variances plotted separately in **(b)**.

toward zero while a few increase. As for the isotropic case the gradient norm is influenced, but now depends on all the variances. As such it is slightly more smooth but also does not trend as clearly to zero. One can note that there is some tendency for the variances of earlier layers to decrease more than for later layers. Specifically, the three variances that increase the most are for the 7th, 6th and 5th layer. This can be speculated to be due to the features of earlier layers being learned sooner during training than features of later layers. This might then result in flatter regions of the loss surface in the respective dimensions and thus decreasing variances.

The variances of the parameter-wise separable Gaussian behave similar to the variances of the layer-wise but in a more extreme way and are shown in Figure 4.18. The maximal and minimal variances change by two orders of magnitude while the median and average variances remain close to the initial value of 0.5. The effect on the gradient norm is smaller but there is a tendency for it to increase compared to the fixed variance isotropic Gaussian in Figure 4.15a. That the parameter-wise variances tend to both increase and decrease by potentially large amounts while the average and median remain approximately at the initial value may indicate that the behaviour is similar to a random walk. This behaviour may be due to the fact that many more variance gradients need to be estimated at each iteration while using the same number of perturbations. Additionally, the gradient of the variance of an isotropic Gaussian is estimated by computing the sum of squares of all elements of each perturbation vector. This holds as well for each variance of a layer-wise separable Gaussian considering only the elements used for each layer. In the parameter-wise case however, a single contribution is made from each element of a perturbation vector. As such, the reason for the random walk behaviour of the variances may be attributable to a poor estimate of their gradients due to a number of effective samples that is too low.

4.7.3 Discussion

The trends in the values of the adapted variances of the separable Gaussians seem to indicate that different layers/parameters have different optimal variances for their perturbations at different times during training. This might help explain why adapting the variance for an isotropic Gaussian sometimes results in divergence while this does not happen for separable Gaussians. When using an isotropic Gaussian, the variance may not be able to appropriately adapt to a wide range of optimal perturbation sizes. Some parameters requiring low variances may then be perturbed with relatively high variance noise and risk losing learned patterns. It should be noted that this is speculative since no experiments have been performed to validate these hypotheses.

The difficulty of drawing benefit from adapting the variance of the search distribution may be partially attributable to the complex geometry of the high dimensional loss surfaces of NNs. As previously mentioned, saddle points are exponentially more prevalent than local minima in NN loss surfaces while a monotonically decreasing minimization path exists on the loss surface. The VO adapted variance attempts to zero in on a local minimum. It does so by decreasing the variance in regions of high convex curvature and increasing it in regions of low concave curvature. While the variance is expected to go to zero at a minimum, this is not necessarily the case at a saddle point since only some dimensions exhibit convex curvature. This is especially obvious in the case of an isotropic Gaussian since this distribution is inherently unable to parameterize differences in the variance between dimensions. The separable Gaussians allow more such flexibility. This may be a dynamic that adds to the results observed. Additionally, the monotonically decreasing nature of some paths along the loss surface may encourage a somewhat constant level of the variance, at least in the isotropic case.

CHAPTER 5

Future work

"I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."

- Turing, Alan (1950) [115]

5.1 Variations on regular Monte Carlo

This chapter describes potential avenues of future work. An obvious area of interest is variations on the regular Monte Carlo estimation procedure. This thesis has for instance considered antithetic sampling and common random numbers but many others exist.

Quasi-Monte Carlo methods are a variation on regular Monte Carlo methods that use low-discrepancy (deterministic) sequences rather than pseudo-random sequences which can speed up convergence in many cases including high dimensions although the benefit is greater the smoother the function and lower the dimension [69]. It is fairly straightforward to implement such a low-discrepancy sequence in place of a random sampling but it is somewhat unclear how gradients should be computed from it given the formalism of search distributions. Randomized quasi-Monte Carlo methods which add a pseudo-random sequence to the low-discrepancy sequence may be a potential solution [52].

As discussed in the thesis, the reuse of information gathered from previous samples seems to be promising as indicated e.g. by the effectiveness of gradient momentum. In Monte Carlo methods, stratified sampling reduces variance by sampling in so-called strata which are predefined regions of the search space. In high-dimensional problems, this approach becomes infeasible due to the exponentially large number of required strata. Adaptive or recurrent stratified sampling [11] attempts to cope with this by allocating strata adaptively during optimization according to the areas of the objective that exhibit the most variation. A major problem with these stratified approaches is the requirement for the number of samples to scale with the dimension, which is high in an NN. The adaptive stratified sampling can however be shown to be at least as efficient as regular Monte Carlo in all cases [11].

Although these methods may be able to improve performance they do not exploit the special structure of the problem. Approaches that do this in some way are discussed below.

5.2 Exploiting problem structure

Although a theoretically substantiated argument for the viability of the local reparameterization trick presented in Section 3.6.4 has been made, the practical effectiveness of this approach remains unknown. In future work, it should be implemented and subjected to experimental evaluation.

The utility function presented in Section 3.5.1 is a central element in the VO algorithm but its selection is largely based on heuristics. It serves to remove the dependency on the size of the objective function and computes fixed size utilities that describe the relative fitness of the samples. In doing so it may however discard valuable information on the amount by which some samples were better or worse than others. To move away from the heuristic definition of fitness transformations, a potential approach could be to examine learnable fitness transformations by use of computationally inexpensive ML methods. Another improvement to the utility function could be the subtraction of baselines as done outside of deep learning [122] similarly to how actor-critic methods achieve a variance reduction [3].

Staying with the idea of applying ML to augment the algorithm, previously observed pairs of sampled parameters and associated fitnesses could potentially be utilized to predict new high performing samples. This could be achieved by using regression models or models for sequence modelling, e.g. a Markov model, given the sequential nature of the algorithm.

CHAPTER 6

Conclusion

"We have shown that for the model described, autonomous learning is possible."

- Block, Knight and Rosenblatt (1962) [9]

This thesis has considered variational optimization (VO) as an alternative method for computing the gradients of a potentially nondifferentiable neural network (NN) with potentially nondifferentiable loss by adapting a search distribution during training and perturbing in the parameter space.

First, machine learning (ML) was introduced leading to the description of unsupervised and supervised learning along with reinforcement learning (RL). Then, deep learning was introduced along with the different basic NN versions and it was shown how to train them using backpropagation. Finally, a small NN toolbox was implemented in Python.

The main part of the thesis considered VO. Initially, the gradient estimator of [90] was derived using a Taylor expansion. Its bias and variance were evaluated and an interpretation of the estimated gradient as sample covariances was presented. The VO upper bound was then formally derived and its properties were discussed. The similarity of VO to policy gradients within DRL was discussed. A discussion on VO in relation to search space dimensionality concluded that VO effectively selects a low-dimensional subspace of the parameter space of the trained NN implicitly and randomly when a parameter perturbation vector is sampled. The estimators for isotropic and separable Gaussian search distributions were derived and used centrally in the thesis. Finally, the special hill-climber version of VO based on a single perturbation sampled from a Cauchy distribution was briefly presented.

The natural gradient was derived as the optimal search direction for VO by constraining the Kullback-Leibler (KL) divergence between the previous and updated search distributions to be constant at each iteration. The natural gradient was then shown to be superior to the regular gradient by a simple example. Sensitivity rescaled perturbations were also presented and applied to VO enabling higher learning rates and more stable learning. Antithetic sampling for variance reduction was derived for the VO gradient Monte Carlo estimator and shown experimentally to improve learning significantly. Common random numbers was shown not to improve learning. A novel method for variance reduction based on the so-called local reparameterization trick and applicable to cases where the objective function is composed of a sum of individual terms was derived and presented. It exploits the structure of a feedforward neural network (FNN) to also reduce the amount of computation required for a forward pass by perturbing in the activation space and propagating a distribution over activations.

Compared to the estimator presented in [90], the VO framework allows for adapting the entire search distribution including the variance of a Gaussian. The effect of adapting the variance of an isotropic Gaussian and per-layer and per-parameter separable Gaussians was evaluated and found to be insignificant. Possible reasons for this were discussed in relation to the geometry of the loss surface including the rarity of local minima, prevalence of saddle points and monotonically decreasing paths which may make adapting the variance difficult.

APPENDIX A

Implementations

This appendix provides references to the programming code developed in this thesis and gives brief descriptions of the code.

A.1 Variational optimization

The implementations made of VO for this thesis can be accessed at

`https://github.com/JakobHavtorn/es-rl`.

This code is divided into different submodules (folders)

- `data-analysis`: Several scripts for analyzing the experimental data of different experiments labelled by the template `EXXX-ABC` where `XXX` denotes the experiment number and `ABC` denotes some descriptive label or shorthand. All plots in the experimental section were created by these scripts and the associated data.
- `es`: The main folder for algorithmic development. Has a set of python code files containing different parts of the code.
  - `algorithms.py`: The main algorithmic development file. Contains each algorithm implemented as a class.
    - The abstract `Algorithm` base class is the parent class for all algorithms.
    - The `StochasticGradientEstimation` class holds general methods for the algorithms that rely on stochastic gradient estimation, i.e. VO.
    - The `ES` class is the algorithm used in [90] with fixed σ in an isotropic Gaussian search distribution.
    - The `sES` uses a separable Gaussian search distribution with a variance either per parameter or per layer and has the ability to adapt the variance using VO.
    - A single variance can also be chosen corresponding to an isotropic Gaussian search distribution.
    - `sES` also has a natural gradients version in `sNES`.
  - `envs.py`: Contains wrappers for the OpenAI Gym RL environments. The only used wrapper is the `AtariPreProcessorMnih2015` class which performs the preprocessing defined in Section 4.2.4 in its `_observation` method.
  - `eval_funs.py`: Defines objective functions for the supervised, `supervised_eval`, and RL, `gym_rollout`, settings. For the supervised setting, the a batch of examples are forward propagated and the NLL loss is computed for the predictions. The RL setting is evaluated by performing a single rollout of the policy encoded by the model. The file also contains methods for testing and a file for rendering the RL environment for visualizing a learned policy.

  - models.py: Defines a series of NN models using PyTorch. ALl models are subclasses of the AbstractESModel module. The model used for supervised MNIST training is MNISTNet and its variations MNISTNetDropout and MNISTNetNoBN. For RL, the DQN is used for Atari and ClassicalControlFNN for classical control problems such as CartPole.

- experiments: Has the main.py file which is the entry point for all executed experiments. Also holds downloaded data as well as data from executed experiments (not included in repository due to size).
- hpc: Contains scripts for setting up the used virtual environment on the DTU HPC cluster as well as scripts for submitting the experiment jobs.
- msc: Contains a number of miscellaneous files such as examples and small experiments not directly related to VO algorithms.
- tests: Holds a couple of tests written for multiprocessing and sensitivity computation.
- utils: Has several files defining some utlities used for plotting, data analysis, uploading to dropbox etc.

This code is relatively voluminous and is characterized by sequential experimentation and development over a long period of time. No claim is made that this code satisfies all standards for good software development, nor was this the goal of the code.

A.2 Neural network toolbox

The on-the-side neural network package can be accessed at

https://github.com/JakobHavtorn/nn.

This code is also divided into different submodules (folders)

- examples: This folder contains examples of code used to train different NNs on data. The models are defined in models.py, the data from torchvision[1] is loaded using the method from loaders.py. Current examples are mnist_fnn.py and mnist_cnn.py.
- nn: This folder holds the main network modules. The names of the contained files are self-descriptive; for instance, the affine (linear) transformation of the FNN is defined as a class in linear.py. All modules are subclasses of the Module base class which defines common behaviour.
- optim: This folder holds code defining the optimizers that can be used to train the models. All optimizers are subclasses of the Optimizer base class.
- utils: Contains some utilities for the package most importantly the Solver class in solver.py which can train a classifier. utils also includes a progress bar and a method for onehot encoding a target label.

Compared to the VO code, this code is well-structured and in this way it serves as proof that the author is indeed capable of developing well-structured code.

[1]torchvision is part of PyTorch and contains among other things datasets for computer vision and data loader classes.

APPENDIX B

Network models

This appendix lists summaries of the models used for this thesis. The `class_name` refers to the name of the used network module in the PyTorch framework. Table B.1 provides descriptions of all modules used in this thesis for reference. The `input_shape` and `output_shape` columns gives the dimensions of the feature vector input to and output from the layer. Here, the -1 dimension indicates the variable mini-batch size. The learned weight tensors and matrices along with bias vectors have dimensions as specified in the column `weight_shapes`. The total number of parameters and the number of those that are trainable are listed in columns `n_parameters` and `n_trainable`. In the `settings` column, additional settings or specifications for layers are given. For additional information on any of these summaries, refer to the PyTorch documentation at `http://pytorch.org/docs/`.

Table B.1: Interpretation of PyTorch class names

`class_name`	Layer type
Conv2d	Convolutional layer (2-dimensional)
Linear	Fully connected linear layer
MaxPool2d	Max pooling layer (2-dimensional)
BatchNorm2d	Batch normalization layer (2-dimensional)
BatchNorm1d	Batch normalization layer (1-dimensional)
ReLU	Rectified linear unit (elementwise)
LogSoftmax	Logarithmic softmax function (elementwise)

```
   class_name     input_shape      output_shape             weight_shapes  n_parameters  n_trainable                                                              settings
1  Conv2d       (−1, 1, 28, 28)  (−1, 10, 24, 24)  [(10, 1, 5, 5), (10,)]  260           260          {'stride': (1, 1), 'padding': (0, 0)}
2  BatchNorm2d  (−1, 10, 24, 24) (−1, 10, 24, 24)  [(10,), (10,)]          20            20           {'affine': True, 'momentum': 0.1}
3  MaxPool2d    (−1, 10, 24, 24) (−1, 10, 12, 12)  []                      0             0            {'stride': (2, 2), 'kernel_size': (2, 2), 'padding': 0, 'dilation': 1}
4  ReLU         (−1, 10, 12, 12) (−1, 10, 12, 12)  []                      0             0            −
5  Conv2d       (−1, 10, 12, 12) (−1, 20, 8, 8)    [(20, 10, 5, 5), (20,)] 5020          5020         {'stride': (1, 1), 'padding': (0, 0)}
6  BatchNorm2d  (−1, 20, 8, 8)   (−1, 20, 8, 8)    [(20,), (20,)]          40            40           {'affine': True, 'momentum': 0.1}
7  MaxPool2d    (−1, 20, 8, 8)   (−1, 20, 4, 4)    []                      0             0            {'stride': (2, 2), 'kernel_size': (2, 2), 'padding': 0, 'dilation': 1}
8  ReLU         (−1, 20, 4, 4)   (−1, 20, 4, 4)    []                      0             0            −
9  Linear       (−1, 320)        (−1, 50)          [(50, 320), (50,)]      16050         16050        −
10 BatchNorm1d  (−1, 50)         (−1, 50)          [(50,), (50,)]          100           100          {'affine': True, 'momentum': 0.1}
11 ReLU         (−1, 50)         (−1, 50)          []                      0             0            −
12 Linear       (−1, 50)         (−1, 10)          [(10, 50), (10,)]       510           510          −
13 LogSoftmax   (−1, 10)         (−1, 10)          []                      0             0            −

Parameters: 22000
Trainable parameters: 22000
Layers: 13
Trainable layers: 7
```

Listing B.1: The network used for MNIST with batch normalization (MNISTNetBatchnorm)

```
   class_name     input_shape      output_shape             weight_shapes  n_parameters  n_trainable                                                              settings
1  Conv2d       (−1, 1, 28, 28)  (−1, 10, 24, 24)  [(10, 1, 5, 5), (10,)]  260           260          {'stride': (1, 1), 'padding': (0, 0)}
2  MaxPool2d    (−1, 10, 24, 24) (−1, 10, 12, 12)  []                      0             0            {'dilation': 1, 'kernel_size': (2, 2), 'stride': (2, 2), 'padding': 0}
3  ReLU         (−1, 10, 12, 12) (−1, 10, 12, 12)  []                      0             0            −
4  Conv2d       (−1, 10, 12, 12) (−1, 20, 8, 8)    [(20, 10, 5, 5), (20,)] 5020          5020         {'stride': (1, 1), 'padding': (0, 0)}
5  MaxPool2d    (−1, 20, 8, 8)   (−1, 20, 4, 4)    []                      0             0            {'dilation': 1, 'kernel_size': (2, 2), 'stride': (2, 2), 'padding': 0}
6  ReLU         (−1, 20, 4, 4)   (−1, 20, 4, 4)    []                      0             0            −
7  Linear       (−1, 320)        (−1, 50)          [(50, 320), (50,)]      16050         16050        −
8  ReLU         (−1, 50)         (−1, 50)          []                      0             0            −
9  Linear       (−1, 50)         (−1, 10)          [(10, 50), (10,)]       510           510          −
10 LogSoftmax   (−1, 10)         (−1, 10)          []                      0             0            −

Parameters: 21840
Trainable parameters: 21840
Layers: 10
Trainable layers: 4
```

Listing B.2: The network used for MNIST without batch normalization (MNISTNet)

```
    class_name     input_shape       output_shape          weight_shapes  n_parameters  n_trainable                                                                        settings
1   Conv2d       (−1, 1, 28, 28)   (−1, 10, 24, 24)  [(10, 1, 5, 5), (10,)]   260           260           {'stride': (1, 1), 'padding': (0, 0)}
2   MaxPool2d    (−1, 10, 24, 24)  (−1, 10, 12, 12)  []                       0             0             {'stride': (2, 2), 'dilation': 1, 'padding': 0, 'kernel_size': (2, 2)}
3   ReLU         (−1, 10, 12, 12)  (−1, 10, 12, 12)  []                       0             0             –
4   Conv2d       (−1, 10, 12, 12)  (−1, 20, 8, 8)    [(20, 10, 5, 5), (20,)]  5020          5020          {'stride': (1, 1), 'padding': (0, 0)}
5   MaxPool2d    (−1, 20, 8, 8)    (−1, 20, 4, 4)    []                       0             0             {'stride': (2, 2), 'dilation': 1, 'padding': 0, 'kernel_size': (2, 2)}
6   ReLU         (−1, 20, 4, 4)    (−1, 20, 4, 4)    []                       0             0             –
7   Dropout2d    (−1, 20, 4, 4)    (−1, 20, 4, 4)    []                       0             0             {'p': 0.2}
8   Linear       (−1, 320)         (−1, 50)          [(50, 320), (50,)]       16050         16050         –
9   ReLU         (−1, 50)          (−1, 50)          []                       0             0             –
10  Dropout      (−1, 50)          (−1, 50)          []                       0             0             {'p': 0.5}
11  Linear       (−1, 50)          (−1, 10)          [(10, 50), (10,)]        510           510           –
12  LogSoftmax   (−1, 10)          (−1, 10)          []                       0             0             –

Parameters: 21840
Trainable parameters: 21840
Layers: 12
Trainable layers: 4
```

Listing B.3: The network used for MNIST with dropout (MNISTNetDropout)

```
    class_name     input_shape       output_shape          weight_shapes  n_parameters  n_trainable                               settings
1   Conv2d       (−1, 4, 84, 84)   (−1, 32, 20, 20)  [(32, 4, 8, 8), (32,)]   8224          8224          {'stride': (4, 4), 'padding': (0, 0)}
2   ReLU         (−1, 32, 20, 20)  (−1, 32, 20, 20)  []                       0             0             –
3   Conv2d       (−1, 32, 20, 20)  (−1, 64, 9, 9)    [(64, 32, 4, 4), (64,)]  32832         32832         {'stride': (2, 2), 'padding': (0, 0)}
4   ReLU         (−1, 64, 9, 9)    (−1, 64, 9, 9)    []                       0             0             –
5   Conv2d       (−1, 64, 9, 9)    (−1, 64, 7, 7)    [(64, 64, 3, 3), (64,)]  36928         36928         {'stride': (1, 1), 'padding': (0, 0)}
6   ReLU         (−1, 64, 7, 7)    (−1, 64, 7, 7)    []                       0             0             –
7   Linear       (−1, 3136)        (−1, 512)         [(512, 3136), (512,)]    1606144       1606144       –
8   ReLU         (−1, 512)         (−1, 512)         []                       0             0             –
9   Linear       (−1, 512)         (−1, 3)           [(3, 512), (3,)]         1539          1539          –
10  LogSoftmax   (−1, 3)           (−1, 3)           []                       0             0             –

Parameters: 1685667
Trainable parameters: 1685667
Layers: 10
Trainable layers: 5
```

Listing B.4: The network used for Atari environments. This is the DQN network used by [68, 103]

APPENDIX C

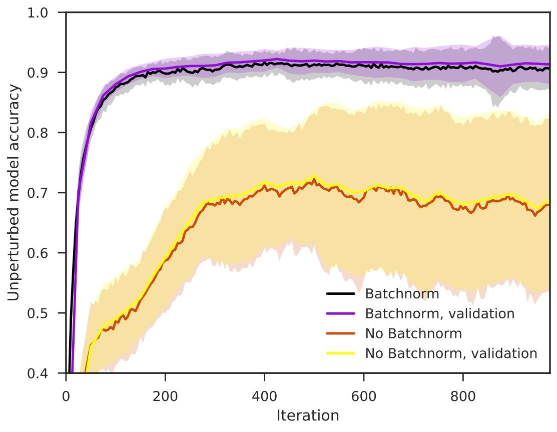
Effects of model and algorithm augmentations (Adam optimizer)

This appendix presents the effects of common model and algorithm augmentations as in Section 4.5 only here the popular Adam optimizer [45] is used instead of regular SGD.

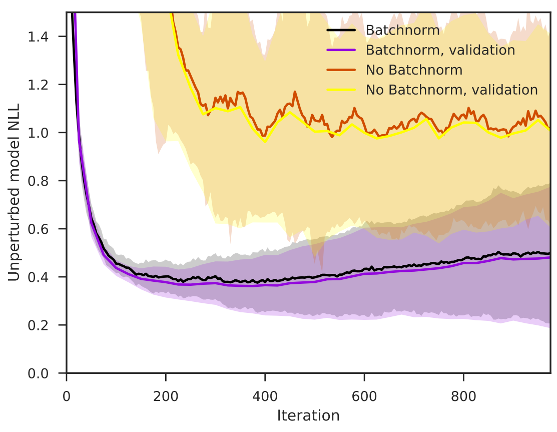
For all figures, **(a)** shows the training and validation set classification accuracy of the unperturbed model while **(b)** shows the training and validation NLL loss.

Some trends from the experiments using the SGD optimizer are also visible when using the Adam optimizer, e.g. the effect of using batch normalization in Figure C.1 and the superiority of using antithetic sampling Figure C.3. Generally speaking however, the Adam optimizer results in more unstable learning with much larger variance within several runs using the same hyperparameters and lower final performance across the line.

The poorer performance of Adam cannot be ruled out to be an issue related to hyperparameter choices since these were found heuristically. It did however prove significantly easier to achieve stable learning using SGD compared to Adam. It is hypothesized that this may have to do with the variance of the stochastically estimated gradient being too high for good estimation of higher order moments, such as those used in Adam. This is in line with the discussion on the variance of the stochastic Hessian estimate presented in Section 3.2.3. It is also supported by the results in Figure C.3 since the between run variance decreases when antithetic sampling is used.

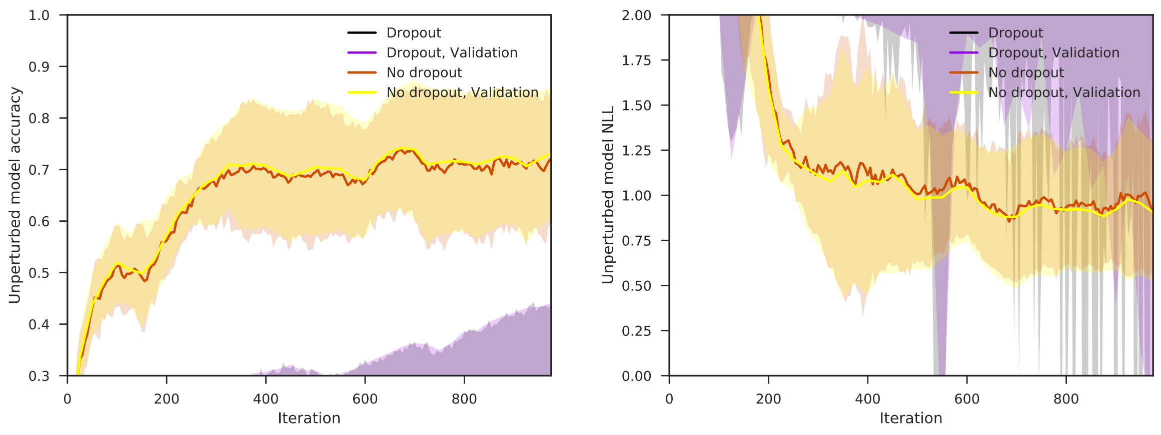


(a)



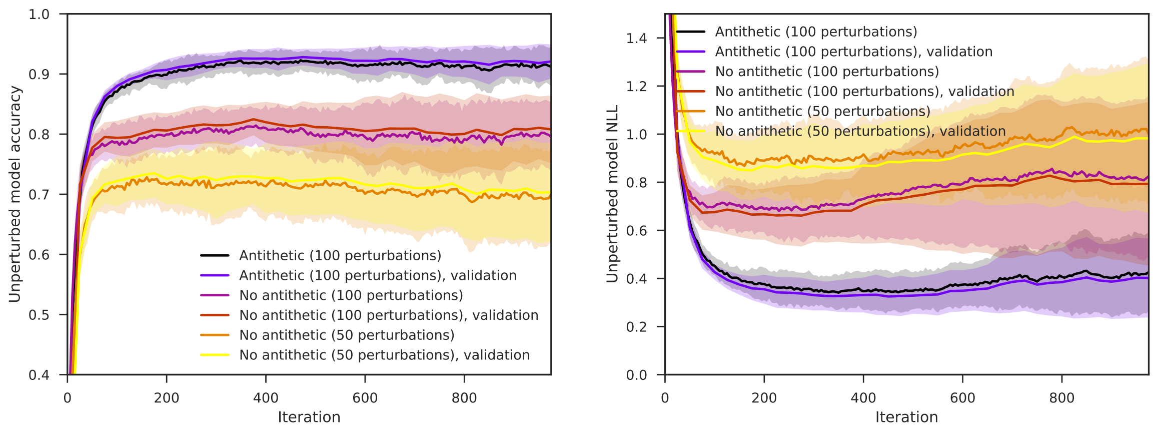
(b)

Figure C.1: Batch normalization experiment using the Adam optimizer.



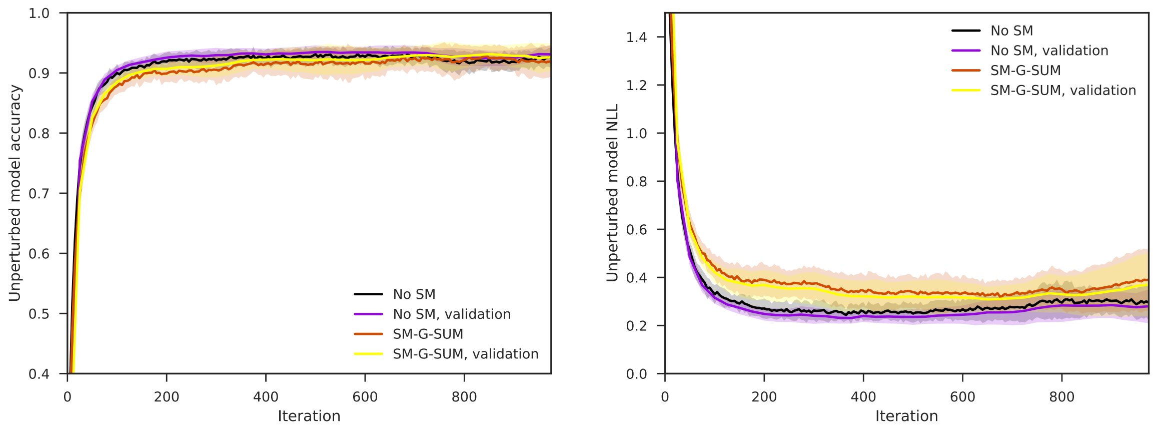
(a) **(b)**

Figure C.2: Dropout experiment using the Adam optimizer.



(a) **(b)**

Figure C.3: Antithetic sampling experiment using the Adam optimizer.



(a) **(b)**

Figure C.4: Safe mutation (SM-G-SUM) experiment using the Adam optimizer.

APPENDIX D

Weighted Mann-Whitney U-test

This appendix defines the weighted Mann-Whitney as it is introduced in [93] for use in adaptation sampling.

D.1 Regular Mann-Whitney U-test

The regular Mann-Whitney U-test seeks to determine if it is equally likely that a random sample from the set $S = \{s_i\}_{i=1}^{n}$ will be less than or greater than a random sample from a second set $S \ = \{s'_i\}_{i=1}^{n'}$ [64]. To this end, the so-called U-test statistic is computed

$$U = \sum_{s_i > s'_j} 1 + \sum_{s_i = s'_j} \frac{1}{2}, \quad \forall i, j \tag{D.1.1}$$

For large samples, U is approximately normally distributed and a standardized statistic can be computed from the estimated mean and standard deviation of U. Let

$$\mu = \frac{nn'}{2} \tag{D.1.2}$$

$$\sigma = \sqrt{\frac{nn'(n + n' + 1)}{12}} \tag{D.1.3}$$

respectively be the mean and standard deviation estimates of U. The standardized statistic is then

$$z = \frac{U - \mu}{\sigma} \tag{D.1.4}$$

The two samples are different with confidence ρ if either

$$\begin{cases} z > 1 - \rho, & S \text{ has larger values} \\ z < \rho, & S' \text{ has larger values.} \end{cases} \tag{D.1.5}$$

D.2 Weighted Mann-Whitney U-test

The generalization provided by [93] introduces weights associated with each of samples in the two sets. Specifically, the sets are now

$$S = \{(w_i, s_i\}_{i=1}^{n}, \quad S' = \{(w'_i, s'_i\}_{i=1}^{n} \tag{D.2.1}$$

for positive weights $w_i, w'_i \geq 0$. The Mann-Whitney test is then generalized by interpreting these weights as fractional occurrence counts. The weighted U-test statistic then becomes

$$U = \sum_{s_i > s'_j} w_i w'_j + \sum_{s_i = s'_j} \frac{1}{2} w_i w'_j, \quad \forall i, j. \tag{D.2.2}$$

The number of samples in each set must then be corrected according to the interpretation of the weights

$$m = \sum_{i=1}^{n} w_i, \quad m' = \sum_{i=1}^{n'} w_i', \tag{D.2.3}$$

and μ and σ computed with m and m' rather than n and n'.

It can be noted that for unit weights, the weighted Mann-Whitney test corresponds to the regular Mann-Whitney test. Additionally, it can be seen that an integer weighted sample, (w_i, s_i) with $w \in \mathbb{N}$, can be replaced by w_i occurrences of the same sample with unit weight.

APPENDIX E

Regular and symmetrized Kullback-Leibler divergence for Gaussian

This appendix derives the regular and the symmetrized KL divergence for a pair of univariate Gaussians. Throughout this appendix, let $p(x) = \mathcal{N}(x|\mu_1, \sigma_1^2)$ and $q(x) = \mathcal{N}(x|\mu_2, \sigma_2^2)$.

E.1 KL divergence

The KL divergence is defined in (3.4.1) and can be written as

$$D_{\mathrm{KL}}(p||q) = \int (\log p(x) - \log q(x)) p(x) \, \mathrm{d}x \tag{E.1.1}$$

where now the univariate case is considered. It follows that

$$D_{\mathrm{KL}}(p||q) = \int_{-\infty}^{\infty} \left(-\frac{1}{2} \log 2\pi\sigma_2^2 - \frac{1}{2} \left(\frac{x-\mu_2}{\sigma_2} \right)^2 + \frac{1}{2} \log 2\pi\sigma_1^2 + \frac{1}{2} \left(\frac{x-\mu_1}{\sigma_1} \right)^2 \right) p(x) \, \mathrm{d}x \tag{E.1.2}$$

$$= \int_{-\infty}^{\infty} \left(\log \left(\frac{\sigma_2}{\sigma_1} \right) + \frac{1}{2} \left(\left(\frac{x-\mu_2}{\sigma_2} \right)^2 - \left(\frac{x-\mu_1}{\sigma_1} \right)^2 \right) \right) p(x) \, \mathrm{d}x \tag{E.1.3}$$

$$= \mathrm{E} \left[\log \left(\frac{\sigma_2}{\sigma_1} \right) + \frac{1}{2} \left(\left(\frac{x-\mu_2}{\sigma_2} \right)^2 - \left(\frac{x-\mu_1}{\sigma_1} \right)^2 \right) \right]_{x \sim p(x)} \tag{E.1.4}$$

$$= \log \left(\frac{\sigma_2}{\sigma_1} \right) + \frac{1}{2\sigma_2^2} \mathrm{E} \left[(x-\mu_2)^2 \right]_{x \sim p(x)} - \frac{1}{2\sigma_1^2} \mathrm{E} \left[(x-\mu_1)^2 \right]_{x \sim p(x)} \tag{E.1.5}$$

$$= \log \left(\frac{\sigma_2}{\sigma_1} \right) + \frac{1}{2\sigma_2^2} \mathrm{E} \left[(x-\mu_2)^2 \right]_{x \sim p(x)} - \frac{1}{2} \tag{E.1.6}$$

Now, the expectation of $(x-\mu_2)^2$ can be written in terms of the known quantities σ_1, μ_1 and μ_2 by force

$$\mathrm{E} \left[(x-\mu_2)^2 \right] = \mathrm{E} \left[(x-\mu_1+\mu_1-\mu_2)^2 \right] \tag{E.1.7}$$

$$= \mathrm{E} \left[(x-\mu_1)^2 + (\mu_1-\mu_2)^2 + 2(x-\mu_1)(\mu_1-\mu_2) \right] \tag{E.1.8}$$

$$= \sigma_1^2 + (\mu_1-\mu_2)^2. \tag{E.1.9}$$

Then

$$D_{\mathrm{KL}}(p||q) = \log \left(\frac{\sigma_2}{\sigma_1} \right) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}. \tag{E.1.10}$$

E.2 Symmetrized KL divergence

The symmetrized KL divergence can be computed by use of its definition and the result for the regular KL divergence as follows.

$$\begin{aligned}
D_{\mathrm{KL}}(p,q) &= D_{\mathrm{KL}}(p||q) + D_{\mathrm{KL}}(q||p) && \text{(E.2.1)}\\
&= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \frac{\sigma_1^2 + (\mu_1-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2} + \log\left(\frac{\sigma_1}{\sigma_2}\right) + \frac{\sigma_2^2 + (\mu_2-\mu_1)^2}{2\sigma_1^2} - \frac{1}{2} && \text{(E.2.2)}\\
&= \log\left(\frac{\sigma_2}{\sigma_1}\right) + \log\left(\frac{\sigma_1}{\sigma_2}\right) + \frac{\sigma_1^2}{2\sigma_2^2} + \frac{\sigma_2^2}{2\sigma_1^2} + (\mu_1-\mu_2)^2\left(\frac{1}{2\sigma_1^2} + \frac{1}{2\sigma_2^2}\right) - 1 && \text{(E.2.3)}\\
&= \frac{\sigma_1^2}{2\sigma_2^2} + \frac{\sigma_2^2}{2\sigma_1^2} + (\mu_1-\mu_2)^2\left(\frac{1}{2\sigma_1^2} + \frac{1}{2\sigma_2^2}\right) - 1 && \text{(E.2.4)}\\
&= \left(\sigma_1^2 + \sigma_2^2\right)\left(\frac{1}{2\sigma_1^2} + \frac{1}{2\sigma_2^2}\right) + \frac{\sigma_2^2}{2\sigma_1^2} + (\mu_1-\mu_2)^2\left(\frac{1}{2\sigma_1^2} + \frac{1}{2\sigma_2^2}\right) && \text{(E.2.5)}\\
&= \frac{1}{2}\Big(\left(\sigma_1^2 + \sigma_2^2\right) + (\mu_1-\mu_2)^2\Big)\left(\frac{1}{2\sigma_1^2} + \frac{1}{2\sigma_2^2}\right) && \text{(E.2.6)}
\end{aligned}$$

APPENDIX F

Interpretation of Boxplots

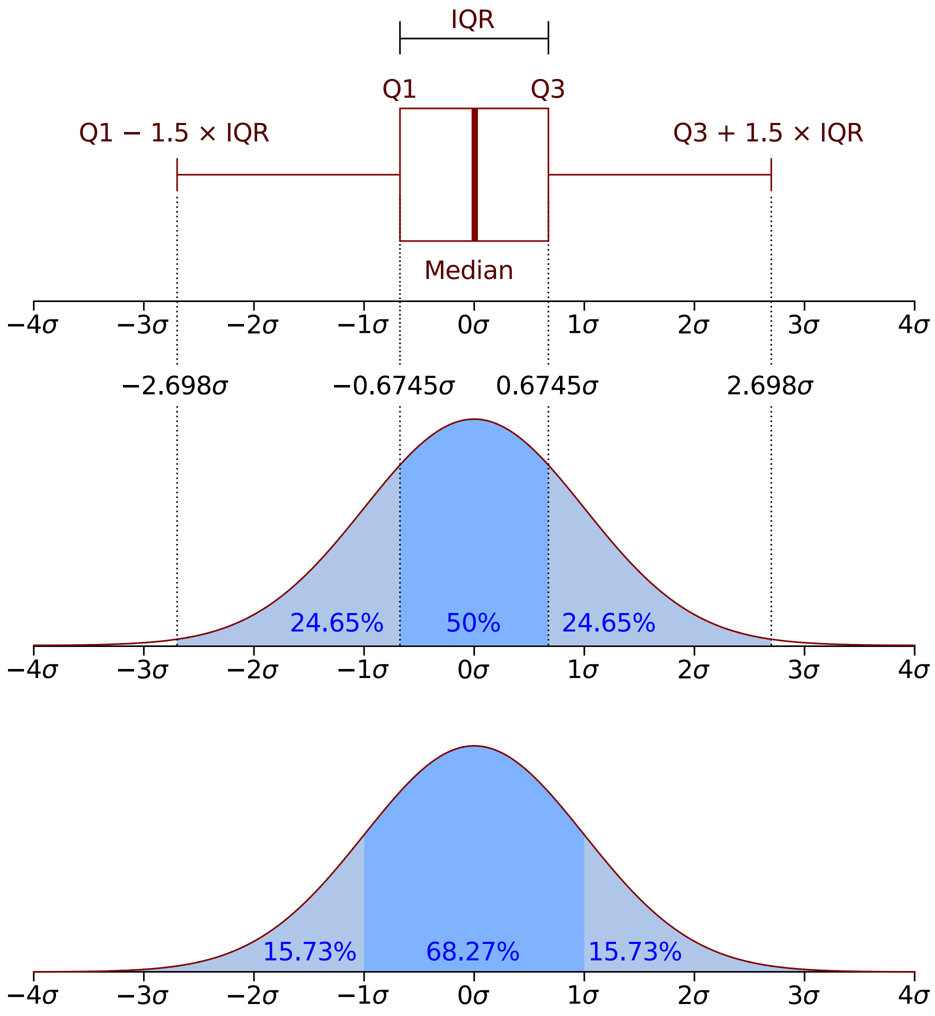


Figure F.1: Interpretation of boxplots. This figures illustrates how to interpret the boxplots shown in this thesis. Note that the median, first and second quartiles and the interquartile range (IQR) are defined in relation to the boxplot and put into the perspective of normally distributed data. Obviously, not all data is normal which can for instance result in non-symmetric boxplots. Figure due to `https://en.wikipedia.org/wiki/Interquartile_range`.

APPENDIX G

Variance of isotropic Gaussian estimator

The variance of the isotropic Gaussian gradient estimator is

$$\mathrm{Var}\left[\nabla_{\mathbf{x}} f(\mathbf{x})\right] \approx \mathrm{Var}\left[\frac{1}{N\sigma}\sum_{n=1}^{N} f(\mathbf{x}+\sigma\boldsymbol{\epsilon}_n)\boldsymbol{\epsilon}_n\right] = \frac{1}{N\sigma^2}\mathrm{Var}[f(\mathbf{x}+\sigma\boldsymbol{\epsilon})\boldsymbol{\epsilon}] \ . \tag{G.0.1}$$

By Taylor expansion for small $\boldsymbol{\epsilon}$ or σ, $f(\mathbf{x}+\sigma\boldsymbol{\epsilon}) \approx f(\mathbf{x}) + \sigma\boldsymbol{\epsilon}^{\mathrm{T}}\nabla_{\mathbf{x}} f(\mathbf{x})$ and

$$\begin{aligned}\mathrm{Var}\left[\nabla_{\mathbf{x}} f(\mathbf{x})\right] &\approx \frac{1}{N\sigma^2}\mathrm{Var}\left[f(\mathbf{x})\boldsymbol{\epsilon} + \sigma\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\nabla_{\mathbf{x}} f(\mathbf{x})\right] \\ &= \frac{1}{N\sigma^2}\left(f(\mathbf{x})^2\mathbf{I} + \sigma^2(d+1)\nabla_{\mathbf{x}} f(\mathbf{x})\nabla_{\mathbf{x}} f(\mathbf{x})^{\mathrm{T}}\right) \\ &= \frac{1}{N\sigma^2} f(\mathbf{x})^2 + \frac{d+1}{N}\nabla_{\mathbf{x}} f(\mathbf{x})\nabla_{\mathbf{x}} f(\mathbf{x})^{\mathrm{T}}\end{aligned} \tag{G.0.2}$$

where it has been used that

$$\begin{aligned}\mathrm{Var}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right] &= \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] - \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right]\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right]^{\mathrm{T}} \\ &= \mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] - \mathbf{I}\end{aligned} \tag{G.0.3}$$

and $\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right]$ has been determined by simulation.

To perform this simulation, $\boldsymbol{\epsilon}$ is drawn as a d dimensional standard Gaussian distributed vector. In $\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}$ the dimensions are

$$(d\times 1)\times(1\times d)\times(1\times d)\times(d\times 1) \ . \tag{G.0.4}$$

First, $\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}$ is computed to form a $d\times d$ matrix after which $\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}$ is computed to form a scalar. In dimensions, this is

$$(d\times d)\times 1 \tag{G.0.5}$$

which is effectively a third order tensor, as expected for the covariance of a random matrix. Then $\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}$ is multiplied by that scalar to give the wanted quantity $\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}$. This is repeated a high number of times and the mean of the result is computed to give the simulated result of $\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right]$. To uncover the dependency on d, this process is then repeated for $d\in[1,\dots,10]$. The result is

$$\mathrm{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}^{\mathrm{T}}\boldsymbol{\epsilon}\right] = (d+2)\mathbf{I} \tag{G.0.6}$$

such that

$$\mathrm{Var}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathrm{T}}\right] = (d+2)\mathbf{I} - \mathbf{I} = (d+1)\mathbf{I} \tag{G.0.7}$$

and the covariance matrix is diagonal.

Bibliography

[1] Alain, Guillaume et al. "Variance Reduction in SGD by Distributed Importance Sampling". In: *arXiv preprint* (November 2015). arXiv: `1511.06481`.

[2] Amari, Shun-Ichi. "Natural Gradient Works Efficiently in Learning". In: *Neural Computation* 10.2 (1998), pages 251–276. ISSN: 0899-7667. DOI: `10.1162/089976698300017746`.

[3] Arulkumaran, Kai et al. *Deep reinforcement learning: A brief survey.* August 2017. DOI: `10.1109/MSP.2017.2743240`. arXiv: `1708.05866`. URL: `http://arxiv.org/abs/1708.05866`.

[4] Belabbas, Mohamed Ali and Wolfe, Patrick J. "Fast low-rank approximation for covariance matrices". In: *IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP).* IEEE, December 2007, pages 293–296. ISBN: 9781424417148. DOI: `10.1109/CAMSAP.2007.4498023`.

[5] Bengio, Yoshua, Boulanger-Lewandowski, Nicolas, and Pascanu, Razvan. "Advances in optimizing recurrent networks". In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 2013, pages 8624–8628. ISBN: 9781479903566. DOI: `10.1109/ICASSP.2013.6639349`. arXiv: `1212.0901`.

[6] Bengtsson, Thomas, Bickel, Peter, and Li, Bo. "Curse-of-dimensionality revisited: Collapse of the particle filter in very large scale systems". In: *Probability and Statistics: Essays in Honor of David A. Freedman.* 2008, pages 316–334. ISBN: 0940600749. DOI: `10.1214/193940307000000518`. arXiv: `0805.3034`.

[7] Bergstra, James and Bengio, Yoshua. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13 (2012), pages 281–305.

[8] Bishop, Christopher M. *Pattern Recognition and Machine Learning.* Springer, 2006. ISBN: 978-0-387-31073-2.

[9] Block, H. D., Knight, Bruce, and Rosenblatt, Frank. "Analysis of a Four-Layer Series-Coupled Perceptron II". In: *Reviews of Modern Physics* 34.1 (1962).

[10] Brockman, Greg et al. "OpenAI Gym". In: *arXiv preprint* (2016). arXiv: `1606.01540`.

[11] Carpentier, Alexandra and Munos, Rémi. "Adaptive Stratified Sampling for Monte-Carlo integration of Differentiable functions." In: *Proceedings of the Conference on Neural Information Processing Systems (NIPS).* 2012, pages 1–23. ISBN: 9781627480031. arXiv: `arXiv:1210.5345v1`.

[12] Chan, T. F., Golub, G. H., and LeVeque, R. J. "Updating formulae and a pairwise algorithm for computing sample variances". In: *Annals of Physics* 54 (1979), page 258. DOI: `10.1007/978-3-642-51461-6_3`.

[13] Chang, Jia-Ren and Chen, Yong-Sheng. "Batch-normalized Maxout Network in Network". In: *arXiv preprint* (2015). arXiv: `1511.02583`.

[14] Cho, Kyunghyun et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *arXiv preprint* (2014). ISSN: 09205691. DOI: 10.3115/v1/D14-1179. arXiv: 1406.1078.

[15] Christensen, Ole. *Functions, Spaces, and Expansions.* Boston: Birkhäuser, 2010. ISBN: 978-0-8176-4979-1. DOI: 10.1007/978-0-8176-4980-7. arXiv: arXiv:1011.1669v3.

[16] Conti, Edoardo et al. "Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents". In: *arXiv preprint* (December 2017). arXiv: 1712.06560.

[17] Cortes, Corinna and Vapnik, Vladimir. "Support-Vector Networks". In: *Journal of Machine Learning* 20.3 (1995), pages 273–297.

[18] Dauphin, Yann et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *arXiv preprint* (2014). ISSN: 10495258. arXiv: 1406.2572.

[19] Dozat, Timothy. "Incorporating Nesterov Momentum into Adam". In: *International Conference on Learning Representations (ICLR) Workshop* 1 (2016), pages 2013–2016.

[20] Duan, Xuefeng et al. "Low rank approximation of the symmetric positive semidefinite matrix". In: *Journal of Computational and Applied Mathematics* 260 (2014), pages 236–243. ISSN: 03770427. DOI: 10.1016/j.cam.2013.09.080.

[21] Duchi, John, Hazan, Elad, and Singer, Yoram. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12 (2011), pages 2121–2159.

[22] Dumoulin, Vincent and Visin, Francesco. "A Guide to Convolution Arithmetic for Deep Learning". In: *arXiv preprint* (March 2016). arXiv: 1603.07285.

[23] Fukushima, Kunihiko. "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* 36.4 (1980), pages 193–202.

[24] Garipov, Timur et al. "Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs". In: *arXiv preprint* (2018). arXiv: 1802.10026.

[25] Gers, F.A. and Schmidhuber, Jürgen. "Recurrent nets that time and count". In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN).* 2000, 189–194 vol.3.

[26] Glasserman, Paul. *Monte Carlo Methods in Financial Engineering.* Volume 53. 2003. ISBN: 978-1-4419-1822-2. DOI: 10.1007/978-0-387-21617-1. arXiv: 1011.1669v3.

[27] Glorot, Xavier and Bengio, Yoshua. "Understanding the difficulty of training deep feed-forward neural networks". In: *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS).* Volume 9. 2010, pages 249–256.

[28] Glorot, Xavier, Bordes, Antoine, and Bengio, Yoshua. "Deep sparse rectifier neural networks". In: *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)* 15 (2011), pages 315–323. arXiv: 1502.03167.

[29] Goodfellow, Ian J., Bengio, Yoshua, and Courville, Aaron. *Deep Learning.* 1st edition. Cambridge, Massachusetts: MIT Press, 2016. ISBN: 9780262035613.

[30] Goodfellow, Ian J, Vinyals, Oriol, and Saxe, Andrew M. “Qualitatively characterizing neural network optimization problems”. In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2015, pages 1–11. arXiv: 1412.6544v5.

[31] Hansen, Nikolaus and Ostermeier, Andreas. “Completely Derandomized Self-Adaptation in Evolution Strategies”. In: *Evolutionary Computation* 9.2 (2001), pages 159–195. ISSN: 1063-6560. DOI: 10.1162/106365601750190398.

[32] Hanson, Stephen José and Pratt, Lorien. “Comparing Biases for Minimal Network Construction with Back-Propagation”. In: *Advances in Neural Information Processing Systems* (1989), pages 177–185.

[33] Hasanpour, Seyyed Hossein et al. “Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures”. In: *arXiv preprint* (2016). arXiv: 1608.06037.

[34] Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome. *The Elements of Statistical Learning*. 2nd edition. Springer, 2009. ISBN: 978-0-387-84857-0. DOI: 10.1007/b94608.

[35] He, Kaiming et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (2015), pages 1026–1034. arXiv: 1502.01852v1.

[36] Heusel, Martin et al. “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium”. In: *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*. 2017. arXiv: 1706.08500.

[37] Hinton, Geoffrey E. and McClelland, James L. *The Development of Time-Delay Neural Network Architecture for Speech Recognition*. Technical report. Carnegie Mellon University, 1988.

[38] Hinton, Geoffrey E. et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint* (2012). arXiv: 1207.0580.

[39] Hochreiter, Sepp and Schmidhuber, Jürgen. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pages 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[40] Huszár, Ferenc. *Evolution Strategies, Variational Optimisation and Natural ES*. 2017. URL: http://www.inference.vc/evolution-strategies-variational-optimisation-and-natural-es-2/ (visited on March 23, 2018).

[41] Ioffe, Sergey and Szegedy, Christian. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. Lille, France, 2015. arXiv: 1502.03167.

[42] Kaelbling, Leslie Pack, Littman, Michael L., and Moore, Andrew W. “Reinforcement learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4 (1996), pages 237–285. DOI: 10.1613/jair.301.

[43] Katharopoulos, Angelos and Fleuret, François. “Biased Importance Sampling for Deep Neural Network Training”. In: *arXiv preprint* (May 2017). arXiv: 1706.00043.

[44] Katharopoulos, Angelos and Fleuret, François. “Not All Samples Are Created Equal: Deep Learning with Importance Sampling”. In: *arXiv preprint* (March 2018). arXiv: 1803.00942.

[45] Kingma, Diederik P. and Ba, Jimmy Lei. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint* (December 2014). arXiv: 1412.6980.

[46] Kingma, Diederik P., Salimans, Tim, and Welling, Max. “Variational Dropout and the Local Reparameterization Trick”. In: *arXiv preprint* (June 2015). arXiv: 1506.02557.

[47] Kingma, Diederik P and Welling, Max. “Auto-Encoding Variational Bayes”. In: *arXiv preprint* (December 2013). arXiv: 1312.6114.

[48] Koutník, Jan and Driessens, Kurt. “A Wavelet-based Encoding for Neuroevolution”. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO).* Denver, CO, USA, 2016, pages 517–524. ISBN: 9781450342063. DOI: 10.1145/2908812.2908905.

[49] LeCun, Yann A. et al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1 (1989), pages 541–551.

[50] LeCun, Yann A. et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pages 2278–2323.

[51] LeCun, Yann A. et al. “Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning”. In: *IEEE Communications Magazine* 27.11 (1989), pages 41–46. DOI: 10.1109/35.41400.

[52] L’Ecuyer, Pierre. “Randomized Quasi-Monte Carlo: An Introduction for Practitioners”. In: *International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing (MCQMC).* 2016.

[53] Lehman, Joel et al. “ES Is More Than Just a Traditional Finite-Difference Approximator”. In: *arXiv preprint* (2017). arXiv: 1712.06568.

[54] Lehman, Joel et al. “Safe Mutations for Deep and Recurrent Neural Networks through Output Gradients”. In: *arXiv preprint* (2017). arXiv: 1712.06563.

[55] Li, Chunyuan et al. “Measuring the Intrinsic Dimension of Objective Landscapes”. In: *Proceedings of the International Conference on Learning Representations (ICLR).* Vancouver, Canada, 2018. arXiv: 1804.08838.

[56] Li, Xiang et al. “Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift”. In: *arXiv preprint* (2018). arXiv: 1801.05134.

[57] Li, Yuxi. “Deep Reinforcement Learning: An Overview”. In: *arXiv preprint* (January 2017), pages 1–70. ISSN: 1701.07274. DOI: 10.1007/978-3-319-56991-8_32. arXiv: 1701.07274.

[58] Lighthill, James. *Lighthill Report: Artificial Intelligence: A Paper Symposium.* Technical report. London, United Kingdom: Science Research Council, 1973.

[59] Lin, Min, Chen, Qiang, and Yan, Shuicheng. “Network In Network”. In: *arXiv preprint* (December 2013), page 10. arXiv: 1312.4400.

[60] Long, Jonathan, Shelhamer, Evan, and Darrell, Trevor. “Fully Convolutional Networks for Semantic Segmentation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CCVPR).* 2015, pages 3431–3440. ISBN: 9781467369640. DOI: 10.1109/CVPR.2015.7298965. arXiv: 1411.4038.

[61] Loshchilov, Ilya and Hutter, Frank. “Fixing Weight Decay Regularization in Adam”. In: *arXiv preprint* (November 2017). arXiv: 1711.05101.

[62] Ly, Alexander et al. “A Tutorial on Fisher Information”. In: *arXiv preprint* (October 2017). arXiv: `1705.01064`.

[63] Magdon-Ismail, Malik and Purnell, Jonathan T. “Approximating the Covariance Matrix with Low-rank Perturbations”. In: *Intelligent Data Engineering and Automated Learning (IDEAL)* (2010), pages 300–307.

[64] Mann, H. B. and Whitney, D. R. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other”. In: *The Annals of Mathematical Statistics* 18.1 (1947), pages 50–60. ISSN: 0003-4851. DOI: `10.1214/aoms/1177730491`.

[65] McCulloch, Warren S. and Pitts, W. “A Logical Calculus of the Idea Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5.4 (1943), pages 115–133. ISSN: 0007-4985. DOI: `10.1007/BF02478259`.

[66] Minsky, Marvin and Papert, Seymour A. *An Introduction to Computational Geometry.* MIT Press, 1969, page 258. ISBN: 0-262-63022-2.

[67] Mnih, Volodymyr et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *Proceedings of the International Conference on Machine Learning (ICML)* (2016). ISSN: 1938-7228. arXiv: `1602.01783`.

[68] Mnih, Volodymyr et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pages 529–533. ISSN: 14764687. DOI: `10.1038/nature14236`.

[69] Morokoff, William J. and Caflisch, Russel E. “Quasi-Monte Carlo Integration”. In: *Journal of Computational Physics* 122.2 (December 1995), pages 218–230. ISSN: 00219991. DOI: `10.1006/jcph.1995.1209`.

[70] Murphy, Kevin P. *Machine Learning.* 1st edition. MIT Press, 2012. ISBN: 9780262018029.

[71] Nesterov, Yurii. “A Method of Solving A Convex Programming Problem With Convergence rate O(1/k^2)”. In: *Soviet Mathematics Doklady* 27.2 (1983), pages 372–376.

[72] Nesterov, Yurii and Spokoiny, Vladimir. “Random Gradient-Free Minimization of Convex Functions”. In: *Foundations of Computational Mathematics* 17.2 (2017), pages 527–566. ISSN: 16153383. DOI: `10.1007/s10208-015-9296-2`.

[73] Nielsen, Michael. *Neural Networks and Deep Learning.* 2015. URL: `http://neuralnetworksanddeeplearning.com/index.html` (visited on May 28, 2018).

[74] Nocedal, Jorge and Wright, Stephen J. *Numerical optimization.* Springer, 2006. ISBN: 0-387-30303-0.

[75] Olah, Chris. *Understanding LSTM Networks.* 2015. URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (visited on May 26, 2018).

[76] OpenCV Development Team. *OpenCV documentation.* 2018. URL: `https://docs.opencv.org/master/`.

[77] Park, Sungheon and Kwak, Nojun. “Analysis on the Dropout Effect in Convolutional Neural Networks”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Volume 10112 LNCS. 2017, pages 189–204. ISBN: 9783319541839. DOI: `10.1007/978-3-319-54184-6_12`.

[78] Paszke, Adam et al. “Automatic differentiation in PyTorch”. In: *Proceedings of the Conference on Neural Information Processing Systems (NIPS).* 2017.

[79] Petersen, Kaare Breandt and Pedersen, Michael Syskind. *The Matrix Cookbook*. Kongens Lyngby, Denmark, 2012. URL: `http://www2.imm.dtu.dk/pubdb/views/edoc%7B%5C_%7Ddownload.php/3274/pdf/imm3274.pdf`.

[80] Qian, Ning. "On the momentum term in gradient descent learning algorithms". In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*. Volume 12. 1. 1999, pages 145–151. ISBN: 1212543521. DOI: `10.1016/S0893-6080(98)00116-6`.

[81] Ranzato, Marc'Aurelio et al. "Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CCVPR)*. June 2007, pages 1–8. ISBN: 1424411807. DOI: `10.1109/CVPR.2007.383157`.

[82] Reddi, S. J., Kale, S., and Kumar, S. "On the Convergence of Adam and Beyond". In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2018, pages 1–23.

[83] Redmon, Joseph et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CCVPR)* (2015), pages 779–788. ISSN: 01689002. DOI: `10.1109/CVPR.2016.91`. arXiv: `1506.02640`.

[84] Risi, Sebastian and Togelius, Julian. "Neuroevolution in Games: State of the Art and Open Challenges". In: *IEEE Transactions on Computational Intelligence and AI in Games* 9.1 (2017). ISSN: 1943068X. DOI: `10.1109/TCIAIG.2015.2494596`. arXiv: `1410.7326`.

[85] Rosenblatt, Frank. "Analytic Techniques for the Study of Neural Nets". In: *Proceedings of AIEE Joint Automatic Control Conference* 401 (1962), pages 285–292.

[86] Rosenblatt, Frank. *The Perceptron - A Perceiving and Recognizing Automaton*. Technical report. Cornell Aeronautical Laboratory, 1957, Report 85–460–1. DOI: `85-460-1`.

[87] Rumelhart, David E., Hinton, Geoffrey E., and Williams, Ronald J. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pages 533–536.

[88] Sabour, Sara, Frosst, Nicholas, and Hinton, Geoffrey E. "Dynamic Routing Between Capsules". In: *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*. Long Beach, CA, USA, October 2017. arXiv: `1710.09829`.

[89] Salimans, Tim and Kingma, Diederik P. "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks". In: *arXiv preprint* (February 2016). ISSN: 09252312. DOI: `http://doi.acm.org.ezproxy.lib.ucf.edu/10.1145/1830483.1830503`. arXiv: `1602.07868`.

[90] Salimans, Tim et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: *arXiv preprint* (2017). arXiv: `1703.03864`.

[91] Samuel, Artur L. "Some Studies in Machine Learning using the Game of Checkers". In: *IBM Journal of research and development* 3.3 (1959), pages 210–229. DOI: `10.1147/rd.33.0210`.

[92] Schaul, Tom. "Investigating the Impact of Adaptation Sampling in Natural Evolution Strategies on Black-box Optimization Testbeds". In: *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)* (2012), page 221. DOI: `10.1145/2330784.2330817`.

[93] Schaul, Tom and Brügge, B. “Studies in Continuous Black-box Optimization”. PhD Thesis. Technical University of Munich, 2011.

[94] Schaul, Tom, Glasmachers, Tobias, and Schmidhuber, Jürgen. “High Dimensions and Heavy Tails for Natural Evolution Strategies”. In: *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)*. 2011. ISBN: 9781450305570. DOI: 10.1145/2001576.2001692.

[95] Scherer, Dominik, Müller, Andreas, and Behnke, Sven. “Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Volume 6354 LNCS. PART 3. 2010, pages 92–101. ISBN: 3642158242. DOI: 10.1007/978-3-642-15825-4_10.

[96] Schmidhuber, Jürgen. “Deep Learning in Neural Networks: An Overview”. In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)* 61 (2015), pages 85–117. ISSN: 18792782. DOI: 10.1016/j.neunet.2014.09.003. arXiv: 1404.7828.

[97] Schmidhuber, Jürgen and Zhao, Jieyu. *Direct Policy Search and Uncertain Policy Evaluation*. Technical report. 1999, pages 119–124.

[98] Schulman, John et al. “Proximal Policy Optimization Algorithms”. In: *arXiv preprint* (2017). arXiv: 1707.06347.

[99] Schulman, John et al. “Trust Region Policy Optimization”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. Lille, France, 2015. arXiv: 1502.05477.

[100] Sehnke, Frank et al. “Parameter-exploring policy gradients”. In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)* 23.4 (May 2010), pages 551–559. ISSN: 08936080. DOI: 10.1016/j.neunet.2009.12.004.

[101] Sermanet, Pierre et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: *arXiv preprint* (December 2013), page 1312.6229. arXiv: 1312.6229.

[102] Silver, David et al. “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the International Conference on Machine Learning (ICML)* (2014), pages 387–395. ISSN: 1938-7228.

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

[104] Spall, James C. “Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation”. In: *IEEE Transactions on Automatic Control* 37.3 (1992), pages 332–341. ISSN: 15582523. DOI: 10.1109/9.119632.

[105] Srivastava, Nitish et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pages 1929–1958.

[106] Staines, Joe and Barber, David. “Variational Optimization”. In: *arXiv preprint* (December 2012). arXiv: 1212.4507.

[107] Stanley, Kenneth O., D ’ambrosio, David, and Gauci, Jason. “A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks”. In: *Artificial Life Journal* 15.2 (2009), pages 1–28. ISSN: 1064-5462. DOI: 10.1162/artl.2009.15.2.15202.

[108] Such, Felipe Petroski et al. “Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning”. In: *arXiv preprint* (December 2017). arXiv: `1712.06567`.

[109] Sugiyama, Masashi, Kawanabe, Motoaki, and Chui, Pui Ling. “Dimensionality Reduction for Density Ratio Estimation in High-dimensional Spaces”. In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)* 23.1 (2010), pages 44–59. ISSN: 08936080. DOI: `10.1016/j.neunet.2009.07.007`.

[110] Sun, Yi et al. “Efficient Natural Evolution Strategies”. In: *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO)*. 2009, page 539. ISBN: 9781605583259. DOI: `10.1145/1569901.1569976`. arXiv: `1209.5853v1`.

[111] Sutskever, Ilya et al. “On the Importance of Initialization and Momentum in Deep Learning”. In: *Proceedings of the International Conference on Machine Learning (ICML)* 28.2010 (2013), pages 8609–8613.

[112] Sutton, Richard S. and Barto, Andrew G. *Reinforcement Learning: An Introduction.* 1998. ISBN: 9780262193986.

[113] Suzuki, Mashbat. “Information Geometry and Statistical Manifold”. In: *arXiv preprint* (October 2014). arXiv: `1410.3369`.

[114] Turek, Javier S. and Huth, Alexander. “Efficient, sparse representation of manifold distance matrices for classical scaling”. In: *arXiv preprint* (2017), pages 1–21. arXiv: `1705.10887`.

[115] Turing, Alan M. “Computing machinery and intelligence”. In: *Mind* 59.236 (1950), pages 433–460.

[116] Wan, Li et al. “Regularization of Neural Networks using Dropconnect”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. 1. 2013, pages 109–111. DOI: `10.1109/TPAMI.2017.2703082`.

[117] Welford, B. P. “Note on a Method for Calculating Correct Sums of Squares and Products”. In: *Technometrics* 4.3 (1962), pages 419–420. ISSN: 00401706.

[118] Wierstra, Daan et al. “Natural Evolution Strategies”. In: *Journal of Machine Learning Research* 15 (2008), pages 3381–3387. ISSN: 15337928. DOI: `10.1109/CEC.2008.4631255`. arXiv: `1106.4487`.

[119] Williams, Ronald J. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Journal of Machine Learning* 8.3 (1992), pages 229–256. DOI: `10.1007/978-1-4615-3618-5_2`.

[120] Wingate, David and Weber, Theophane. “Automated Variational Inference in Probabilistic Programming”. In: *arXiv preprint* (2013). arXiv: `1301.1299`.

[121] Xiao, Han, Rasul, Kashif, and Vollgraf, Roland. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. In: *arXiv preprint* (2017). arXiv: `1708.07747`.

[122] Yi, Sun et al. “Stochastic Search using the Natural Gradient”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. Montreal, Quebec, Canada, 2009, pages 1161–1168. ISBN: 9781605585161. DOI: `10.1145/1553374.1553522`.

[123] Zeiler, Matthew D. “ADADELTA: An Adaptive Learning Rate Method”. In: *arXiv preprint* (December 2012). arXiv: 1212.5701.

[124] Zhang, Xingwen, Clune, Jeff, and Stanley, Kenneth O. “On the Relationship Between the OpenAI Evolution Strategy and Stochastic Gradient Descent”. In: *arXiv preprint* (2017). arXiv: 1712.06564.

[125] Zhao, Peilin and Zhang, Tong. “Stochastic Optimization with Importance Sampling”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. Lille, France, 2014. arXiv: 1401.2753.

[126] Zhou, Y. T. and Chellappa, R. “Computation of Optical Flow using a Neural Network”. In: *Proceedings of the IEEE International Conference on Neural Networks*. 86. IEEE, 1988, 71–78 vol.2. ISBN: 0-7803-0999-5. DOI: 10.1109/ICNN.1988.23914.