Comparison and Implementation of Algorithms for Neural Network Training

Mateusz Pyla

Bachelor of Science

School of Informatics

University of Edinburgh

2020

Abstract

Over the course of last couple of years, deep learning has become overwhelmingly popular among all machine learning methods. In all supervised learning approaches, the model parameters are determined by the learning algorithms - in the case of neural networks these parameters are the weights between the layers. In many instances, the process is equivalent to minimizing a differentiable objective function using some optimization methods. This includes the most popular one; backpropagation is an effective way of implementing dynamic programming and chain rule with gradient descent. However, there are a variety of competitive algorithms. The aim of this thesis is to present and compare them discussing their potential limitations and pointing out the situations in which it makes sense to use each approach. We claim that when working with feedforward neural networks, there are various algorithms that can train neural networks equally well, however the biggest advantage of the standard backpropagation is its universality.

Keywords: neural network, deep learning, learning algorithms, supervised learning, backpropagation, quadratic programming, plausible deep learning, conjugate gradient, entropy algorithms, stochastic gradient method

Acknowledgements

I would like to thank my supervisor, Dr Shay B. Cohen who suggested this insightful topic to me, accompanied me and kept me focused throughout the whole project.
I am also very thankful for my family for guiding me throughout my whole education and supporting me in making my crucial decisions.
I would like to thank Azucena Garvia for her suggestions and comments about the clarity of the report.

Table of Contents

Chapter 1

Introduction

Machine learning is underlying learning from past by trying to fit parameters of a model in order to capture the relation between the observed data. For different families of architectures there are different learning algorithms. Due to the rapid growth of deep learning, in this thesis we will focus on feedforward neural networks. We aim to create a comprehensive and self-contained review and to compare learning algorithms for neural networks.

Due to their complex structures, understanding the differences between different approaches towards building neural networks models requires relevant mathematical knowledge; this will be presented in chapter 2. We will go through probability in subsection 2.1.1, information theory (subsubsection 2.1.1.1), and some approaches in mathematical optimization (subsection 2.1.3 and subsection 2.1.4). We will define, describe, and formulate machine learning, its objectives, general methodology (section 2.2) and discuss neural networks, their classification and some related problems which often appear (section 2.3).

In chapter 3 we will present different learning algorithms for neural networks. We will start with the most popular technique: gradient descent with error backpropagation, and discuss it in detail, pointing out its strengths and weaknesses. There are several techniques that aim to amend these weaknesses and we claim that this makes them advantageous in the certain types of situations. In this chapter, we show numerous optimizers which facilitate the training of neural networks with gradient descent. We present the Hilbert-Schmidt Independence Criterion, preReLU-TLRN with quadratic programming approach, the Broyden–Fletcher–Goldfarb–Shanno algorithm, probably approximately correct, and conjugate gradient methods. We state that there are many algorithms that achieve similar performance as backpropagation.

In section 3.6 we will present our contribution, an stochastic algorithm which finds solutions using conjugate gradient approach and probabilistic techniques.

In chapter 4 we will present some experimental results on different data sets; a few of these are created by us whereas the others are well known in the field.

Finally, in chapter 5 we will wrap up our discussion with denouements and inferences. Furthermore, we will indicate the directions in which we would follow in the future and present the early insights that we discovered which we have not finished due to the time constraints.

Chapter 2

Background

In this chapter we will introduce the notation used in this thesis and show the most relevant mathematical concepts, recalling the most important results in probability, statistics, information theory and mathematical optimization which are essential for building intelligent systems. We will present the background needed to study machine learning and finally we will focus on architecture of neural networks.

2.1 Relevant mathematics

Working with complex models is usually challenging and requires links between a few branches of mathematics. The main reason of mentioning all of these branches is to assure that all readers coming from different backgrounds are able to see the connections without requiring prior reading. Proper and longer definitions and derivations are attached for the curious reader in the appendix.

In this thesis, we will denote the mathematical space in calligraphic font (e.g. $\mathcal{H}$). We will use bolded lower case letters to denote multidimensional variables in order to emphasise that it is a vector rather than just a single number. The p-norm of a vector is denoted by $\|\mathbf{y}\|_p$. We will use capital letters to denote matrices. For matrix A, A_{ij} is the cell corresponding to an element in i^{th} row and j^{th} column, whereas $\mathbf{A_k}$ corresponds to k^{th} row. The identity matrix $D \times D$ will be written as $\mathbf{1}_D$. The `ReLU` function is given by $x^+ := max\{x, 0\}$ and the `sigmoid` function $\sigma(x) = \frac{e^x}{e^x+1}$.

2.1.1 Probability

`Probability` is a real-valued function, usually defined from an event sample space Ω to the unit interval $I = [0,1]$ and we will denote it by P. We require that P obeys all measurable conditions, therefore $P(\Omega) = 1$ and $P(\emptyset) = 0$. If a set A has measure 0 this implies $P(\Omega - A) = 1$.

A `probability mass function` for a discrete random variable $\mathbb{X}$ is given by $p_{\mathbb{X}}(x_i) = P(\mathbb{X} = x_i)$ where x_i are drawn from sample space, i.e. $x_i \in \Omega$. For continuous case, it makes more sense to talk about `probability density functions` defined as $P[a \leq X \leq b] = \int_a^b f_{\mathbb{X}}(x)dx$ for any $a, b \in \mathbb{R}$.

The most important property of random variables is `expectation`, which is defined as $\mathbb{E}[X] = \int_\Omega X(\omega)\, d\mathrm{P}(\omega)$. For discrete random variables, the integrand becomes finite sum.

Furthermore, note that we can compute the expected value of the product of two random variables as well as the expected value of a measurable function applied to a random variable:

$$\begin{aligned} \mathrm{E}(\alpha(X)\beta(Y)) &= \int_{\Omega_2}\int_{\Omega_1} \alpha(X(\omega_1))\beta(Y(\omega_2))\, d\mathrm{P}(\omega_1) d\mathrm{P}(\omega_2) \\ &= \int\int_{\mathbb{R}^2} \alpha(x)\beta(y)\; f_{XY}(x,y)\; \mathrm{d}x\mathrm{d}y \end{aligned}$$

where we integrate over the cartesian products of both sample spaces.

For two events A and B, the `conditional probability` of A given B is denoted as $P(A|B)$ and equals $\frac{P(A \cap B)}{P(B)}$. If $P(A|B) = P(A)$ then we call the events A and B `independent`. We say that two random variables $\mathbb{X}$ and $\mathbb{Y}$ are `independent` if and only if for all events A (from Ω_X) and B (from Ω_Y) we have $P(A|B) = P(A)$ and $P(B|A) = P(B)$.

One of the most important (and simultaneously simplest) results in classical probability is `Bayes formula`. It can be stated as: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$. It is easily applied to any artificial intelligence problem where we need to do inference or estimate the probability of unobservable data. The theorem can also be visualized as the equality between posterior distribution and normalized product of likelihood and prior distributions.

2.1.1.1 Information Theory

In the classical sense, information `entropy` is the average rate at which information is produced by a stochastic source of data. Let us define `Shannon entropy` to be $H = -\sum_i P_i \log P_i$ where P_i corresponds to some probability mass function. We will also use the notation $H(\mathbb{X}) = -\sum_{x \in \mathbb{X}} p(x) \log p(x)$ in order to emphasise when we talk about entropy of random variable.

Therefore we define the `joint entropy` of two random variables $\mathbb{X}$ and $\mathbb{Y}$ as $H(\mathbb{X}, \mathbb{Y}) = \mathbb{E}_{X,Y}[-\log p(x,y)] = -\sum_{x,y} p(x,y) \log p(x,y)$.

In a similar way we define the `conditional entropy`. Although the definition is naturally inherited from previous definitions, the very important formula related to conditional entropy is: $H(X|Y) = H(X,Y) - H(Y)$.

We define the `mutual information` between two random variables to be $I(X;Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)\,p(y)}$.

One can show that $I(X;Y) = H(X) - H(X|Y)$, [Gray, 2013]

Combining all equations, we are left with the relation between mutual information and entropies: $I(X;Y) = H(X) + H(Y) - H(X,Y)$.

2.1.2 Linear Regression

`Linear regression` is a very common statistical method which is used to find a relationship between random variables; they are usually denoted by the predictor variable and the responses variable. In general, we want to find a linear model to reflect the relation between them in the best way.

When we work on data with dimensionality 2, i.e. we have exactly one explanatory random variable we call the process `simple linear regression` which is equivalent to finding a line in $\mathbb{R}^2$ that minimizes the distance (a.k.a. `residuals`) between the points and the line. Linear regression is the generalization of this idea for arbitrary multidimensional data.

Assume we work on $(p+1)$-dimensional data. Let us denote the first p independent dimensions in matrix form as X and the last dimension (on which we perform regression) as $\mathbf{y}$, where

$$X = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \mathbf{x}_2^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix} \text{ and } \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

Let us notice that all elements in the first column of X are deliberately set to 1, as they correspond to the constant term, which we call bias.

Our objective is to find the parameters of the model $\hat{\mathbf{y}} = XW$ which minimize the distance between the vectors $\mathbf{y}$ and $\hat{\mathbf{y}}$ where $W = [\mathbf{w}_0 \mathbf{w}_1 \dots \mathbf{w}_p]^T$. Usually we use L_2 `Lasso normalization`, i.e. we are interested in minimizing $\|\mathbf{y} - \hat{\mathbf{y}}\|_2$, or equivalently $\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$. Using least-squares technique (sum of mean squared loss) the analytical solution (in a closed form) is given by: $\hat{W} = (X^T X)^{-1} X^T \mathbf{y}$.

Using Bayes theorem, one can establish posterior belief about the parameters of a linear regression. This approach is called `Bayesian linear regression` and it has become increasingly popular in machine learning. Consider: $y_i = \mathbf{x}_i^T \beta + \varepsilon_i$, $\varepsilon_i \sim N(0, \sigma^2)$.

In order to use the Bayesian method one begins with finding out the likelihood function of the data. Then we either:

- choose a suitable a prior probability distribution (using the technique of conjugate distribution)
- apply `Markov Chain Monte Carlo` method; which is a numerical algorithm for sampling from a difficult distribution

to compute the analytical or estimated solution, i.e. parameters and model evidence. Bayesian linear regression is not a deterministic model; for each datapoint, our output y will vary in accordance to the found probability distribution. Such models are much more flexible and able to represent more complex models. [Minka, 2000]

2.1.3 Optimization and Quadratic Programming

Let us consider the following problem:

minimize $\frac{1}{2}\mathbf{x}^{\mathrm{T}}Q\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x}$
subject to $A\mathbf{x} \preceq \mathbf{b}$ where $\mathbf{x} \geq 0$

We call this class of `Quadratic Programming` problems as the objective function is of degree 2 and all constraints are linear.

Depending on the constraints, convexity of the feasible set and semi-positiveness of the matrix Q given in the objective function, there are various solvers. Among them, there are at least two very general approaches: the `line search` and the `interior-point method`. Interior-point methods have been shown to underperform in practice and their worst case complexity time is not polynomial. [Nocedal and Wright, 2006] In line search we choose the optimal scalar for how much we want to move along the preferred direction. One of the variation, the `Conjugate gradient method` will be presented in 2.1.4.

Quadratic programming problems are a very powerful group of problems in mathematical optimization, especially in `convex optimization` (the one where the objective and all constraint function satisfy $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$ for all x and y in convex feasible set). However it can be easily extended to non-convex problems. Let us consider a general constrained optimization problem.

minimize $f(\mathbf{x})$
subject to $g(\mathbf{x}) \preceq \mathbf{0}$

Let us recall the `Taylor series` at a point a of a complex-valued function f is the infinite sum $\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$. One can approximate the Taylor series by a Taylor d-degree polynomial g of the function f at a point a given by $g(x) = \sum_{n=0}^{d} \frac{f^{(n)}(a)}{n!}(x-a)^n$. If we use second-order approximation of twice differentiable f, we notice that it will be in a form equivalent to objective function in a standard quadratic programming form.

This observation leads to the next well-known method in optimization called `sequential quadratic programming`.

Let us pick a feasible point. If we restrict our attention to a small area around that point, we can approximate our objective function f with its second-order Taylor expansion. This approach is called the `Trust-region methods`. Then iteratively, we are able to specify the more sub-optimal points by solving a standard quadratic programming program and repeating the process.

In `sequential linear-quadratic programming` we not only focus on estimating the model as quadratic functions but also we apply linear programming to determine the next step within feasible set.

In mathematical optimization, each program has its associated `dual` (proper definition in appendix). It is very common that the dual problem is easier to solve, however the `dual gap` (i.e. the difference between the optimal solutions for the primal and for the dual programs) is not always zero. There are some cases, for instance in convex optimization under Karush-Kuhn-Tucker conditions, in which the strong duality theorem is guaranteed to hold. [Robinson, 2013] Therefore we are not always guaranteed that the optimal solution for a dual problem is optimal solution for its primal. Nevertheless, in almost all situations, the dual is very useful to provide a bound for the original problem.

2.1.4 Gradient Descent and Conjugate Gradient Method

In optimization and particularly in machine learning, our task is equivalent to finding the global minimum of a differentiable function f defined from $\mathbf{R}^d$ to $\mathbf{R}$. The standard iterative procedure guesses the initial point $\mathbf{x}_0$ and computes the gradient $\nabla f(\mathbf{x}_0)$. The next point would be given by $\mathbf{x}_{i+1} = \mathbf{x}_i - \eta \nabla f(\mathbf{x}_0)$. This method, proposed by A. Cauchy [Cauchy, 1847] in the XIX century is called `gradient descent` and it is the base for many optimization methods. The biggest challenges related to this approach are:

1. How to find the most suitable learning rate η and how to determine that value.
2. What if evaluating the gradient is expensive or it is even impossible to compute the gradient because of singularity.

We will address these questions in later chapters (subsection 3.1.3). There are many variations on the idea of gradient descent and some of them will be presented in 3.1.

Another very powerful method in optimization, on which we would like to focus later (section 3.6) is the `conjugate gradient method`. Let us consider the unconstrained minimization problem given by $f(x) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$ where $x \in \mathbf{X} \subset \mathbb{R}^d$. The key point is to notice that A is symmetric, as otherwise the coefficient next to $x_i x_j$ would differ from the coefficient corresponding to $x_j x_i$. Starting our iterative process with $\mathbf{x}_0$ we write our gradient in the standard form $\mathbf{d}_0 = -\nabla f(\mathbf{x}_0)$. However, we can

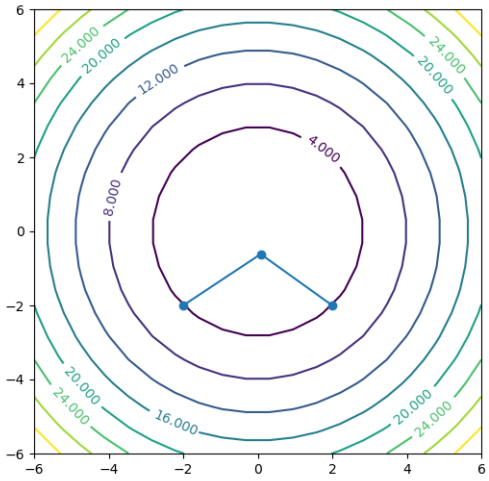


Figure 2.1: Let A be $\left(\begin{smallmatrix}3 & 2\\ 2 & 6\end{smallmatrix}\right)$. We notice that vectors: $\left(\begin{smallmatrix}0.08-(-2)\\ 0.613-(-2)\end{smallmatrix}\right)$ and $\left(\begin{smallmatrix}2-0.08\\ -2-(-0.613)\end{smallmatrix}\right)$ are not orthogonal, but they are conjugate with respect of A. They form a basis for $\mathbb{R}^2$.

take advantage of the fact that the matrix A is symmetric in order to find the efficient learning rate η. We want to minimize the function

$$\begin{aligned} g(\alpha) &= f(\mathbf{x}_0 + \alpha\mathbf{d}_0) \\ &= \frac{1}{2}(\mathbf{x}_0 + \alpha\mathbf{d}_0)^T A(\mathbf{x}_0 + \alpha\mathbf{d}_0) + \mathbf{b}^T(\mathbf{x}_0 + \alpha\mathbf{d}_0) + c \\ &= \frac{1}{2}\alpha^2 \mathbf{d}_0{}^T A\mathbf{d}_0 + \mathbf{d}_0{}^T(A\mathbf{x}_0 + b)\alpha + (\frac{1}{2}\mathbf{x}_0{}^T A\mathbf{x}_0 + \mathbf{x}_0{}^T\mathbf{d}_0 + c). \end{aligned}$$

In order to find the minimum of that quadratic function g we just compute the derivative and set it to 0: $g'(\alpha) = (\mathbf{d}_i{}^T A\mathbf{d}_i)\alpha + \mathbf{d}_i{}^T(A\mathbf{x}_i + \mathbf{b}) = 0$.

If A is positive-definite then $\mathbf{d}_i{}^T A\mathbf{d}_i > 0$ Hence the solution is given by:

$$\alpha = -\frac{\mathbf{d}_0{}^T(A\mathbf{x}_0 + \mathbf{b})}{\mathbf{d}_0{}^T A\mathbf{d}_0}.$$

So far we proceeded in the same way as we would in gradient descent with line search for the η parameter. However, in conjugate gradient descent we add another restriction, we want our direction in each iteration to be orthogonal to all the previous ones. This is why this method is called `conjugate`: we say that a vector $\mathbf{u}$ is conjugate to $\mathbf{v}$ with symmetric matrix A if $\mathbf{u}^T A\mathbf{v} = 0$. Therefore we want $\mathbf{d}_i^T A\mathbf{d}_j = 0$ for all $i, j \in \{1, 2, ..., d\}$ and $i \neq j$. The example of the conjugacy of two vectors with respect to a matrix is given in Figure 2.1.

The natural question that arises is how to find said direction, that which is optimal under the conjugacy condition. In each iteration, we proceed with the `Grahm-Schmidt` procedure:

Firstly, let us define the residuals: $\mathbf{r}_i = \mathbf{b} - \mathbf{A}\mathbf{x}_i$.

Then we set new direction to be $\mathbf{d}_i = \mathbf{r}_i - \sum_{j<i} \frac{\mathbf{d}_j^\top \mathbf{A}\mathbf{r}_i}{\mathbf{d}_j^\top \mathbf{A}\mathbf{d}_j}\mathbf{d}_j$.

Thus our next point would be calculated as $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i$ using line search.

We can notice that since $dim(\mathbf{X}) \leq d$ we need to do this iteration at most d times. After d times we are guaranteed that the next vectors will be linearly dependent in the set of all previous direction vectors and hence all directional vectors will vanish.

In practice, we intend on approximate the solution rather than compute the exact value. Therefore usually we do not perform $dim(\mathbf{X})$ iterations. In general, there is no closed form recipe after how many iterations we should do. Instead, in each iteration we check if the improvement exceeds a threshold.

There is a relation between the conjugate gradients and eigenvalues and eigenvectors of matrix A. Depending on the matrix, we may end up after a few iterations - this implies that the biggest eigenvalues are significantly larger than the smaller ones, or we may need to perform $dim(\mathbf{X})$ iterations - this implies that the eigenvalues are well-balanced.

In mathematical optimization it is very common to use Cholesky decomposition or Singular Value Decomposition when working with conjugate gradients. Both represent a matrix as the product of simpler matrices, which is useful for efficient numerical computations. Moreover, we can infer the eigenvalue decomposition; this can implicitly indicate after how many iterations we should obtain a good enough approximation.

After $dim(\mathbf{X})$ iterations we obtain the solution conjugate gradient method given by $\mathbf{x}_* - \mathbf{x}_0 = \sum \alpha_i \mathbf{d}_i$ for the conjugate basis $\{\mathbf{d}_i\}$. We assumed that A is a positive definite matrix, therefore we are guaranteed that $\mathbf{x}_*$ is a unique global minimum. However, if we relax this condition, we may end up in a local extremum or a saddle point.

2.2 Relevant background in machine learning

A very general, yet comprehensive, definition of `machine learning` is the combination of all data analysis methods and algorithms which enable machines to act without the necessity of explicitly programming them. It is one of the most promising subfield

of artificial intelligence and it is widely applied across all scientific disciplines. We can distinguish three different types of machine learning:

1. `Supervised learning`. With respect to the task, we can divide machine learning problems into `regression` (prediction of a single numerical value) or `classification` problems (assignment to distinguishable clusters). In supervised learning, the model is provided with labelled data and based on this data, our task is to find the most suitable classification or regression for unlabelled data. We will focus on this type in this dissertation.

2. `Unsupervised learning`. This type focuses on finding the similarities and the groupings (clusters) of elements that share common patterns. Semi-supervised technique is a mix between supervised and unsupervised learning.

3. `Reinforcement learning`. This approach allows the model to interact with the environment in order to draw conclusions about appropriate behaviour and decision making.

In supervised techniques, we usually define the cost (objective) function as an error between the predictions and ground-truth labels; the goal is to minimize this objective function. The process is very much related to the optimization and statistics themes introduced in the previous section.

In classification problems we distinguish three approaches:

1. `Generative models` in which we model the prior and likelihood probabilities, and with Bayes theorem we find posterior probability. This model usually requires a lot of parameters and it is expensive, however we have access to probability distributions from which we are able to sample.

2. `Discriminative models` where we directly model the probability and use decision theory. These methods are usually faster and lead to quicker and more accurate results but they require advanced optimization methods.

3. `Discriminant based models` where we directly find a (discriminant) function which maps each data point to a class. Such models are often considered black box methods; they are more efficient but have no meaning probabilistically.

Despite the astonishing results accomplished in machine learning field, we are still very often unable to answer many crucial questions and justify the techniques used and

applied in the state-of-the-art models. There are also many approaches which were introduced, but their full potential is to be discovered in many years after the introduction - for instance `Support Vector Machines` is a widely used learning algorithm which was discovered in 1960s, but only thanks to non-linear basis functions, introduced around 30 years later, SVMs became a real breakthrough.

One of the biggest problems that we encounter in machine learning is the trade-off between bias and variance. We do not want to make our model fit the training data too much and thus not be robust enough and yet we want it to reflect the data structure we provided. This is very closely related with `overfitting` and `underfitting`, respectively.

There are three very common techniques which try to tackle overfitting: gathering more data, `regularization` (i.e. penalizing high coefficients which models too much training data rather than generalize the phenomenon) and model selection (tuning hyperparameters), [Bishop, 2006]. Overfitting is a really important issue in machine learning and assessing a machine learning model should heavily rely on appropriate regularization beforehand, [Kukačka et al., 2017].

Many machine learning models require or simply work better when we firstly preprocess the training data. There are many interesting ways to do so, one of the most notable ones is `principal component analysis`, which maps data points living in $\mathbb{R}^d$ to $\mathbb{R}^k$ where $k < d$ and uses the analysis of eigenvalues to determine the dimension of a new space, in which features are linearly uncorrelated.

2.2.1 Error functions

In this project, we will mainly use two standard loss functions.

- For regression problems, we denote $\mathbf{y}$ to be true output vector, and $\hat{\mathbf{y}}$ to be the predicted vector by a model with learnt parameters θ. Let n be the number of data points and $\mathbf{y}^{(i)}$ to be the output of the i^{th} datapoint. We define the loss (objective) function: the `mean squared error(MSE)` to be:

$$E[\theta] = \frac{1}{n}\sum_i \|\mathbf{y} - \hat{\mathbf{y}}\|_2.$$

- For classification problems with C classes, let p and q be two distributions: true and predicted by the model with the parameters θ respectively. Recall multiclass

log cross-entropy between p and q to be:

$$E[\theta] = -\sum_{i}^{C} p_i \log q_i.$$

We usually use one-hot labeling and we apply *softmax* to maintain the predicted vector to have a probabilistic interpretation; hence we are able to use the statistical functions as loss functions.

2.3 Neural Networks

The term 'neural network' has its origins in attempts to find mathematical representations of the way in which information is processed in biological systems, [Bishop, 2006]. We can define neural networks as layer-structured directed graphs in which we call nodes neurons and these are linked through edges (weights) such that there are only connections between the previous layer and the following one.

Nowadays, the idea of neural networks and in general, deep learning (feedforward neural networks with at least two layers) is capturing all derived ideas, "many of which have been the subject of exaggerated claims regarding their biological plausibility", [Bishop, 2006]. Biological plausibility will be one of the criteria in assessing learning algorithms.

Although the format of neural networks can differ, usually the most standard (feedforward) neural networks model contains:

1. An input vector of dimension d. For multidimensional data, it is very common practise to flatten the input vectors.

2. Hidden layers (the ones which are neither input nor output layers), consisting of m_i neurons. There can be any natural number of hidden layers, and each of them can be composed of any positive natural number of neurons.

3. An output vector of dimension n. For regression analysis, $n = 1$. In literature, one-hot representation describes a binary vector $\mathbf{y}_n$ in which $(\mathbf{y}_n)_i = 1$ if and only if $\mathbf{y}_n$ belongs to the class C_i.

4. An activation function for each non-input layer; this function enables the model to catch nonlinear relationships. Examples of these can include: `tanh`, `sigmoid`, `linear`, `ReLu`, `Noisy ReLu`, `Leaky ReLu`.

5. Connections and weights associated with the layers.

It makes a lot of sense to talk about weights as matrices, since for each neighbouring pair of layers consisting of d_1 and d_2 neurons there are $d_1 \times d_2$ edges that can be grouped according to the neurons which they go into (those correspond precisely rows of the matrix) and the neurons that they are coming from (corresponds to columns of the matrix). Thus we can denote a matrix A, in which A_{ij} for $1 \leq i \leq d_2$ and $1 \leq j \leq d_1$ is the weight between j^{th} neuron in the previous layer and i^{th} neuron in the next layer.

For classification problems, one of a common technique is to set the number of neurons in the last layer to be equal to the number of classes and apply `softmax` function defined as:

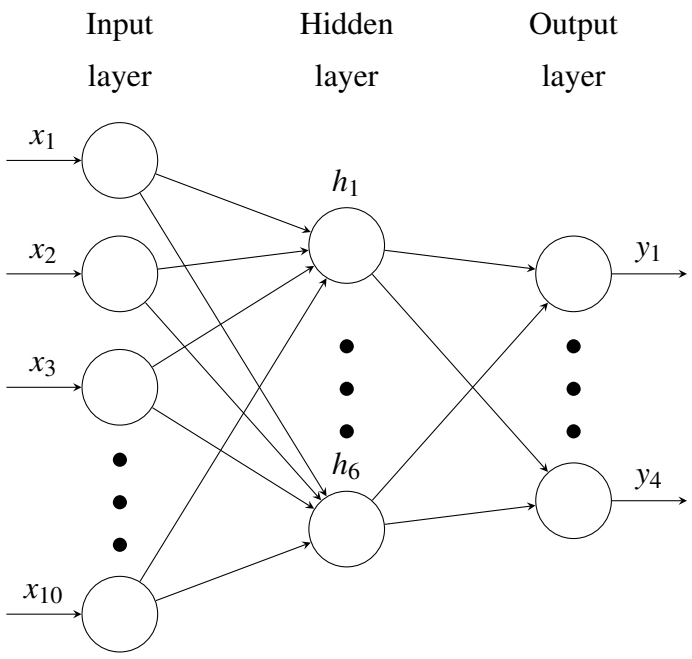
$$\sigma(\mathbf{y})_i = \frac{e^{y_i}}{\sum_{j=1}^{d} e^{y_j}} \text{ for } i = 1, \ldots, d \text{ and } \mathbf{y} = (y_1, \ldots, y_d) \in \mathbb{R}^d.$$

Let us notice that $\sigma : \mathbb{R}^d \to \mathbb{R}^d$ and $\sum_i \sigma(\mathbf{y})_i = 1$, which can have probabilistic interpretation.

To be more concrete and explicit, we will provide an example of such a model, which is built of:

1. An input vector $\mathbf{x}$ with has 10 dimension.
2. One hidden layer $\mathbf{h}$. It consists of 6 neurons.
3. An output vector $\mathbf{y}$ which has dimension 4.

It can be represented graphically as:



For the same reason for which we introduced constant terms in regression analysis, in neural networks the last units in each non-output layer tend to be defined as `biases`. It gives the model more freedom and helps to ameliorate the robustness.

Feedforward neural network are neural network in which there are no loops between the neurons. On contrary, `recurrent neural networks` relax this constraint - there are weights between the adjacent layers as well as weights coming in and out of the same layers. `Boltzmann machines` are special types of recurrent neural networks, in which there are connections between all layers. [Ackley et al., 1985] The `restricted Boltzmann machines` do not allow intralayer connections between hidden units. There is also a lot of attention around `Convolution Neural Networks` (CNN) which are a variant of feedforward network in which there are feature maps between convolution layers; they encode relevant information locally to retrieve the most characteristic features. This kind of architecture is particularly successful in image recognition and natural language processing.

For feedforward neural networks, on which we will focus the most, there is a very important theorem - `Universal approximation theorem`. It states that all feedforward neural networks can be approximated almost surely by another neural network with one single hidden layer (informally, we can say that all models in deep neural networks can be squished into wide models), [Nilsson, 1965].

Chapter 3

Overview of learning algorithms

The purpose of this chapter is to introduce the reader to a few families of learning algorithms used to train neural networks. These will be strongly based on the mathematical tools already reviewed in chapter 2.

In this chapter we present a few learning algorithms which we decided to be particularly exceptional or worth mentioning in this overview.

After considering over 40 different approaches, we selected those methods which:

- are flagship examples of broader a family of algorithms,
- at least in certain situations, achieve a competitive performance; usually we compare to the baseline - backpropagation with the gradient descent,
- are robust and can be applicable in many situations,
- have potential to be extended or combined with a different methods. The more compatible an algorithm is, the more likely it works with numerous architectures.

We will present and compare the algorithms from a theoretical point of view: we will discuss their limitations, their time and space complexity and the assumptions that they require. Additionally, we will point out the biggest advantages of each algorithm.

We will start, in section 3.1, with the predominant method for training neural networks which is gradient descent with a backpropagating error loss function across all layers. We will present the state of art techniques which facilitate this process very quickly.

We will discuss a family of approaches related to entropy and mutual information, which is used the current state of art method, HSIC, in section 3.2

We will also show some approaches which use quadratic programming in section 3.3. These methods are especially important since in certain situations with appropriate assumptions they outperform standard backpropagation.

In order to faithfully present the available methods for training neural networks we also briefly discuss some completely different approaches - derivative free methods in section 3.4, and quantum machine learning approaches in section 3.5.

The last section, section 3.6 is entirely based on our work. We will present the stochastic optimization algorithm which is a competitive replacement for gradient descent. It uses the technique of conjugate gradient.

3.1 Backpropagation and gradient descent

Backpropagation is an efficient way of implemententing the chain rule using dynamic programming in order to train neural networks. It was introduced along with the development of artificial neural networks, however the whole process was described 20 years later by Le Cun [LeCun, 1985] and by Rumelhart, [Rumelhart David E. et al., 1986]. Since then, the computation of gradient descent has become much more approachable and many computer scientists has been working actively on many optimizers, [Ruder, 2016].

To be precise, backpropagation is not a learning algorithm and usually what we mean when we say that a neural network is trained by backpropagation is that we apply the gradient descent method with the backpropagation technique.

Backpropagation and gradient descent are extremely popular learning algorithms because of their conceptual simplicity universality - it works for all types of feedforward neural networks, [LeCun et al., 1998]. It can also be extended to all types of recurrent neural networks and other more complex architectures.

3.1.1 Delta rule and how to compute it

All algebraic steps for exact computations are explained in [Bishop, 2006]. However, because the efficient gradient computation is essential not only for backpropagation with gradient descent, we recall its derivation.
Let us consider problem of learning the parameters (weights) in a neural network through backpropagation and steepest descent.
Let $\mathbf{x}$ be an input vector.

Let us define the loss function given as L_2 norm of the difference vector between the predictions $\mathbf{t}$ and the true labels $\mathbf{y}$, i.e. $E = \sum_j \frac{1}{2}(t_j - y_j)^2$.
Let us fix a non-input layer and consider the i^{th} neuron in that layer for which f is an activation function. Let us denote W to be a matrix of weights between the previous layer and the investigated one.
Let us consider a single connection between the j^{th} neuron in the previous layer and our neuron.

If we denote the sum of input values weighted by W as

$$g_j = \sum_i x_i W_{ji},$$

then we obtain the relation ([LeCun et al., 1998])

$$\frac{\partial E}{\partial W_{ji}} = -\left(t_j - y_j\right) g'(h_j) x_i.$$

Therefore our delta change is given by : $\Delta W_{ji} = \eta(t_j - y_j) f'(g_j) x_i$.

Thus $\delta_j = f'(g_j) \sum_i W_{ji} \delta_i$ for the j^{th} neuron in the output layer ([Bishop, 2006]).

In many modern computers, the computation architectures are based on GPU. Therefore, very often, it is extremely important to be able to represent an algorithm in matrix notation. For neural network with $m-1$ hidden layers, where each consisting of $a(i)$ units and the m^{th} layer as the output layer, we define diagonal square matrices D_i such that:

$$(D_i)_{kk} = (o_k^{(i)}(1 - o_k^{(i)})) \text{ for } 1 \leq i \leq m \text{ and } 1 \leq k \leq a(m),$$

where $o_k^{(i)}$ is stored output for the k^{th} neuron in the i^{th} layer.

We also define the error vector:

$$e_k = o_k^{(m)} - y_k = t_k - y_k \text{ for } 1 \leq k \leq a(m).$$

Then $\delta^m = D_m e$, and $\delta^i = D_i W_{i+1} \delta^{i+1}$ for $1 \leq i \leq m$.

3.1.2 Stochastic gradient descent and Mini-batch gradient descent

In machine learning, when working with gradient descent, our objective function is a function of many parameters. Very often we train the model with huge data sets. All of these imply that each iteration in the gradient descent method is very expensive. For instance, the asymptotic time complexity for training a neural network that has 4 layers

with respectively $a(1) = d$, $a(2)$, $a(3)$ and $a(4) = m$ neurons, with n training examples and t epochs is $O(nt(da(2) + a(2)a(3) + a(3)m))$.

It is worth noticing that rather than investigating a full data set we can randomly select one data point and quickly compute the derivative for that given data point. This does not guarantee that we move in the direction of the steepest slope, but assuming we repeat the procedure many times, statistically we are moving towards the minimum. This method is called `stochastic gradient descent` and it is significantly faster than standard full-batch gradient descent, [LeCun et al., 1998]. The biggest disadvantage of this approach is that it is not stable and with certain probability we may actually not make any progress or even drift away. Finally, because of fluctuations we may have difficulties in finding the appropriate time to stop our training.

`Mini-batch gradient descent` [LeCun et al., 1998] is somehow a combination between the previous two methods. Rather than a full data set or a single point we shuffle the data in order to retrieve random subsets of it. It turns out that this guarantees us to move towards the minimum in quicker manner, [Bishop, 2006].

We can express these three ideas in mathematical formulations. Let us denote θ_t the parameters of the model at iteration t and by J the objective function. By ∇J_i we denote the value of ∇J at the i^{th} point.

1. Full batch gradient descent $\theta_{t+1} := \theta_t - \eta \nabla J(\theta_t)$.

2. Single point stochastic gradient descent $\theta_{t+1} := \theta_t - \eta \nabla J_i(\theta_t)$.

3. Mini-batch gradient descent $\theta_{t+1} := \theta_t - \eta \sum_{i=1}^{n} \nabla J_i(\theta_t)/n$.

The problem with choosing a suitable learning rate in stochastic approaches is even more transparent than it was previously. For instance, given sparse data and the features with different variances, we would like to introduce a higher learning rate for features that seem to appear less often and, in general, it is not recommended to change all of them to the same rate, [Ruder, 2016].

It is widely accepted that stochastic gradient descent and mini-batch gradient descent represent the same concept and one of them is just a special case of the other (i.e. when the subset is a singleton). Therefore we will also refer to stochastic gradient descent as taking subsets consisting of n data points in order to determine the direction for each iteration. In the special case $n = 1$.

3.1.3 Optimizers: From Nesterov Accelerated Gradient up to Adam

In this paragraph we will address the problem of finding the appropriate learning rate. We will look at the problem more globally, by not considering each iteration independently, but by accumulating the history of directions. One of the assumptions that we silently put in the formulas in the section above is that η does not change. We will relax it and inspect how this can be advantageous when we modify the learning rate in each iteration. We will present only those optimizers which are widely popular in the field and discussed often, [Ruder, 2016].

1. `Momentum gradient`. We introduce a very natural efficiency which is keeping track of the previous direction steps. For the majority of iterations, the direction of two consecutive iterations is highly correlated, [LeCun et al., 1995]. Thus we introduce γ parameter, smaller than 1, so that $\gamma^n \to 0$. For each iteration, the new gradient direction would be a linear combination of the current gradient and the sum of geometric series of the previous directions.

   $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta_t)$
   $\theta_{t+1} = \theta_t - v_t$

2. `Nesterov accelerated gradient`. The previous approach may have an issue related with overshooting. This time we introduce upgraded momentum with look ahead. Not only do we store the sum of the geometric series of the previous directions but we optimize what would happen if we actually make a step. Therefore, we take the derivative with respect to the proposal step.

   $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$
   $\theta_{t+1} = \theta_t - v_t$

3. `AdaGrad` - Adaptive Gradient Algorithm. In this approach we finally will adjust the learning parameter for each iteration. We do that by accumulating the norms of the direction vectors, this will result in increasing sequence. Then, for each iteration we scale the learning rate by the accumulated sum of the norms so that the history will be less and less relevant to our updates. ε plays a smoothing role here to prevent division by 0.

   $G_t = \sum_{i=0}^{t} G_i^2$
   $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \nabla_\theta J(\theta_t)$

4. `Adadelta` - the smarter brother of Adagram. The major issue with AdaGrad is the fact that the decay of the the learning rate is very rapid (as the denominator grows after more and more epochs, learning rate becomes insignificant). The solution is to introduce a substitute for G_t which takes into consideration the decay over time (therefore we are slowing the increment):

   $E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$.
   We also define Root Mean Squared:
   $RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \varepsilon}$
   Combining, we express our update as:
   $\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$
   $\theta_{t+1} = \theta_t + \Delta\theta_t$

5. `RMSProp` - the brother of Adadelta. We maintain the exponentially modelled decay of the learning rate, however we focus on updating the learning rate for each step.

   $E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$
   $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$
   In his paper, Geoff Hinton is setting $\gamma = 0.9$ and $\nabla = 0.001$

6. `Adam` - Adaptive Moment Estimation. It has very similar concept to AdaGrad but we introduce idea of forgetting the irrelevant history. Adam keeps an exponentially decaying average of past gradients, which we can interpret as slowing down the momentum with time. Thus, we define:

   $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$
   $v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$

   Which yields:
   $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
   $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$

   Hence our parameter update in each iteration has the following form:
   $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$.

Equations are following the same notation as in [Ruder, 2016].

3.1.4 Issues

Despite the fact that backpropagation is the predominate approach in machine learning, there are numerous disadvantages to this method.

- Vanishing and exploding gradient. For very deep neural networks or for recurrent ones (RNNs, LSTMs, GRUs) the activation functions such as *sigmoid* or *tanh* tend to create the problem with gradient update which is very close to the boundary cases which makes learning highly slow or difficult.
- It only guarantees to find the local extrema.
- It has been proven that in nature the requirement of symmetric synaptic weights is relaxed. However, backpropagation is efficient as it assumes that the feedback is symmetric, [Lillicrap, 2016].
- Algorithm stability, which is to be discussed in next chapter.
- Issues with parallelization - it is a highly sequential algorithm, in which each layer must wait for the signal to propagate. With modern CPU architecture it has been more and more of a drawback.

However, there are however multiple solutions that try to mitigate these issues. For instance in order to reduce the significance of the vanishing/exploding gradient we use [Ioffe and Szegedy, 2015]:

- Adjusting appropriately mini-batches.
- Using Relu and Leaky relu activation functions.
- Autocoders, an unsupervised machine learning approach which learns the representation of data.
- Modifying the architecture, for instance by highway nets.

3.2 Hilbert-Schmidt Independence Criterion

3.2.1 Motivation for HSIC.

In section 3.1 we pointed out that one of the biggest disadvantages of the backpropagation is the vanishing and exploding gradient problem; this is targeted in the `Hilbert-Schmidt Independence Criterion (HSIC)` method. [Ma et al., 2019] Furthermore, symmetric weights are not needed in that model. Moreover, backpropagation has the update locking problem and in this sense HSIC is advantageous over backpropagation since it relaxes this condition. This implies that the HSIC method sounds very appealing and is worth inspection. Although its goal is not biological plausibility, it aims to update locking problem and weight transportation. [Lillicrap, 2016]

3.2.2 General setting for algorithms basing on information theory

HSIC suggests that rather than defining loss functions using cross-entropy and backpropagating the error function, we can define the learning objective to be the conditional entropy $H(Y|X)$. Due to Fano's inequality we know that conditional entropy directly affects the probability of classification error. [Ma et al., 2019]

Furthermore, we can use the fact that conditional entropy and mutual information $I(X,Y)$ are anticorrelated to express problems through that. This is advantagous since the training involves the representation of X and we are able to compute entropy of distributions. Therefore the method's objective is to maximize $I(X,Y)$ layer-wise.

In a standard information bottleneck approach [Tishby et al., 2000] we set the objective to be minimized as $I(X,T) - \beta I(T_i,Y)$ where T_i is the i^{th} layer's representation. Usually, for continuous distributions it is impossible to model mutual entropy ($I(X,Y) \to \infty$) however there are some approaches to overcome these obstacles such as discretization. [Amjad and Geiger, 2019]

HSIC stands out from similar approaches using information theory as it also indicates statistical independence between layers in a neural network. Hence, for all pairs of neighbouring layers in the network we aim at maximize the function HSIC between the layer activation and the desired output and minimize HSIC between that layer activation and the input. [Ma et al., 2019]

3.2.3 Mathematics behind HSIC

This method is an approximation of information bottleneck. The truth-ground mutual information, i.e. the information bottleneck objective is substituted by HSIC. In a mathematical way we replace the standard IB objective given by min $I(X,T_i) - \beta I(T_i,Y)$ by min nHSIC$(Z,X) - \beta$ nHSIC (Z,Y) for hidden representation Z. The architecture can be expanded in a natural way for many hidden layers implying multiple hidden representations Z_i.

The reason behind this is that HSIC can be calculated in quadratic time with respect to data points.

Let us define most common kernel, the `Gaussian Kernel`:
$k(x;y) \propto \exp(-\frac{1}{2}\|\mathbf{x}-\mathbf{y}\|/\sigma^2)$. It is called a kernel as we can define the feature map $\phi_x = k(x,\cdot)$.
Let us recall a standard `covariance` $Cov_{XY} = E_{YX}[XY])$. Let us extend this definition to the `cross-covariance` operator as $C_{XY} = \langle f, Cov_{XY} g\rangle = E_{YX}[f(X)g(Y)]$.
Following the identical relationships in classical probability, $P(\mathcal{Y},\mathcal{X}) = P(\mathcal{Y}|\mathcal{X})P(\mathcal{X})$, we deduce the embedding of a `conditional distribution operator` as $C_{YX} = C_{Y|X}C_{XX}$ and since the auto-covariance operator C_XX is invertible we can express the conditional covariance operator as $C_{Y|X} = C_{YX}C_{XX}^{-1}$.
The truth-ground HSIC as a function of joint probability distributions is $HSIC(\mathcal{P}_{XY}) = |C_{XY}|$

Finally, we focus on the last layer. Minimizing the information loss technique does indeed encapsulate all the information needed for classification through representation. However, an extra step is required in order to obtain the proper class labels from it. Usually this involves learning the correct permutation of classes.

Authors of the paper claim that setting the activation function to be RELU seems to work the best.

3.2.4 Consequences

Given the architecture and the experimental results we can point out many advantages of HSIC over standard backpropagtion with gradient descent.

- Layers can be learnt in parallel, backpropagation suffers from update locking problem. [Lillicrap, 2016]

- Because it is not a sequential procedure, HSIC does not have the exploding/vanishing gradient problem.
- There are no backwards steps thus it relaxes the constraint with symmetric weights.
- Boost in convergence, i.e. it is much more stable, especially for the unformatted approach.
- It is more expressive - it can capture some architectures with which backpropagation and gradient descent struggle, [Ma et al., 2019].

3.2.4.1 Issues

- Problems with finding other hyperparameters σ and β.
- A single sigma seems not to be enough to capture all the dependencies.
- Extra work is required on top of the last layer in order to find permutations. We discovered that there are some issues with it; for problems with numerous classes, the classifier will have problems with assigning the bijection.

3.3 Zhunxuan Wang's preReLU-TLRN

The following section is adapted from from the Zhunxuan Wang's dissertation, which is yet to be published. Therefore we present the method without any explicit citations.

In this section we will assume that our architecture consists of a two layer feed-forward neural network with one hidden and one output layer. In the first of them, we set the activation function to be ReLU. We require that input and hidden layers consist of the same number of neurons; hence for $\mathbf{x} \in \mathbb{R}^d$ this number would be d. We also assume that the last layer will consists of $m \geq d$ units, where $y \in \mathbb{R}^m$.

Furthermore, we state that the architecture follows the formula:

$$\mathbf{y} = B((A\mathbf{x})^+ + \mathbf{x}) \tag{3.1}$$

where $A \in \mathbb{R}^{d \times d}$ and $B \in \mathbb{R}^{m \times d}$. We require parameters in first layer to be non-negative, i.e. $\forall i \forall j A_{ij} \geq 0$.

A neural network satisfying such requirements is called a `preReLU-TLRN`.

3.3.1 Motivation

It is worth noticing that for a non-positive input vector, our equation (3.1) becomes $\mathbf{y} = B\mathbf{x}$ which is a system of linear equation.

Now, let's assume that input vector is non-negative, therefore (3.1) becomes $\mathbf{y} = B(A + \mathbf{1}_d)\mathbf{x}$. Once we know B, we can find its left pseudo-inverse C (hence $CB = \mathbf{1}_d$) using the system of linear equations. In order to find A we need to subtract the identity matrix.

Solving a system of linear equation of dimension d requires at least d linearly independent data points. This is due to rank-nullity theorem.

Since in general, we do not want to put any constraints on input space, we will approach the problem empirically and try to estimate A and B.

3.3.2 Risk minimization

Let us denote n to be the number of samples. We are interested in finding a matrix C such that $C\mathbf{y} = (A\mathbf{x})^+ + \mathbf{x}$.

Due to the nonlinearity caused by an activation function we artificially set a positive function estimator for it, i.e. we define $h : \mathbb{R}^d \to \mathbb{R}^d$ such that $h(\mathbf{x}) \approx (A\mathbf{x})^+ + \mathbf{x}$.

Therefore, for all data points $(\mathbf{x}, \mathbf{y})$ we are interested in minimizing the difference between vectors $C\mathbf{y}$ and $h(\mathbf{x}) + \mathbf{x}$. Using L2 regularization we intend on minimizing $E_{\mathbf{x}}[\|-C\mathbf{y} + h(\mathbf{x}) + \mathbf{x}\|_2^2]$.

Since we do not have access to ground true $E_{\mathbf{x}}$ we estimate it by $\hat{E}_{\mathbf{x}}[\|-C\mathbf{y} + h(\mathbf{x}) + \mathbf{x}\|^2] = \frac{1}{n}\sum_{i=1}^{n} \left\|-C\mathbf{x}^{(i)} + \mathbf{E}_i + \mathbf{x}^{(i)}\right\|^2$, where $\mathbf{E}_i = h(\mathbf{x}^{(i)})$.

Thus, we reduce problem to finding a matrix $\hat{C}$ which minimizes the equation above. This $\hat{C}$ will estimate C.

If $m = n$ then the matrix $\hat{B}$ such that $\hat{C}\hat{B} = \mathbf{1}_d$ is unique. In fact, it is $\hat{C}^{-1}$.

Otherwise, we need to scale $\hat{C}$ before setting $\hat{B}$ to be its right pseudo-inverse. We define a matrix $\hat{K} = \text{diag}\hat{k}$ where $\hat{k}_j = (\hat{C}\mathbf{y})_j / \mathbf{x}_j$ if it is constant, otherwise it is set to 1. Then we can easily find $\hat{B}$ by finding the right inverse of $\hat{K}\hat{C}$.

Let us define the f_2 problem:

minimize $f_2(E, C, \mathcal{S}) := \frac{1}{n}\sum_{i=1}^{n} \left\|-C\mathbf{x}^{(i)} + \mathbf{E}_i + \mathbf{x}^{(i)}\right\|^2$
subject to $E \succeq 0$
where $\mathcal{S} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$

In a similar manner we tackle solving the first layer. Let's firstly notice that
$h(\mathbf{x}^{(i)}) = (A\mathbf{x}^{(i)})^+ = (A\mathbf{x}^{(i)})^+ - (A\mathbf{x}^{(i)}) + (A\mathbf{x}^{(i)}) = (-A\mathbf{x}^{(i)})^+ + A\mathbf{x}^{(i)}$.
Thus combining it with (3.1) we obtain that our objective becomes to find the minimizers for $\frac{1}{n}\sum_{i=1}^{n}\left\|r(\mathbf{x}) + A\mathbf{x}^{(i)} - h^{(i)}\right\|^2$, where $r(\mathbf{x}) \approx (A\mathbf{x}^{(i)})^+$.
In matrix notation, by setting $\mathbf{R}_i = r(\mathbf{x})$, let us define the f_1 problem:

$$\begin{aligned}&\text{minimize } f_1(A,P,\mathcal{S}) := \tfrac{1}{n}\textstyle\sum_{i=1}^{n}\left\|\mathbf{R}_i + A\mathbf{x}^{(i)} - h^{(i)}\right\|^2\\ &\text{subject to } R \succeq 0\\ &\text{where } \mathcal{S} = \{(\mathbf{x}^{(i)}, \mathbf{h}^{(i)})\}_{i=1}^{n}\end{aligned}$$

As in the second layer, we need to scale the matrix $\hat{A}$, the optimal solution of the problem above.

For each dimension from 1 up to d we find the scaling factors independently as the linear coefficient (in the simple linear regression) for the data given by $\{h_j^{(i)}, (\hat{A}_j x^{(i)})_j\}$.

We can see the whole process encapsulated in Algorithm 1 on the next page.

3.3.3 Issues

The biggest issues of this approach are that it requires a two-layer architecture and it restricts the first layer weights to be positive. Although the are no restrictions on the input space, the dimensionality of the output space has to be at least the dimensionality of the input space. This can be to some extent fixed by pre-processing data, for instance by introducing dimensionality reduction.

This approach also requires the activation function to be ReLU, and even for similar functions (such as eLu or leaky ReLU) the process may not give the expected results. This was inspected by us and we encountered some inconsistency.

Finally, this approach requires extra work related to scaling each layer. Depending on different optimizers we may find different optimal results which yield different weight values.

3.3.4 Consequences

Despite the restrictions, the algorithm has few advantages.

- The clear advantage of this algorithm is the fact that solving two problems, namely f_1 and f_2 are standard convex problems. In certain cases the problems

Algorithm 1: Learning preReLU-TLRN by QP

Input: n pairs of ground truth input output training data

Data: $\mathcal{S} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and $\mathbf{y}^{(i)} \in \mathbb{R}^m$

Result: weights of neural network as matrices A and B

begin

```
/* Learning second layer                                          */
// Solve QP equation for second layer
```

$\hat{C}, \hat{E} :=$argmin$f_2(C, E, \mathcal{S})$ subject to $E \geq 0$

if $m == d$ **then**

$\hat{B} \leftarrow \hat{C}^{-1}$

else

// Estimate vector $\hat{\mathbf{k}}$ of scaling factors

for $j = 1, ..., d$ **do**

$\mathcal{S}_n \leftarrow (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) | x_j \leq 0\}$; // Points with positive d coordinate

$\mathcal{S}_p \leftarrow (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) | x_j \geq 0\}$; // Points with negative d coordinate

$\mathcal{S}_m \leftarrow$ bigger set out of $\mathcal{S}_n, \mathcal{S}_p$

$\hat{k}_j = 1$

if $(\hat{C}\mathbf{y})_j / x_j$ *is equal to some scalar* c *for all* $(\mathbf{x}, \mathbf{y})$ **then**

$\hat{k}_j \leftarrow 1/c$

end

$\hat{K} \leftarrow \text{diag}(\hat{\mathbf{k}})$

$\hat{B} \leftarrow$ right pseudo-inverse of $\hat{C}\hat{K}$

end

for $i = 1, ..., n$ **do**

$\hat{h}^{(i)} \leftarrow \hat{K}^{-1}(\mathbf{E}_i + \mathbf{x}^{(i)}) - \mathbf{x}^{(i)}$

end

end

Learning first layer

$\mathcal{S} \leftarrow \{(\mathbf{x}^{(i)}, \mathbf{h}^{(i)})\}_{i=1}^{n}$ // Solve QP equation for first layer

$\hat{A}, \hat{P} :=$argmin$f_1(A, P, \mathcal{S})$ subject to $P \geq 0$

for $j = 1, ..., d$ **do**

$\hat{k} \leftarrow$ the constant $(\hat{A}_j \mathbf{x})^+$

$\hat{\mathbf{A}}_j \leftarrow \hat{\mathbf{A}}_j / \hat{k}$

end

end

can be simplified to solving simple linear programming problems.

- The objective functions are quadratic, and the constraints are linear (and very often can be represented as sparse matrices). Due to many efficient optimizers, those subproblems are solvable quickly and guaranteed to be solvable in polynomial time.
- If we have access to only positive or negative data points, we can speed up the computation time.
- The structure of the algorithm is simple.
- It does not require much extra memory space.
- It has a very strong error convergence.
- It can be expanded to allow noisy models, given by $\mathbf{y} = B((A\mathbf{x})^{+} + \mathbf{x}) + \mathbf{z}$.

3.4 Derivative-free algorithms

The overview of learning algorithms would not be complete without examples of derivative-free approaches. There are some issues with the previously mentioned techniques: we may face the situation in which the derivative information is unavailable (for instance piece-wise definition of objective function), unreliable (for example noisy), or impractical (expensive to obtain). In this chapter we relax the condition that we are able to compute the derivative of the loss function. The family of such approaches is called `Derivative-free algorithms (DFO)`. In DFO, in every step, we aim to determine the next point to evaluate without computing the exact gradient.

We may distinguish different approaches; for instance we can do classification between direct and model-based, [Rios and Sahinidis, 2013]. Direct algorithms compute the values of the objective function directly (discriminative) and determine search directions, whereas model-based algorithms establish the search process by construction a model (generative).

We can also differentiate between local and global approaches. In global optimization we refine the search domain arbitrarily and the most famous representantative of such an approach is called BARON optimizer, [Sahinidis, 1996]. We classify algorithms further - as stochastic or deterministic, depending upon the random search steps. Finally we distinguish those algorithms which partition the search space and

those which do not (the next point to be evaluated may be located anywhere in the search space). [Rios and Sahinidis, 2013] As it is not central to our problem we will not focus on that.

The following questions that we face when designing any DFO model:

- Does an increase in the problem size affect performance and, if so, how much?
- Given a nonconvex program, what kind of solution (global, local near-global or local optimum) are we expected to obtain.
- Is it robust? Can this approach be applied a large fraction of problems and can it be combined with other solvers independently.

3.4.1 Broyden–Fletcher–Goldfarb–Shanno algorithm

One of the most important algorithms is `BFGS`. It also gives inspiration for the conjugate gradient method and the algorithm described in section 3.6.

BFGS is an iterative method for solving unconstrained optimization problems. It is a hill-climbing approach in which we seek for points for with gradient vanishes (necessary condition for differentiable functions). It uses second order Taylor expansion and rather than computing the exact Hessian matrix, the method approximates it. Although the method requires having access to first derivative, the key point is that we do not need to compute the second derivative despite the constant necessity for it in our calculations.

As always, we intend to minimize $f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^d$. Let us denote B to be the approximation for the Hessian matrix. We will base ourselves on Newton's equation: $B_k \mathbf{p}_k = -\nabla f(\mathbf{x}_k)$, and we will iteratively update B with:

$$B_{k+1}(\mathbf{x}_{k+1} - \mathbf{x}_k) = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k).$$

Let α_k be the optimal scalar in updating the direction towards $\mathbf{p}_k$, i.e. $\alpha_k = \operatorname{argmin} f(\mathbf{x}_k + \alpha \mathbf{p}_k)$.

The trick in this approach is to assume that the update of B is given by the sum of two matrices, which are symmetric and of rank 1. Therefore our standard Quasi-Newton update looks like $B_{k+1} = B_k + \alpha \mathbf{u}\mathbf{u}^\top + \beta \mathbf{v}\mathbf{v}^\top$. By choosing $\mathbf{u}$ to be the difference in gradient ($\mathbf{u}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$) and $\mathbf{v}$ to be the update vector ($\mathbf{v} = \alpha_k \mathbf{p}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$) we obtain:

$$B_{k+1} = B_k + \frac{\mathbf{u}_k \mathbf{u}_k^\mathrm{T}}{\mathbf{u}_k^\mathrm{T} \mathbf{v}_k} - \frac{B_k \mathbf{v}_k \mathbf{v}_k^\mathrm{T} B_k^\mathrm{T}}{\mathbf{v}_k^\mathrm{T} B_k \mathbf{v}_k}$$

where $\mathbf{u}_k$ and $\mathbf{v}_k$ indicate that those values change in every iteration.

We notice that the process is very simple and requires standard mathematical operations. The biggest disadvantage is that B_k does not necessarily converge to the true Hessian matrix although it correctly approximates it.

3.4.2 Limitations, Problems and Discussion

Given that we use more and more data in our machine learning problems we notice that the biggest challenge is related to dimensionality. For very high dimensional functions, DFO algorithms are not competitive with derivative based ones. The biggest disadvantage of evolutionary approaches is that the the complexity time is significantly larger than for the competitors. There are many parameters - in machine learning vocabulary, hyperparameters. Although there are various ways to get around it such as dimensionality reduction, the standard backpropagation works very well with autocoders. Global Optimizers suffer from the curse of dimensionality, even after appropriate preprocessing, [StackExchange, 2013].

Very often, for evolutionary algorithms to have similar performance on training data sets we need to overfit the models significantly and such models are not very robust.

The are some approaches in machine learning that use evolutionary algorithms as meta-heuristics for traditional backpropagation, [Rios and Sahinidis, 2013].

3.5 Quantum Machine Learning

In the light of the intense development of two very promising fields in computer science - machine learning and quantum informatics, one of the most natural question that arose was an idea of combining the two disciplines. Quantum-enhanced machine learning is a hybrid between quantum physics theory and machine learning analysis. We feel that the comparison would not be complete without mentioning the class of these approaches.

Promising discoveries and engineering solutions in the field of quantum computers suggest that there may also be potential for improving the performance of learning algorithms using quantum theory. [Bernstein and Vazirani, 1997] Despite much effort put into this field by the researchers from the biggest tech companies, until today, there are no technical papers published proving the existence and functionality of computers

with many qubits (the single unit information in quantum algorithms). Therefore the whole discussion is still at very theoretical level. Moreover, those which are claiming to be successful, have very simple functionalities and are very limited (because of low computational power related to physical constraints).

3.5.1 Quantum Algorithms

Let us briefly present the relevant basics in quantum mechanics and quantum algorithms theory.

In quantum mechanics, everything has a probabilistic representation; this is opposite to classical mechanics. Each state is in superposition and only measurement can reveal its actual properties. Some states are entangled with each other so that state of one of the qubit partially determines the state of another one. Almost all quantum algorithms are represented through diagram consisting of gates, which represent operations on qubits. Many of them are based on the Quantum Fourier Transform, Shor's algorithm and Grover's algorithm, [Shor, 1997]. Without delving into too much detail, there are some principles that differentiate the classical and quantum approaches. For instance, due to the no cloning theorem [Lindblad, 1999] we cannot save information in arrays as we tend to do in a standard programming and Holevo's theorem tells us about an upper bound for the amount of information (entropy) that we can deduce given a quantum state.

However, from complexity theory, there is a very important result which claims that $BPP \subset BQP$ which implies that all problems solvable by probabilistic Turing machine are also by quantum computers, [Bernstein and Vazirani, 1997].

The biggest advantages related to quantum computing are to do with probabilistic sampling. It is believed that quantum fields will be the most beneficial for probabilistic graphical models and for probabilistic learning algorithms, such as Monte Carlo Markov Chain. There were already some efforts made into that direction, for instance the Probably Approximately Correct method in context of quantum informatics, [Arunachalam and de Wolf, 2017]. This model was proposed as learning some fixed learning distributions for Boolean functions in classification problems. There is also some progress in quantum unsupervised learning, where quantum algorithms use similar concepts such as nearest neighbourhood and k-mean clustering. [Aïmeur et al., 2013]

There should be a distinction between comparing classical and quantum approaches

with respect to the number of queries and the running time. The quantum paradigm is equivalent in exact learning (membership queries) or in PAC learning, with respect to the number of queries (equivalence between polynomial number of queries), [Servedio and Gortler, 2004]. On the other hand, for computing time there are examples that are can be learnable in polynomial-time from quantum membership queries but not from classical queries, [Servedio, 2001].

For now, we do not have enough tools or knowledge in order to train the model or make a reasonable prediction, however once researchers tackle `Quantum Error Correction` issues and develop `category theory` as a tool for working on high-level languages with quantum computing, the future of the hybrid between machine learning and quantum algorithms is very promising.

3.6 Stochastic Conjugate Gradient

3.6.1 Family of Conjugate Gradient approaches

Conjugate gradient (CG) is a well known approach in mathematical optimization, we discussed it in subsection 2.1.4. In this section we will present a few variations of CG. Although they all find a sequence of conjugate direction vectors, they differ in way in which they optimize the scalar.

We emphasise that in machine learning we work with a vector of parameters, therefore θ is the flattened vector of the weights and biases. Hence we start with the a random initial guess θ_0. We assume that we are able to compute the gradient $\nabla\theta_i$ at any point (therefore we assume that our objective function is smooth).

In general, since the conjugate gradient method is an iterative procedure, our goal is to find a sequence of orthogonal directions $\mathbf{d}_i$ which yield a sequence of updated parameters: $\theta_{i+1} = \theta_i + \eta_i \mathbf{d}_i$.

In the very first iteration, $\mathbf{d}_0$ is simply the negative gradient $-\nabla\theta_0$, the direction with the steepest descent. We notice that the residual and the direction coincide.

Then, in the following iterations, we compute the direction:

$$\mathbf{d}_i = \mathbf{r}_i - \sum_{j<i} \frac{\mathbf{d}_j^\top \mathbf{A}\mathbf{r}_i}{\mathbf{d}_j^\top \mathbf{A}\mathbf{d}_j}\mathbf{d}_j.$$

However this requires a lot of computation in the multidimensional problems such as training large neural networks. Furthermore, because the problem is not quadratic,

the algorithm will not necessarily converge in $dim(\mathbf{X})$ iterations. Moreover, the algorithm can incorrectly converge due to the loss of conjugacy arrising from algebraic multiplicity of eigenvalues.

However, the main issue remains to find the Hessian matrix; this can be done in $O(N^3)$ calculation complexity and requires extra space $O(N^2)$, [Hestenes, 2012]. Therefore it is very common practise to approximate $H\mathbf{v}$ instead of computing H and then evaluating the product. [Hestenes, 2012] proposes the following formula for sufficiently small ε:

$$\mathbf{y}_i = H\mathbf{v} = (\nabla^2\theta_i)\mathbf{v} \approx \frac{\nabla(\theta_i + \varepsilon\mathbf{v}) - \nabla\theta_i}{\varepsilon}.$$

We note that this is exactly what we encountered in section 3.4 in BFGS algorithm. The time and space complexity is improved - $O(N^2)$ and $O(N)$ respectively, [Hestenes, 2012].

Therefore, we usually want to simplify the equation to the form:

$$\mathbf{d}_{i+1} = -\nabla\theta_{i+1} + \beta_{i+1}\mathbf{d}_i.$$

We notice that in each iteration we need to choose β_i adequately. There are three very common proposals for doing this, [Livieris and Pintelas, 2013]:

- `Fletcher-Reeves(FR)` update: $\beta_{i+1}^{FR} = \frac{\|\nabla\theta_{i+1}\|_2^2}{\|\nabla\theta_i\|_2^2}$
- `Polak-Ribiére(PR)` update: $\beta_{i+1}^{FR} = \frac{\nabla\theta_{i+1}^T(\nabla\theta_{i+1} - \nabla\theta_i)}{\|\nabla\theta_i\|_2^2}$
- `Hestenes-Stiefel(HS)` update: $\beta_{i+1}^{FR} = \frac{\nabla\theta_{i+1}^T(\nabla\theta_{i+1} - \nabla\theta_i)}{\mathbf{d}_i^T(\nabla\theta_{i+1} - \nabla\theta_i)}$.
- [Livieris and Pintelas, 2013] proposes adapted positive HS by taking $(\beta_{i+1}^{FR})^+$.

We notice that all computations can be done by dot product as the square of the norm of a vector it is just the dot product of the vector with itself.

It has been showed that all variations achieve very similar results, [Møller, 1990]. However [Møller, 1990] proposes an alternative, faster approach. Instead of the time-consuming line search, we use the trust-region approach. [Møller, 1990] proposes to add a regularization term to fix the Hessian to be positive definite:

$$\mathbf{y}_i = \nabla^2\theta_i \approx \frac{\nabla(\theta_i + \varepsilon\mathbf{p}_i) - \nabla\theta_i}{\varepsilon} + \lambda_i\mathbf{p}_i.$$

The method proposed by [Møller, 1990] results in a faster and equally accurate approach than its competitors; the experimental results are recalled in Figure 3.1.

TABLE 1
Results From the Parity Problem

	BP	SCG	CGL	BFGS
Par.	av./st. dev./fai.	av./st. dev./fai./sp.	av./st. dev./fai./sp.	av./st. dev./fai./sp.
3	3475/1020/0	413/306/1/8.4	1232/1383/1/2.8	736/473/0/4.7
4	16427/10185/1	1727/725/2/9.5	3320/3147/1/4.9	3004/3458/0/5.5
5	9864/5651/2	2131/1494/1/4.6	3682/2029/0/2.7	3246/2387/3/3.0
6	28671/20727/6	2811/1548/2/10.2	5435/6036/1/5.3	5601/3021/2/5.1
7	48878/38293/4	3801/3593/1/12.9	9903/12545/1/4.9	9343/10902/2/5.2
8	134130/64572/2	6206/3077/1/21.6	12518/14012/2/10.7	11426/8575/4/11.7
9	189453/53535/4	8105/5879/0/23.4	25855/22094/3/7.3	25748/24165/0/7.4

av. = Average Number of cu's. *st. dev.* = Standard Deviation. *fai.* = Number of Failures. *sp.* = Speed-Up Relative to BP.

Figure 3.1: Comparison between SCG, BP and the alternatives, [Møller, 1990].

3.6.2 Krylov subspace

A very important technique related with conjugate gradient is `Krylov subspace`. Given a matrix A and vector $\mathbf{u}$ we construct the subspace spanned by $\mathcal{K}_r(A,\mathbf{u}) = \{\mathbf{u}, A\mathbf{u}, A^2\mathbf{u}, A^3\mathbf{u}..., A^{r-1}\mathbf{u}\}$.

The most important properties of Krylov subspace, related with conjugate gradient, are:

- These sets are nested, i.e. with adding new vectors, we either increase or do not change the space.
- There exists R such that all vectors $\{A^r\mathbf{u}\}_{r=0}^{R}$ are linearly independent.
- The residual in each iteration, $\mathbf{r}_i = b - A\mathbf{x}_i$, is orthogonal to $\mathcal{K}_i$.

Furthermore, we define Krylov matrix to be the matrix with columns taken from Krylov subspace in the order. If the matrix has R (or less) columns then it is non-singular.

Krylov subspaces are extremely useful when it comes to working with conjugate gradients. We will not present the insights, however for the curious reader we refer to [Saad, 1981].

3.6.3 Stochastic Conjugate Gradient

We hereby declare that this algorithm was developed by ourselves, and any similarities between this approach and other methods imply independent discoveries.

CG method does not solve the problem with local extrema unless we are guaranteed that the hessian is positive definite. However we are able to approximate the solution in fewer iterations, each iteration is much more intense. The biggest drawbacks related

with CG were computational problems. Either in the standard procedure, we need to compute exact Hessian, or in FR, PR or HS update we need to do line search; in both cases a single iteration is expensive since we need to use full-batch descent.

For stochastic gradient descents there are many optimizers which significantly improve performance; this was discussed in subsection 3.1.3. However, for the conjugate approach, we are unable to use mini-batch, since we require the exact conjugation with respect to the Hessian. However, we should definitely use the fact that the conjugate gradient converges in relatively few steps, not only when it comes to possible early termination but also to infer about the type of the solution.

3.6.3.1 Bayesian statistics and conjugate gradient

In Bayesian approach, we assign a distribution to parameters rather than an exact values. In subsection 2.1.2, we provided with an example - Bayesian linear regression.

Given a mutually conjugate set $\{\mathbf{d}_i\}_i^N$ with respect of A we define the total direction vector to be:

$$\mathbf{d} = \sum_i^N \alpha_i \mathbf{d}_i.$$

However, any vector $\mathbf{z}$ we can represent in the basis $\{\mathbf{d}_i\}_i^N$ as

$$\mathbf{z} = \sum_i^N \gamma_i \mathbf{d}_i, \text{ where } \gamma_i = \frac{\mathbf{z}^T A \mathbf{d}_i}{\mathbf{d}_i^T A \mathbf{d}_i}.$$

If $\{\mathbf{d}_i\}_i^N$ is normalized, then the denominator will be 1, which simplifies further calculations. Let us take $\mathbf{x}_0$ to be an initial guess. After $\mathcal{M}$ iterations, our updated vector would be

$$\mathbf{x}_{\mathcal{M}} = \mathbf{x}_0 + \sum_{i=0}^{\mathcal{M}-1} \gamma_i \mathbf{d}_i.$$

We notice that by choosing appropriately the hyperparamter $\mathcal{M}$, even for $\mathcal{M} \ll dim(\mathcal{X})$, we should be able to achieve a significant improvement on the solution; this is closely related to the eigenvalue decomposition. There are various tools, for instance the Lanczos algorithm for finding the most extreme eigenvalues and this information can be useful in order to estimate $\mathcal{M}$.

Note that we do not require to have a good approximation of the extremum - we just intent on achieving a noteworthy progress towards the goal.

Let us now suppose that we know about two extrema. After the first few iterations, with certain confidence, we should be able to determine whether we are converging

to one of them, or neither of them. This idea can be extended further and it is an inspiration for algorithm 2. We propose the following algorithm.

1. We randomly select the initial positions of $\mathcal{N}$ trials. Initially, we believe that all trials have equal probability to be the most relevant.
2. For each trial, we apply first $\mathcal{M}$ conjugate gradient iterations.
3. We determine our initial belief distribution - it can be either uniform, dependent on the obtained values of the loss function, or dependent on the distance.
4. While the termination conditions are not met, such as the maximum number of iterations, the adequate convergence or obtaining the satisfying error:
    - Draw a trail according to our belief distribution.
    - Compute gradient of this trial.
    - Compute second order approximation - Hessian, for instance using Cholesky decomposition and determine using Sylvester's criteria if it is the positive definite. If not, regularization is needed. Alternatively, one can approximate the product of Hessian and a vector as in BFGS algorithm.
    - Compute the directional vector. Determine if it is pointing to the common goal for rest of the trials.
    - Determine whether to accept or reject the trial.
    - If the trial is accepted then we move all relevant trials towards the common goal.
    - Update the belief distribution. It will depend on both the individual loss value and the number of trials going towards the same goal.
    - If we do not want to continue the procedure for any reason, we may always terminate it and potentially finish the gradient descent with the most promising trial.
5. Consider all those trials, with high probability we will explore the most promising extrema and therefore we pick the best one accordingly to our beliefs.

Algorithm 2: Stochastic conjugate gradient method

Input: Hyperparameters: $\mathcal{N}$ - the number of trials; $\mathcal{M}$ - the number of conjugate gradient iterations for each trial; K_{max} - the maximum number of iterations; ε - the convergence threshold; $\mathcal{Z}$ - the search space for parameters θ

Data: Training data set $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{n}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and $\mathbf{y}^{(i)} \in \mathbb{R}^m$; we assume to have an access to $f(\mathbf{x}^{(i)})$ and the derivative of $f(\mathbf{x}^{(i)})$ with respects of parameters θ.

Result: weights θ of the NN. // Assume we know the shape of the NN

/* We denote $E(\theta)$ to be the loss function, $\nabla E(\theta)$ to be the gradient. */

/* We denote m_* to be the most promising trial, E_* to the most promising value of the loss function, z_* the most promising parameters. */

begin

  /* Assume no prior information about the trials. */

  for $n = 1, ..., \mathcal{N}$ **do**

    $\mathbf{z}_0^{(n)} \leftarrow Uniform(Z)$ // Sampling randomly initial guesses.

    if $E_* > E(\mathbf{z}_0^{(n)})$ **then** // Keep track of the most best trial.

      $m_* \leftarrow m$ and $E_* \leftarrow E(\mathbf{z}_i)$

    end

  end

  $\mathcal{D} \leftarrow Uniform(\mathcal{N})$

  for $n = 1, ..., \mathcal{N}$ **do** /* First stage of the algorithm */

    Perform the standard `Conjugate gradient`: we set the initial value to $\mathbf{z}^{(n)}$ and the number of iterations to $\mathcal{M}$

    Save the improvement of the solution: $\Delta E^{(n)} \leftarrow E(\mathbf{z}_{\mathcal{M}}^{(n)}) - E(\mathbf{z}_0^{(n)})$

    Save the total direction $\mathbf{d}^{(n)}$.

  end

  Update $\mathcal{D}$ accordingly to $\{\Delta E^{(i)}\}_{i=0}^{\mathcal{N}}$ - the results from conjugate gradient.

  while *not termination condition* **do** /* Max iteration or convergence. */

    $n_0 \leftarrow$ Sample accordingly to $\mathcal{D}$.

    $\mathbf{u} \leftarrow$ Perform one gradient update for trial n_0.

    if `Accept` $\mathbf{u}$ **then**

      foreach $n = 1, ..., \mathcal{N}$ **do** // Trials going towards the same goal

        If $E(\mathbf{z}_{-1}^{(n)} + \mathbf{u}) < E(\mathbf{z}_{-1}^{(n)})$ $\mathbf{z}_{-1}^{(n)} \leftarrow \mathbf{z}_{-1}^{(n)} + \eta\mathbf{u}$

      end

    else

      Reject sample, change the distribution $\mathcal{D}$ draw again.

    end

  end

end

We notice the special cases of the algorithm:

1. For $\mathcal{N} = 1$ and $\mathcal{M} = 0$ our algorithm is equivalent to the gradient descent.
2. For $\mathcal{N} = 1$ and $\mathcal{M} = dim(X)$ our algorithm is the conjugate gradient descent.

We immediately notice numerous advantages of the algorithm:

- Comparing with the standard gradient descent approach, we converge to a more promising local extremum, or at least (in the worst case), to one as good as in the gradient descent.
- It is a Bayesian approach.
- It combines partition and non-partition search space approaches. For the appropriate boundary selection, we are guaranteed limiting the unexplored set to be of measure 0. This is very important for convergence, however yet to be examined.
- Conjugate gradient can be replaced with other approaches if the decomposition of leading eigenvalues is flat or it is hard to compute. Gradient descent can be replaced with another method.
- It is very compatible with ensemble learning, for instance bagging.
- If, during the process of learning, we encounter that the difference between uniform distribution and our belief distributions is not statistically significant, then we can halt the process and finish it with the standard, well-known methods.
- Fine-tuning is possible. We just start by modifying the initial belief distribution.

On the other hand, the stochastic conjugate gradient has couple of disadvantages:

- Hyperparameter tuning is very relevant; the hyperparameters to be tuned in this approach are the number of trials, the number of iteration in the conjugate gradient, the maximum number of iterations, and the initial sampling distribution.
- The method is noticeable slower than the competitors.
- It assumes that the first iterations of conjugate gradient will lead significantly towards the extremum.
- It requires the access to the gradient.

Chapter 4

Comparison and Results

In chapter 3 we presented a few algorithms which can be used to train neural networks. Once we are familiar with them, we can apply them to solve problems; these includes convex, non-convex, smooth and non-smooth problems. The tasks will also be of a different nature - regressions, binary and multiclass classifications. We will apply the algorithms to various data sets, we will work on synthetic data as well as on multivariate numerical data sets tackling tasks from image recognition to natural language processing.

In this chapter we will present the results of applying the methods introduced in chapter 3. Since our discussion is theoretical, and the point is to compare the methods from a general point of view, rather than at a specific situation, we will try not generalize and guess the abstract patterns. We will highlight observations and indicate issues.

We will focus on five learning algorithms for neural networks:

1. Gradient descent with backpropagated error, presented in section 3.1, using various optimizers described in subsection 3.1.3.
2. HSIC [Ma et al., 2019] presented in section 3.2.
3. preReLU-TLRN by Zhunxuan Wang, presented in section 3.3.
4. BFGS, presented in section 3.4.
5. Stochastic conjugate gradient, the method proposed by us, which is explained in section 3.6.

Our objective for this chapter is to present empirical results of the above algorithms.

For each data set we will:

1. introduce characteristics of the data set, its objective, features and origin,
2. discuss the theoretical aspects,
3. talk about its limitations,
4. present various performances, and
5. draw relevant conclusions with an effort to back them up using theory.

In terms of code maintenance, we have decided to divide it by the tasks, for instance there are the directories in which we download, organize, pre-process the data and save it in an appropriate format. The are directories which are responsible for applying the algorithms and showing the results; the directories of scripts in which the code is sufficiently small to be captured within one file.

The Implementation of the HSIC and QP approaches can be found in the materials. We also attached the code shared by the authors which we have modified for our own purposes. For the HSIC approach, the plots and performance results are generated by the code shared by the authors due to the efficiency.

For the logistic reasons, the code is not ready to be run. We removed the data sets, We follow the submission guidelines provided by the Honours project coordinators.

For quadratic programming risk minimization problems we used CVXPY library, which provides very efficient convex solvers. An important part of experiments were run on cloud computers or on GPU provided by the university, [Tange, 2011]. We would like to thank Dr Shay B. Cohen for the facilitation and all help resolving our technical difficulties.

4.1 Synthetic data set - sine and cosine

We take advantage of two very simple smooth functions - *sine* and *cosine*, which are well-known to all. We present the synthetic data set which can be visualised as a table with columns (as features):

- real values x (target)
- sin values $\sin x$ (feature 1)
- cos values $\cos x$ (feature 2)

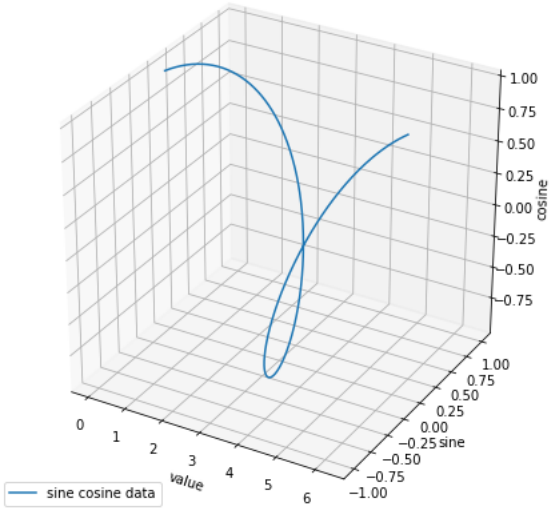


Figure 4.1: Visualization of the data set.

It is worth noticing that both $\sin x$ and $\cos x$ are periodic functions so we restrict our attention to $0 \leq x \leq 2\pi$ in order to avoid the situation that a model predicts x_0 instead of $x_0 + 2k\pi + \varepsilon$ for $k \in \mathbb{Z}$. We stress that the data is not linear nor quadratic.

4.1.1 Learning with backpropagation and SGD with Adam

Firstly, we will inspect neural networks with two hidden layers (each consisting of 20 neurons) with *tanh* and *sigmoid* activations and randomly initialized weights according to a normal distribution. We set the following hyperparameters: *number of epochs* is set to 150, *batch size per epoch* is set to 32. We will use the regular mean squared error as our loss function and test it against two standard metrics: the *MSE* and *MAE* techniques. We will apply stochastic gradient descent and the Adam optimizer.

After training the network for 20.44 seconds, the mean squared error on 51 test point samples for the model defined above was 0.24, however the maximum deviation was pretty high; this can be seen in Figure 4.1. The model fits the data pretty well, however there are major issues with training the model for values close to boundaries, i.e. values near 0 and 2π.

sin(x)	cos(x)	x	$\hat{x}$	sin($\hat{x}$)	cos($\hat{x}$)
0.1494	0.9888	0.15	2.7048	0.42297	-0.9061
-0.3739	0.9275	5.9	4.0189	-0.7690	-0.6392

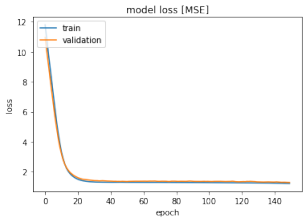


Table 4.1& Figure 4.2: Errors in two hidden layers NN trained with BP, tanh+sigmoid

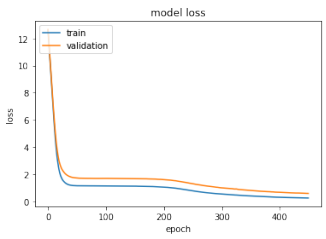


Figure 4.3: *Tanh&sigmoid*

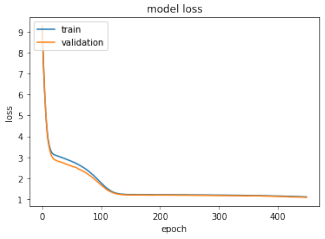


Figure 4.4: Twice *sigmoid*.

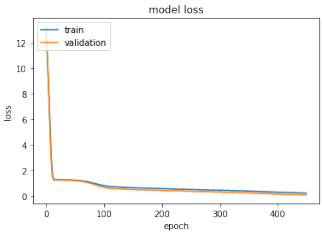


Figure 4.5: Twice *tanh*.

Changing activation functions to *sigmoid* in both hidden layers we obtain a mean squared error of 0.166184, so the improvement is significant. It is worth pointing out that the model struggles the most with the boundary points. The deviation is higher (the biggest difference between prediction and ground truth is equal to 3.05). Using *tanh* twice we can obtain a mean squared error of 0.051906 with 2.75 being the maximal square of the residual.

Finally, we should inspect the learning process. Visualizations above (Figure 4.3, Figure 4.4, Figure 4.5) of the model's loss functions during the training shows that depending on activation functions, the model is learning differently.

Let us now inspect one hidden layer version with 10 neurons and *tanh* activations and randomly initialized weights according to a normal distribution. The number of hidden neurons were optimized empirically. We set the following hyperparameters: the *number of epochs* is set to 150, and the *batch size per epoch* is set to 64. We will use the regular mean squared error as our loss function and test against two standard metrics: the *MSE* and the *MAE* techniques. We will apply stochastic gradient descent and the Adam optimizer.

For 10 neurons in the hidden layer, training lasts 17.18 seconds and the mean squared error for such model was 0.236, furthermore the deviation remains pretty high. The results are presented in Figure 4.2. The very interesting observation is that the error distribution is almost the same, and the algorithm struggles with the same problem, the boundary points.

sin(x)	cos(x)	x	$\hat{x}$	sin($\hat{x}$)	cos($\hat{x}$)
0.1494	0.9888	0.15	2.830	0.3063	-0.9519
-0.3739	0.9275	5.9	3.876	-0.6703	-0.742

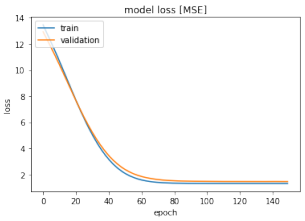


Table 4.2& Figure 4.6: Errors in one hidden layer NN trained with BP (tanh)

For a single layer neural network with *sigmoid* activation we obtained 0.3 as the mean squared error with higher variance. Furthermore, the learning curve decreased slower than previously. However, when working with networks with more than 4 hidden layers we saw a noticeable slowdown in learning process.

Finally, we comment on the model's hyperparameter, the number of hidden units. After conducting hypertuning analysis:

- For all proposals between 5 and 50 hidden units, the mean squared error oscillated between 0.2 and 0.35. However, we notice that the error is not highly dependent on the number of neurons; repeating the experiment yields different results.
- By running the experiments multiple times, we observe that the value of the error is mostly dependent on random guesses; this also depends on the training and test split rather than any particular fixed value.
- We can draw the conclusion that for such simple data sets the number of hidden units is not very relevant.

However, the activation function matters. Models with *tanh* seems to outperform those with *sigmoid* in this particular task.

From these relatively simple examples we can draw very important conclusions:

- Activation functions play a very important role and they should always be suitably chosen.
- In edge cases, neural networks need more data points in order to learn more unusual dependencies.
- Hypertuning neural networks is important, however it is very likely that similar models (of the same family) will struggle with same problems.

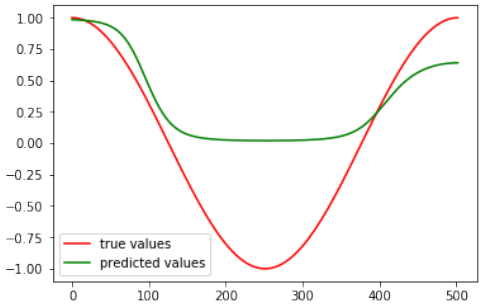


Figure 4.7: Conjugate gradient descent.

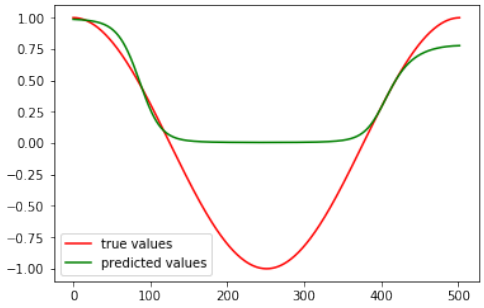


Figure 4.8: Stochastic conjugate gradient.

Figure 4.9: Problems with local extrema in conjugate gradient methods.

- Despite the theorem that every feedforward network with a single hidden layer containing a finite number of neurons can approximate any complex deep neural network model, for this data set the gap is noticeable. In practice we would require very good hyperparameter selection in order to achieve similar results. This can be noticed already, on relatively small networks. This issues scales appropriately for larger architectures, such as CNNs, [Bishop, 2006].

4.1.2 Conjugate gradient descent

When working with this data set we will use Fletcher-Reeves (FR) updates. We assume that the intial guesses are sampled from a Gaussian distribution. We set the convergence threshold to 0.02.

Conjugate gradient descent results in a very similar performance as backpropagation with gradient descent. After a slightly longer training, on average 20% longer, MSE also oscillates between 0.15 and 0.25 for the data set, depending on the initial guesses.

Although backpropagation with gradient and conjugate descents both suffer from the same issue - local extrema, there are significant differences in error distribution. Backpropagation is extremely inaccurate when it comes to the boundary cases, whereas conjugate gradient descent requires more time to capture the dependency, Figure 4.7.

4.1.2.1 Stochastic conjugate gradient

If stochastic conjugate gradient is appropriately tuned, then in the same time, it is able to improve the performance of the traditional conjugate gradient descent method, Figure 4.8. Typically, the grid search is much more demanding.

Figure 4.10: Feature correlations.

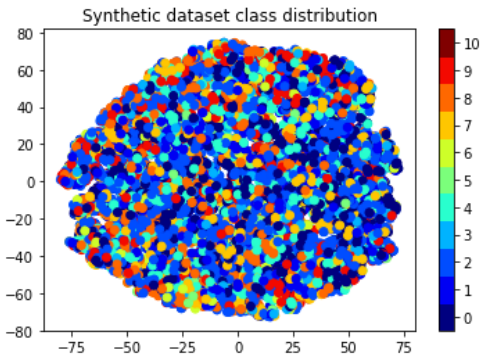


Figure 4.11: tSNE plot of the data.

Figure 4.12: Synthetic data set.

With a simple data set, such as the one that we provided, one can notice that the actual outcome of the algorithm in most cases is equivalent to a standard conjugate gradient descent. Both approaches usually lead to the same neighbourhoods; this is caused by the structure of the local extrema. In order to gain any improvement, after $\mathcal{M}$ conjugate gradient iterations, a trial has to lie nearby between two local minima, and has to be drifted by the other trials. In practise, this situation does not happen very often, at least when working with toy examples.

4.1.3 ReLU and quadratic programming

For such data sets, in order to apply the quadratic risk minimization algorithm, we would have to assume that the architecture follows Equation 3.1. Unfortunately, for the majority of regression problems, the dimensionality of the output space is smaller than the dimensionality of the input space. Therefore, this constraint seems to be very limiting.

4.2 Synthetic data set - Different random distributions as features

Now we inspect a classification problem, in which the training set consists of $10,000$ data points - each of them represented as a vector with 10 features. All of them are numerical values. The values for each feature are generated independently, Figure 4.10, and identically according to the distributions:

1. *Beta* distribution with parameters $\alpha = 2$ and 3.
2. *Uniform* distribution with parameters $a = -1$ and $b = 5$.
3. *Poisson* distribution with parameter $\lambda = 2$.
4. *Exponential* distribution with parameter $\lambda = 5$.
5. *Normal* distribution with parameters $\mu = 1$ and $\sigma^2 = 2$.
6. *Uniform* distribution with parameters $a = -1$ and $b = 1$.
7. *Negative Binomial* distribution with parameters $r = 15$ and $p = 0.4$.
8. *Normal* distribution with parameters $\mu = 0$ and $\sigma^2 = 1$.
9. *Poisson* distribution with parameter $\lambda = 4$.
10. *Uniform* distribution with parameters $a = 1$ and $b = 20$.

The choice of such features can be justified as following:

- Well-known distributions describe lots of phenomena occurring in nature or our everyday life.
- It is very common in machine learning to pre-process the data so that all features are statistically non-correlated, or at least the null-hypothesis is satisfied.
- If a model is capable of making accurate predictions on such a data set, other data sets whose features can be approximated with standard distributions should also be learnable for this model.

Therefore models should be able to adapt to such situations. The labels were created in a decision tree manner, depending on the deviations between the expectations of distributions and the randomly generated values for data points and non-linearity was introduced. Thus, data points within the same class share a few common features. However, the differences are very subtle, Figure 4.11.

The class distribution is not evenly balanced, the frequency of classes is given by the following:

$$\{0:1809, 1:1036, 2:2488, 3:344, 4:1308, 5:120, 6:216, 7:804, 8:1022, 9:853\}.$$

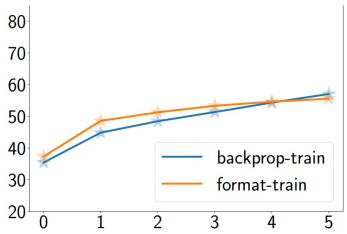


Figure 4.13: Synthetic data, 2 layer NN, we notice very similar performance from both algorithms.

This will be crucial in observing how learning algorithms behave when we are not able to provide much data about some classes or we have a priori knowledge about the class distributions.

Moreover, class 0 was created for these data points which did not underline any specific features. In machine learning, we sometimes face up the situations in which we have noisy labels or we are not aware of all classes a priori but we still want to perform classification, not clustering.

To see that the classification is not trivial one can examine its tSNE plot, Figure 4.11. Since this data set is much more complex than the previous one, we must allow the model to have more parameters. The clustering is highly overlapping, so we may expect that the classification errors will be very high for some pairs of classes. On the other hand, we expect that feature uncorrelatedness should improve the classifiers performance.

4.2.1 Training with BP and HSIC

We set up a neural network with three layers - the first hidden layer has 20 neurons and $tanh$ as the activation function, the second hidden layer has 20 neurons and $ReLU$ activation. The output layer is scaled by $softmax$, and consists of 10 neurons corresponding to the number of classes as we use one-hot encoding.

We performed with different activation functions and we claim that this combination is very promising and achieves relatively very good accuracy. Examining Figure 4.15 we notice that 0.58 seems to be the accuracy obtained by backpropagation on the training data set, however on the test data set it jumped to 0.65. Furthermore, comparing the loss on training and validation data sets the classifier seems not to overfit the data, but it struggles with certain pairs of classes. After inspection, we notice that there are pairs of classes (for instance class 1 and class 4) which are separable, however

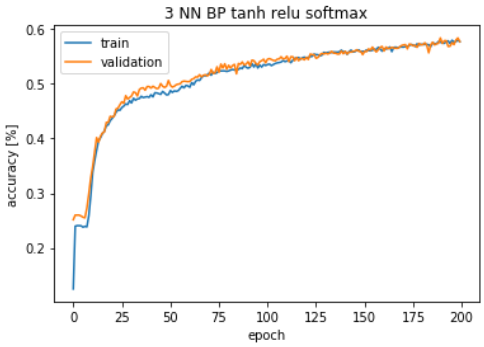


Figure 4.14: Synthetic data set, backpropagation, training and validation accuracy.

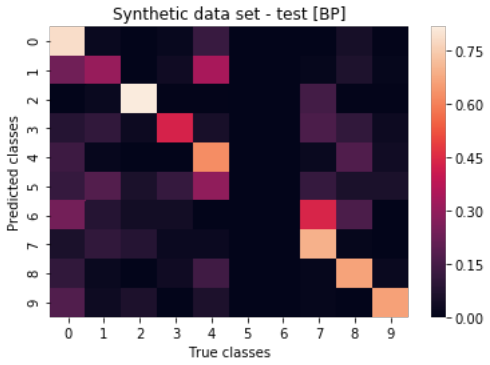


Figure 4.15: Synthetic data set, backpropagation, test data, error matrix.

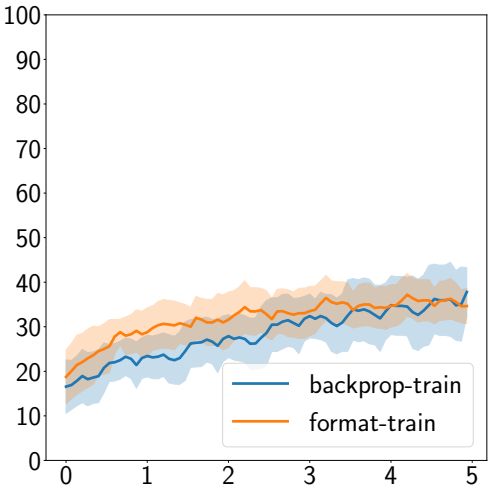


Figure 4.16: Learning curves for training a 50 layer NN.

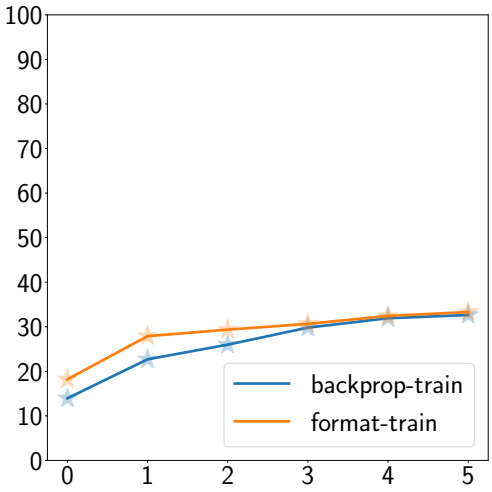


Figure 4.17: Accuracy for testing data, 50 layer NN.

Figure 4.18: Synthetic data set - HSIC.

there are pairs of classes (for instance classes 3 and 7) which are closer to each other. However, it is hard to see overall patterns in these misclassifications.

Figure 4.13 indicates clearly that the learning curve for backpropagation and format training are very similar, with respect to epochs.

Figures 4.19-4.25 presents a study ablation carried out by us for HSIC.

ReLU and *elu* seem to be the most appropriate activation functions for the HSIC method, which confirms the authors' claim.

Figure 4.25 points out that it is advantageous not to choose too many parameters as the models requires more time to train, and the extra information captured is not significant. At least for such simple data sets like this one.

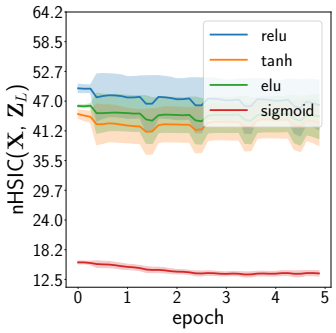


Figure 4.19: HSIC: Synthetic data set, HSIC, activations, XZ values

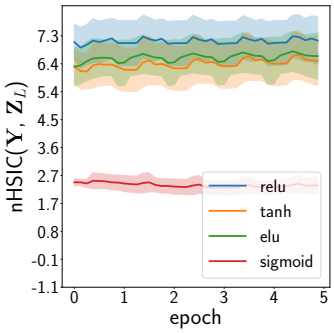


Figure 4.20: HSIC: Synthetic data set, HSIC, activations, YZ values.

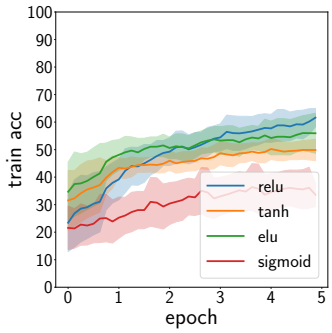


Figure 4.21: Synthetic data set, HSIC, activations, accuracy.

Figure 4.22: Inspecting different activation functions - HSIC.

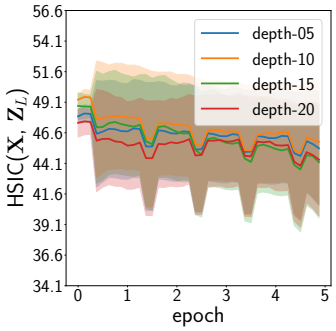


Figure 4.23: HSIC: XZ.

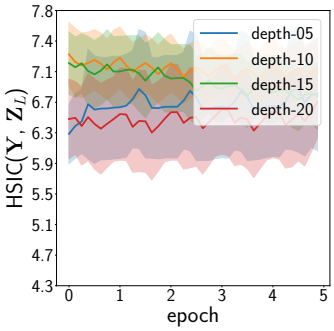


Figure 4.24: HSIC: YZ.

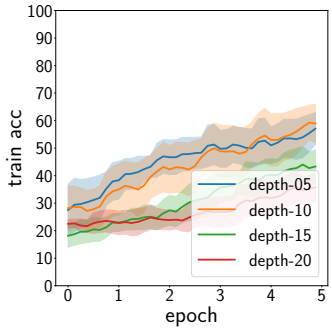


Figure 4.25: Accuracy.

Figure 4.26: Synthetic data set, HSIC - Inspecting depth of models.

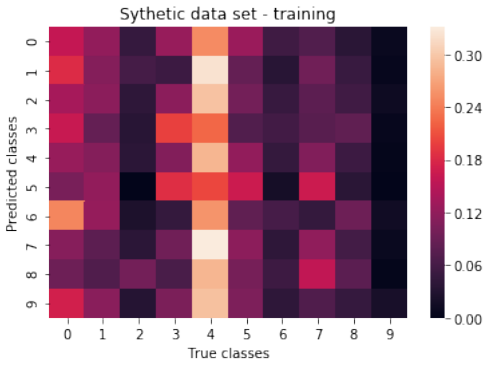


Figure 4.27: Synthetic data set, inaccuracy matrix for synthetic in QP.

4.2.2 Training preReLU-TLRN by QP.

Equation 3.1 requires the output space to be at least the same dimensionality as the input space. We use one-hot encoding so that for 10 classes this number is equal to the number of features.

The results can be seen in Figure 4.27. The training accuracy is around 15% which proves that the relation cannot be captured in a such demanding architecture.

4.3 Wine data set

We present a relatively simple, small and real data set representing the results of a chemical analysis of wines from Italy cultivated by three different methods but in the same area. Each feature (13 of them) corresponds to the constituents contained in each of the type of wine. [Forina et al., 2012]

We claim that pre-processing is not needed in this case and all models should be able to learn the characteristics even with a small number of samples. We present a plot which shows the correlation between the features in the data set (Figure 4.28). Here, the first column is class indicator (0 or 1 or 2). From the previous plot, we draw the conclusion that few features in this case are dominant - for instance the correlation between class and features 11,12 or 13 is very negative.

Training the neural network with backpropagation (with optimizer rmsprop) yields a high accuracy 0.944 with very fast convergence (both asymptotic and real time). However due to the size of the data, for batch sizes (roughly $\frac{1}{6}$ of the full data) we encounter huge oscillations.

Conjugate gradient descent methods obtain almost identical performance. We could not find the appropriate hyperparameters for the stochastic conjugate gradient

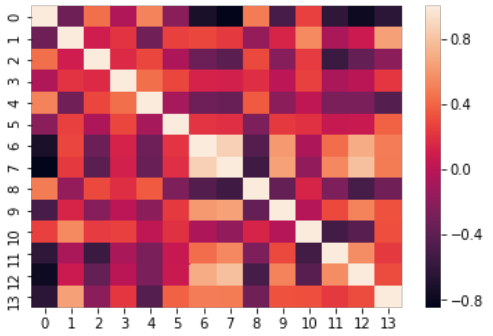


Figure 4.28: Wine data set - correlations.

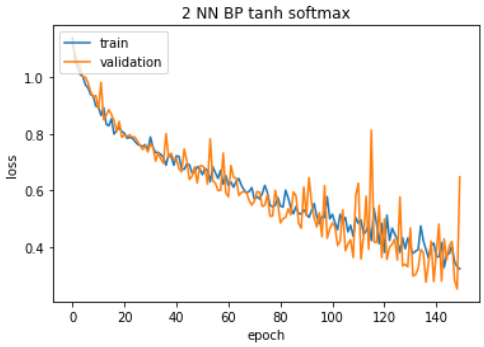


Figure 4.29: Training BP - oscillations.

algorithm which could improve the performance.

It is worth noticing that even when blindly set the number of trials to 3, the real time needed the algorithm increases by a factor of around 3.

4.4 Amazon customer reviews

In this section we would like to focus on one of the tasks in natural language processing - sentiment analysis. We will use the data set of the reviews from Amazon customers, [Mudambi and Schuff, 2010].

Although there are various architectures that are more suitable for the tasks such as recurrent neural networks, we will again use only feedforward architecture. We apply a suitable pre-processing (word embedding, TF-IDF features extraction).

We set the number of relevant words in vectors embedding for TF-IDF to 500. We use a 5 layer neural network architecture with 5 one-hot encoding since we predict the rating on a scale of 1-5.

The results of applying a standard backpropation with gradient descent and HSIC can be seen in Figure 4.30. We notice that for this task, both algorithms have almost identical performance.

4.5 Microsoft COCO

COCO is an image data set, [Lin et al., 2014]. Although it can be the underlying data set for many tasks in machine learning, we will use it for object detection. Given time constraints, we are not able to provide any detailed outcomes from the experiments, but the preliminary results are attached in the project materials.

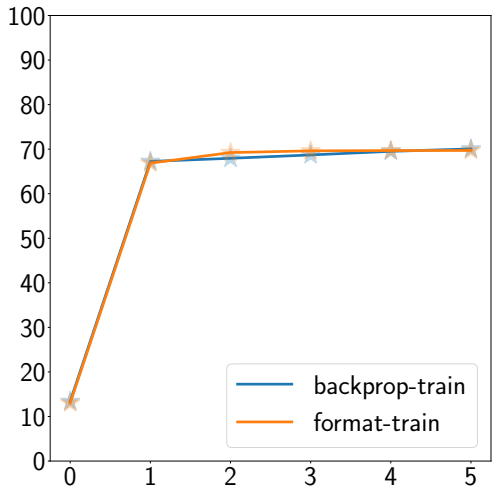


Figure 4.30: Amazon data set, training accuracy.

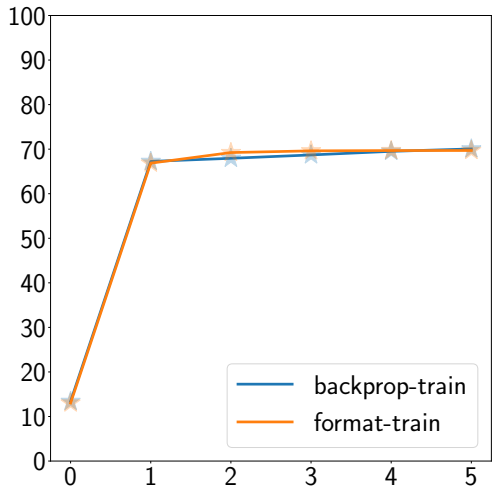


Figure 4.31: Amazon data set, training accuracy.

COCO is a significantly bigger data set than than the the previous ones. The state of art techniques (using CNNs) obtain even up to 70% accuracy. For feedforward networks, all algorithms: gradient descent, conjugate gradient descent and HSIC obtain around 45% accuracy. Stochastic gradient descent is significantly slow when the number of trials is not 1.

4.6 Chess Data

We present the data set of chess game endings, [Bain and Muggleton, 1994]. It consists of the positions of the black king, white king and white rook. It assumes that it is black's turn. The class for each situation (data point) is either draw or number in which white can give checkmate to black (a non-negative integer with 16 as upper bound). The pre-processed class distribution can be seen in Figure 4.32. We immediately notice that in this case the boundary classes, i.e. draw and initial check mate should be easy to detect.

In this case, the algorithms have to adopt to very abstract learning. They are required to understand several characteristics - for instance the symmetry of the chequerboard or the significance of the difference between draw and other classes. Moreover, the misclassification of a data point belonging to checkmate in 4 moves to the class with 5 moves should be less significant than to one move. That is why we will also consider regression. Finally, the algorithm should have a notion of chess rules. [Bain, 1992]

In this data set, backpropagation seems to be working slightly better than HSIC;

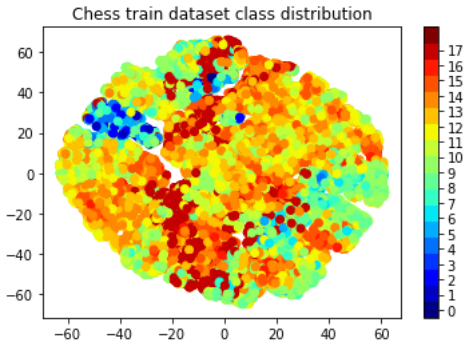


Figure 4.32: Chess data set, class distribution

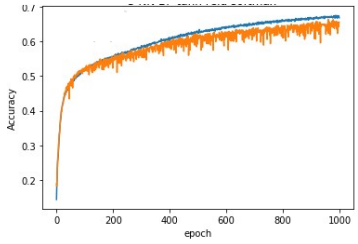


Figure 4.33: CG and BP, converging to the same local minimum

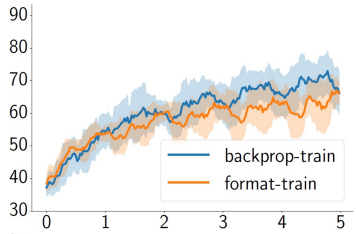


Figure 4.34: HSIC and BP

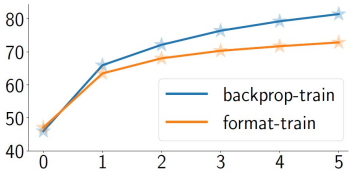


Figure 4.35: HSIC and BP

Figure 4.36: Chess data set trained by CG, BP and HSIC.

the relevant plots are presented in Figure 4.36.

Combining these observations with our results from the sine-cosine data set, we conclude that in the situations where the boundary cases are characteristic, backpropagation with gradient descent outperforms HSIC.

After conducting the experiments multiple times, we notice that conjugate gradient descent and stochastic gradient descent both converge to the same local mininima. The real time needed for training the chess data set with conjugate gradient is around 25% longer. This highly depends on the line search. With 15% acceleration, the accuracy drops by around 8%.

Chapter 5

Conclusions

In this dissertation, we addressed the problem of training neural networks.

We provided an overview of neural networks and the way how that they are trained. We presented a few methods that find the optimal parameters for neural networks. We pointed out the strengths and the weaknesses of each approach. Although we focused on the feedforward neural network architecture, the methods discussed can be easily extended to recurrent neural networks, including the most popular - the convolutional neural networks.

Despite the fact that gradient descent (with error backpropagation) seems to be a naive procedure, there are various optimizers which make the learning efficient. Furthermore, backpropagation with gradient descent only requires a differentiable loss function, and hence it is universal.

There are numerous algorithms that can learn the parameters of neural networks equally good and fast learn the parameters of neural networks as backpropagation with gradient descent.

Our last contribution is the stochastic conjugate gradient algorithm, which aims to optimize the extrema that descent algorithms return.

5.1 Future work

Due to time constraints, we did not manage to finish everything that we planned. Followingly, we outline directions for further work would be done towards the following directions.

5.1.1 Difference target propagation

In the second half of April 2020, just a couple days before the deadline for this report, a very promising article was published, [Lillicrap et al., 2020]. The paper gives an excellent overview of learning algorithms from a biological point of view, including a comparison between the structure of the feedback. Although it is still not certain whether backpropagation is the way in which a signal is processed in nature, the authors suggest that it is likely that the structure is somehow similar to the backpropagation algorithm.

`Difference target propagation` is a very promising algorithm presented in [Lillicrap et al., 2020]. It stands out from the rest of the approaches - it locally approximates the feedback signals, which causes the relaxation of the necessity of passing the error feedback sequentially. This results in driving effective learning.

The architecture is based on encoders, neural networks which encrypt and decrypt the information from the input. Therefore, we assume that for f, feedforward activation, we have an estimation of its inverse, g, the backward pass. Thus we intend on minimizing the difference $g \circ f - \mathbf{1}$.

In backpropagation, we would backpropagate that error futher. Here instead, we find the value of the inverse function applied to target value. For all hidden layers, we can locally compute the difference, which is the error.

In reality, we do not have a perfect inverse function, but only approximation. Firstly, we compute the inaccuracy vector, the difference between the neuron and the inverse applied to what we have got in the forward pass. Then, we compute the value of the inverse applied to the output target, and then we shift it by the inaccuracy vector to obtain the hidden target. This can be captured in the formula

$$\tilde{\mathbf{h}}_l = \mathbf{h}_l - \hat{\mathbf{h}}_l + g_{l+1}(\tilde{\mathbf{h}}_{l+1}).$$

Our error is computed locally, and does not need to be passed backward:

$$\mathbf{e}_l = \tilde{\mathbf{h}}_l - \mathbf{h}_l.$$

The algorithm looks very promising, and the whole process is captured on the diagram, *Figure* 5.1.

Difference target propagation would be definitely next algorithm that we would like to test and it seems to be highly biologically plausible. Moreover, it should avoid problems with vanishing (or exploding) gradient.

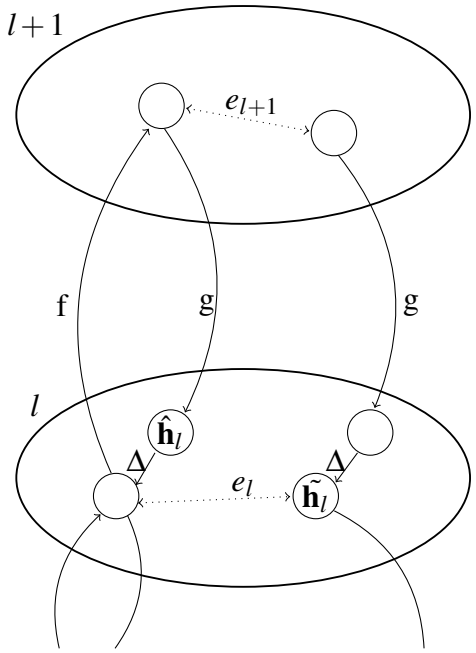


Figure 5.1: Visualization of the difference target propagation algorithm. We notice that the error is computed locally. g is the estimation of the inverse of f, the feedforward pass. The output target is the right neuron in the top layer, the hidden target is denoted as $\tilde{\mathbf{h}}_l$ and f is pointing to the feedforward activations neurons. Δ is the inaccuracy vector.

5.1.2 More work on the experiments

Given the time constraints, we had to somehow find the balance between exploration and exploitation of the topic, between researching for different approaches and examining them. We dived deep into the exploration - examining a lot of different approaches, based the various mathematical techniques. The following work included both understanding the mechanisms behind the algorithms as well as inspection of the results.

We believe that the exploration part was done diligently, however we could balance it more towards the work on the experiments. Granted more time, we would focus more on the experiments, rather than research, re-implementing or processing the data.

5.1.3 Quantum machine learning

Due to the logistic reasons, the approaches related with quantum informatics were not appropriately tested. We are looking forward to be able to implement and conduct the experiments on quantum computers. Although the section 3.5 looks pretty futuristic, there is much potential in the field and we believe that the discussion about QML paradigms is worth emphasizing now.

5.1.4 Stochastic conjugate gradient

We have spent a large amount of the time on the design of stochastic conjugate gradient method, discussed in section 3.6. We had underestimated the time needed for the development, and in the future we would spend more time on testing this approach.

5.2 Learning outcomes

Working on this project was very impactful in our university career, particularly our final year of studies. It allowed us to dive into our favorite field of computer science, machine learning, and moreover it had many links to other areas of computer science which we were simultaneously studying at university.

This project was a great chance to put our theoretical knowledge of machine learning into practice. Further, the coding and planning which went into this project significantly improved our software engineering and managerial skills, both of which are highly profitable when it comes to searching for future internships and jobs.

Finally, this project confirmed our desire to pursue graduate studies; it was an excellent way to get a feel for what research is really like and it even provided us with a great outlook on potential research directions.

Bibliography

[Ackley et al., 1985] Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169.

[Aïmeur et al., 2013] Aïmeur, E., Brassard, G., and Gambs, S. (2013). Quantum speed-up for unsupervised learning. *Machine Learning*, 90(2):261–287.

[Amjad and Geiger, 2019] Amjad, R. A. and Geiger, B. C. (2019). Learning representations for neural network-based classification using the information bottleneck principle. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[Arunachalam and de Wolf, 2017] Arunachalam, S. and de Wolf, R. (2017). A survey of quantum learning theory.

[Bain, 1992] Bain, M. (1992). Learning optimal chess strategies. In Muggleton, S., editor, *ILP 92: Proc. Intl. Workshop on Inductive Logic Programming*, volume ICOT TM-1182. Institute for New Generation Computer Technology, Tokyo, Japan.

[Bain and Muggleton, 1994] Bain, M. and Muggleton, S. (1994). Learning optimal chess strategies. *Machine intelligence*, pages 291–309.

[Bernstein and Vazirani, 1997] Bernstein, E. and Vazirani, U. (1997). Quantum complexity theory. *SIAM Journal on computing*, 26(5):1411–1473.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

[Cauchy, 1847] Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. pages 536–538. Comp. Rend. Sci. Paris.

[Forina et al., 2012] Forina, M., Aeberhard, S., and Leardi, R. (2012). Wine data set. *PARVUS, Via Brigata Salerno, htps://archive. ics. uci. edu/ml/datasets/Wine*.

[Gray, 2013] Gray, R. M. (2013). Entropy and information theory. pages 35–42. Stanford.

[Hestenes, 2012] Hestenes, M. R. (2012). *Conjugate direction methods in optimization*, volume 12. Springer Science & Business Media.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

[Kukačka et al., 2017] Kukačka, J., Golkov, V., and Cremers, D. (2017). Regularization for deep learning: A taxonomy.

[LeCun, 1985] LeCun, Y. (1985). Une procédure d'apprentissage pour réseau a seuil asymmetrique (a Learning Scheme for Asymmetric Threshold Networks). In *Proceedings of Cognitiva 85*, pages 599–604, Paris, France.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Orr, G., and Muller, K. (1998). Efficient BackProp. In Orr, G. and K., M., editors, *Neural Networks: Tricks of the trade*. Springer.

[LeCun et al., 1995] LeCun, Y., Jackel, L. D., Bottou, L., Cortes, C., Denker, J. S., Drucker, H., Guyon, I., Muller, U. A., Sackinger, E., Simard, P., and Vapnik, V. (1995). Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition. In Oh, J. H., Kwon, C., and Cho, S., editors, *Neural Networks: The Statistical Mechanics Perspective*, pages 261–276. World Scientific.

[Lillicrap, 2016] Lillicrap, T. P. (2016). Random synaptic feedback weights support error backpropagation for deep learning. *Nature Communications*.

[Lillicrap et al., 2020] Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J., and Hinton, G. (2020). Backpropagation and the brain. *Nature Reviews Neuroscience*, pages 1–12.

[Lin et al., 2014] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollar, P., and Zitnick, L. (2014). Microsoft coco: Common objects in context. In *ECCV*. European Conference on Computer Vision.

[Lindblad, 1999] Lindblad, G. (1999). A general no-cloning theorem. *Letters in Mathematical Physics*, 47(2):189–196.

[Livieris and Pintelas, 2013] Livieris, I. E. and Pintelas, P. (2013). A new conjugate gradient algorithm for training neural networks based on a modified secant equation. *Applied Mathematics and Computation*, 221:491–502.

[Ma et al., 2019] Ma, W.-D. K., Lewis, J. P., and Kleijn, W. B. (2019). The hsic bottleneck: Deep learning without back-propagation.

[Minka, 2000] Minka, T. (2000). Bayesian linear regression.

[Møller, 1990] Møller, M. F. (1990). *A scaled conjugate gradient algorithm for fast supervised learning*. Aarhus University, Computer Science Department.

[Mudambi and Schuff, 2010] Mudambi, S. M. and Schuff, D. (2010). Research note: What makes a helpful online review? a study of customer reviews on amazon. com. *MIS quarterly*, pages 185–200.

[Nilsson, 1965] Nilsson, N. J. (1965). Learning machines.

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. (2006). *Numerical optimization*. Springer Science & Business Media.

[Rios and Sahinidis, 2013] Rios, L. M. and Sahinidis, N. V. (2013). Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293.

[Robinson, 2013] Robinson, R. C. (2013). Introduction to mathematical optimization. pages 75–94. Northwestern University.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms.

[Rumelhart David E. et al., 1986] Rumelhart David E., Hinton Geoffrey E., and Williams Ronald J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[Saad, 1981] Saad, Y. (1981). Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of computation*, 37(155):105–126.

[Sahinidis, 1996] Sahinidis, N. V. (1996). Baron: A general purpose global optimization software package. *Journal of Global Optimization*.

[Servedio, 2001] Servedio, R. A. (2001). Separating quantum and classical learning. In Orejas, F., Spirakis, P. G., and van Leeuwen, J., editors, *Automata, Languages and Programming*, page 1065–1080, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Servedio and Gortler, 2004] Servedio, R. A. and Gortler, S. J. (2004). Equivalences and separations between quantum and classical learnability. *SIAM Journal on Computing*, 33(5):1067–1092.

[Shor, 1997] Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509.

[StackExchange, 2013] StackExchange (2013). Problems with dimensionality in genetic algorithms. `https://stats.stackexchange.com/questions/55887/backpropagation-vs-genetic-algorithm-for-neural-network-training`.

[Tange, 2011] Tange, O. (2011). Gnu parallel - the command-line power tool.

[Tishby et al., 2000] Tishby, N., Pereira, F. C., and Bialek, W. (2000). The information bottleneck method. *arXiv preprint physics/0004057*.

Appendix A

To be more pedantic

Probability

Measure theory offers the analogous definitions of probability, more from the mathematical analysis point of view.

Let $\mathbb{V}$ be topological space, with Borel sigma-algebra [1] $\mathcal{B}(\mathbb{V})$ defined on it. Let us define `random variable` $\mathbb{X}$ as a measurable function $\mathbb{X} : \mathbb{V} \to \mathbb{R}$. We call random variable `discrete` when it takes finitely many values, i.e. $\mathbb{X}$ is a simple function.

Dual Problem

`Lagrangian multipliers`, or simply Lagrangian, is a method of relaxing the constraint in order to solve the optimization problem. For instance, for f objective function, and g the zero constraint function that is $\mathcal{L} := f - \lambda g$, where $\lambda \in \mathbb{C}$ is called the Lagrangian multiplier.

`Lagrangian dual problem` is the problem obtained by Lagrangian of the primal problem.

[1] We want to have probabilistic notion of closure under complement (negation of an event), countable sum (one of the event happened) and intersection (many events simultaneously happened).