# Université de Montréal


# Factorized second order methods in neural networks


par

## Thomas George

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences


Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

August 31, 2017

# SOMMAIRE

Les méthodes de premier ordre (descente de gradient) ont permis d'obtenir des succès impressionnants à l'aide de réseaux de neurones artificiels. Les méthodes de second ordre permettent en théorie d'accélérer l'optimisation d'une fonction, mais dans le cas des réseaux de neurones le nombre de variables est bien trop important. Dans ce mémoire de maitrise, je présente les méthodes de second ordre habituellement appliquées en optimisation, ainsi que des méthodes approchées qui permettent de les appliquer aux réseaux de neurones profonds.

**Mots-clés**: Apprentissage automatique, apprentissage profond, optimisation, second ordre, gradient naturel

# SUMMARY

First order methods (gradient descent) enabled impressive successes using artificial neural networks. Second order methods theoretically allow accelerating optimization of functions, but in the case of neural networks the number of variables is far too big. In this master's thesis, I present usual second order methods, as well as approximate methods that allow applying them to deep neural networks.

**Keywords**: Machine learning, deep learning, optimization, second order, natural gradient

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# REMERCIEMENTS

Passer ses journées à étudier, apprendre et explorer un sujet aussi passionant que l'intelligence artificielle est une chance. Celle-ci nous a été rendu possible, et à moi aussi, par des milliers d'années de construction d'une civilisation humaine et de savoirs scientifiques. Tous les acteurs de ces constructions, illustres et inconnus, sont à remercier.

De manière plus prosaïque je souhaite remercier le Québec et tous les québécois que j'ai rencontré pendant ces 2 dernières années, qui m'ont accueilli dans cette étonnante ville de Montréal: Merci.

À mes proches, famille et amis, à qui j'ai parfois eu beaucoup de mal à expliquer à quoi j'occupais mes journées. Pour leur regard extérieur souvent circonspect, mais surtout pour les bons moments passés: Merci.

Aux membres du jury qui s'apprêtent à prendre connaissance du fruit de mon travail: Merci.

Aux étudiants et professeurs du MILA, et en particulier César toujours présent pour discuter d'une nouvelle idée et l'implémenter à n'importe quelle heure de la nuit pendant les semaines de deadline: Merci.

À Pascal Vincent qui m'a supervisé dans ce travail de recherche, tout en me donnant les clés pour débloquer les situations où j'avais déjà retourné le problème dans tous les sens: Merci.

Et merci à Lisa bien sûr... pas le laboratoire !

# INTRODUCTION

This thesis presents my work during my master at MILA under the supervision of Pascal Vincent.

Artificial neural networks are a powerful machine learning tool for modeling complex functions. Training a neural network for a given task often reduces to minimizing a scalar function of several millions of variables, which are the parameters of the model. While optimization is a full field of research on its own, usual methods do not scale to the order of magnitude of several millions of variables. For this reason neural networks practitioners stick to first order optimization methods, while not benefiting of the acceleration provided by using more powerful methods. Amongst the family of optimization methods, second order methods are a conceptually simple way of accelerating optimization. But practically, they require too much memory and computational power in order to be really useful when scaled to millions of parameters. We circumvent these practical constraints by approximating second order methods, trading off between computational cost, and speed up.

This work is mostly focused on optimization applied to artificial neural networks. My contributions are a deeper understanding of the many techniques involving second order techniques applied to neural networks, the derivation of new expressions tuned for the particular structure of neural networks, and their use in the definition of a new algorithm that competes with current state of the art on a standard benchmark. Crucially, this benchmark is a deep network with several millions of parameters, and is trained to convergence in approximately 1h on a single computer.

In the process, I also explore alternatives to the backpropagation technique, which is used to efficiently obtain gradients in neural network optimization. As a core component of training neural networks, backpropagation has been the object of much research efforts since it was first used in the 1980s, but it has remained exactly the same since then. We contribute to this research by exploiting the sequential computations of backpropagation. We derive an alternative to backpropagation and experimentally show that it is able to find better update directions, at the cost of more computation. This contribution is of no practical use as is, because it requires too much computation. However it is a proof a concept that backpropagation can be improved. As the foundation of the whole training procedure of

neural networks, a computationnally cheaper method of improving backpropagation would impact all other optimization methods that rely on computing the gradients.

This document is organized as follows:

- the first chapter sets up the basic framework of machine learning and introduces neural networks ;
- in chapter 2 we introduce the usual methods of optimization that have enabled the recent successes in deep learning ;
- in chapter 3 we review 2 second order methods called *Gauss Newton* and *natural gradient*, and we show how they relate and how they differ ;
- in chapter 4 we describe the experimental setup that we use next to assess the performance of our algorithms, and we contribute a simple hyperparameter tuning procedure ;
- in chapter 5 we contribute a new technique that modifies backpropagation ;
- the last chapter presents a factorization that we can use to efficiently approximate second order methods. We contribute a detailed derivation of second order methods for the particular structure of neural networks, and we highlight the links with some old tricks. We also contribute a new optimization algorithm that exploits this factorization.

# CONTENTS

# Chapter 1

# NEURAL NETWORKS

In this chapter, we will introduce concepts and techniques that are used in artificial intelligence tasks. In particular, we will introduce neural networks, that have proven a powerful model and produced state of the art results in a variety of tasks.

## 1.1. Artificial intelligence

Intelligence is a difficult concept to define. We will use the following definition: the ability to make sensible decisions in a given situation, possibly making use of a memory of past events that share similarities with the current situation. The most intelligent individual agent that we are aware of nowadays is certainly the human being, amongst other animals. Human beings are constantly making decisions given their perception of the world that is provided by their 5 senses, using knowledge that they have studied or experienced in their life. But there is no *a priori* reason to think that intelligence could not be present in other systems, and in particular artificial intelligence is a scientific field that aims at implementing intelligence in non-living machines.

How our society of humans can benefit from artificial intelligence is still an open question, out of the scope of the present document. Regardless, given the recent popularity of artificial intelligence among public research laboratories and in the industry, and the recent successes at solving complex tasks, we can say without taking risks that artificial intelligence will continue to play a big role in shaping the future of our society.

From a more practical perspective, implementing an artificial intelligent machine requires designing a system that takes data that represent the current situation, data that represents the memory of the machine, and output a decision using this data.

To put things into context, we will now describe an example task. We want to design a program that takes a picture of an animal and a sound as input, and outputs whether it thinks the animal present in the picture makes the provided sound. In a computer, a picture is often encoded as a mathematical tensor of scalar values or pixels, the sound as a timeseries of samples of the sound wave, and the final decision can be a single scalar ~~values~~value, which

will be close to 0 ~~is~~ if the animal is very unlikely to make the noise, or to 1 if the animal is very likely to make the noise. The complex machinery inbetween is the intelligent part.

Manually designing a program for such a task is an overwhelming task. Even provided that the input image is quite a small image of $32 \times 32$ RGB pixels and the sound lasts $1s$ recorded at a sample rate of 20kHz we have a total of $32 \times 32 \times 3 + 20\,000 = 23\,072$ scalars. If we restrict each of these numbers to have 256 possible values, it leaves us with $256^{23\,072} \approx 10^{55\,000}$ possible combinations. Even if we only keep the combinations that are plausible, there is too many to create a naive program. Even with carefully engineered feature extractors based on image and sound processing techniques, the remaining work ~~of~~ is still challenging.

Instead, the most successful attempts at solving such tasks use a procedure called **machine learning**: instead of manually defining our program, we define a generic model, and we use a dataset of annotated examples of picture, sound, and the corresponding answer, and we leave to the computer the task of extracting information from the dataset to tune the model so as to obtain the desired program.

## 1.2. Machine learning

### 1.2.1. Parametric functions and learning

Generally speaking, machine learning consists in finding an unknown function $f$ from a family of functions $\mathcal{F}$, that will solve a certain task. We typically restrict our search to a ~~smaller~~ small family of functions, which consists in parametrized functions $\mathcal{F}_\theta$. We will denote by $f_\theta$ such a function, parametrized by a vector of parameters $\theta$. Adapting the value of the parameters will change the output of the function $f_\theta$. The challenges of machine learning are to find a correct parametrization so that our desired function can be approached by a member of $\mathcal{F}_\theta$, and to learn the parameters of this target function.

To this end, we need a measure of the performance of a given function at solving our task. We ~~chose~~ choose a loss function ~~$l$~~$\ell$, adapted to this task. The better our function, the lower the value of ~~$l$~~$\ell$. The remaining ingredient is a data generating distribution $p$ from which we sample datapoints $x \sim p$ that are our examples. A measure of the performance of a function $f_\theta$ for the given task is given by the risk:

$$\mathcal{R}(\theta, p) = \mathbb{E}_{x\sim p}\left[\ell\left(f_\theta(x), x\right)\right]$$

$\mathcal{R}$ is a scalar value. If this value is high, then $f_\theta$ is bad at solving the desired task. In the opposite, the best function can be found by adjusting $\theta$ so as to reach the smallest value of $\mathcal{R}$. The best value for the parameter vector $\theta^*$ is given by:

$$\theta^* \quad = \quad \text{argmin}_\theta \mathcal{R}(\theta, p)$$

Finding this value $\theta^*$ is the task of learning from the data.

We now present two common tasks and their corresponding loss functions. We will restrict to the less general setting of *supervised* learning, where each data point is composed of an input $x$ and a true target $y$. The risk can be written as $\mathcal{R}(\theta, p) = \mathbb{E}_{x,y\sim p}[\ell(f_\theta(x), y)]$ meaning that the function $f_\theta$ only uses the input, and returns a target $\hat{y} = f_\theta(x)$. In supervised learning, the loss function compares the true target $y$ with the current estimated target $\hat{y}$.

In **regression**, the input vector $x$ is mapped to a numerical value $y$. To assess the performance of $f_\theta$, we use the loss function ~~$l(f_\theta(x), y) = \|f_\theta(x) - y\|_2^2$~~ $\ell(f_\theta(x), y) = \|f_\theta(x) - y\|_2^2$, called the quadratic error. It reaches its minimum 0 when $f_\theta(x) = y$. For example we can design a model that predicts the price of a real estate, given some features such as the size of the house, the number of bedrooms and whether it possesses a fireplace.

In supervised **classification**, we classify each data point $x$ into a category $y$. A natural loss that comes up is the misclassification indicator function $\mathbf{1}(f_\theta(x), y) = \{0 \text{ if } f_\theta(x) = y \text{ or } 1 \text{ otherwise}\}$. It counts the examples that are misclassified. This function present the disadvantage of not being differentiable (it is not even continuous), and we will see in future sections that differentiability is a valuable property for machine learning. Instead, we usually make our function $f_\theta$ output a vector of the number of categories, which represents computed probabilites of being a member of each category (a scalar between 0 and 1). We use the loss called *cross entropy* ~~$l(f_\theta(x), y) = -\log\left((f_\theta(x))_y\right)$~~ $\ell(f_\theta(x), y) = -\log\left((f_\theta(x))_y\right)$. This will push the probability of the correct category toward 1. An example classification task is proposed by the ImageNet project [Deng et al., 2009] where the task is to classify images to detect what they represent such as an animal, or a car and so on.

### 1.2.2. Empirical risk and bias-variance tradeoff

In practice, often, we do not have access to a data generating function $p$, but instead we have a limited number of samples from it. This dataset of examples gives us an estimate of the true risk, by replacing the expectation with a finite sum, called the empirical risk $R$:

$$\mathcal{R}(\theta, p) \quad \approx \quad R(\theta, \mathcal{D}) \quad = \quad \frac{1}{n}\sum_{x\in\mathcal{D}} \ell(f_\theta(x), x) \tag{1.2.1}$$

$n$ is the total number of examples in $\mathcal{D}$.

For random values of the parameters $\theta$, the empirical risk and the true risk will have similar values. But this is not the case when the parameters have been tuned so that the empirical risk is minimum. In the extreme case, consider a model that has memorized all

examples of the training set by heart. In order to make a prediction for a new example, this model will seek the closest example in $\mathcal{D}$, in term of the euclidean distance, and output the exact same answer than this closest example. This model is called a 1-nearest neighbour regressor or classifier regarding the considered task. In this case the empirical risk is 0, but we have no guarantee that the model generalizes on new examples.

A model with too much expressivity, or *variance*, will be able to learn all examples in the training set by heart without having the ability to generalize on new examples, which is called *overfitting*. A model with not enough expressivity will not be able to perform well even on the training set, which is called *underfitting*. In the meantime it will have a similar performance on the true data generating distribution. We say that there is a *bias* toward a family of model. The bias-variance tradeoff consists in selecting a model that has sufficient expressivity to have a good performance on the train set, while not having too much expressivity so that it will not overfit, and still have good performance on the true data generating distribution.

### 1.2.3. Regularization

A way of combatting overfitting is to use regularization. It is a way of constraining the values of the parameters of a function using priors. For example L2 regularization penalizes the squared norm of the parameter vector. It constrains all values to stay small.

Data augmentation is another mean of combatting overfitting. We can use the knowledge that we have of our dataset to create new examples. For example for a classification task of images, we know from our experience of the world that rotating or translating an image will not change its content. We can thus artificially augment our training set by including rotated and translated versions of the same images.

## 1.3. Neural networks

Neural networks are a family of parametrized models. They have empirically proven very powerful at solving complex tasks. Along with the availability of easy to use frameworks to build neural networks and learn from data, ~~has developed new interests~~ new interests have developed from industry to integrate artificial intelligence inspired techniques in more and more products. The first commercial successes date back to the 90s when AT&T developed an automated system to read handwritten digits on bank checks, using convolutional neural networks [LeCun et al., 1998]. Recent successes include advances in machine translation, image and voice recognition, close-to-realistic image generation. They have applications in online services integrated in smartphones, but also enable the invention of new automated systems that will benefit more traditional industries, (energy, agriculture, arts, ..)

## 1.4. Common types of neural networks

### 1.4.1. Multilayer perceptron

We now define the simplest neural network structure called the perceptron [Rosenblatt, 1961]. From an input data vector $x$, it creates a prediction $y$ using the relation $y(x) = f(\langle w, x \rangle + b)$. $w$ is called the weight vector, and $b$ is the bias. $f$ is a function, and is sometimes called the nonlinearity or activation function as it allows the function $y$ to be different ~~than~~ from just a linear function of its input $x$. From a trained perceptron, we take a decision for an example $x$ by comparing the value of the corresponding $y$ using a threshold value. Perceptrons were implemented before the invention of modern computers, as complex electronic circuits. The weights were encoded in hardware potentiometers and trained using an error-propagating process. Remarkably, these complex pieces of machinery were capable of obtaining good results for the task of recognizing simple shape images.

These perceptrons were designed to approximately replicate the computations made by a network of biological neurons. Each neuron gets input data from several other neurons, consisting in voltage spikes. The rate at which these spikes occur can be intepreted as whether a neuron is excited or not. Each neuron has different sensibilities regarding how it will react to an increase in spike rate from other neurons, this sensibility being mimicked by the weights in artificial neural networks. In its most simple modeling, the human brain is just a very complex network of these neurons. This is the inspiration for artificial neural networks.

This single perceptron is extended in a more complex model called the multilayer perceptron (MLP). It consists in alternatively stacking layers of linear transformation $a = Wx + b$ and nonlinearities $y = f(a)$, using a vectorized generalization of the perceptron: $y(x) = f(Wx + b)$. $W$ is now a weight matrix, and $b$ a bias vector. $f$ is often an elementwise function. We stack these transformations to get more complex functions. An example for 2 layers gives a function $y(x) = f_2(W_2 f_1(W_1 x + b_1) + b_2)$. The intermediate values obtained at each layer $f_1(W_1 x + b_1)$ are called the hidden representations as they are new representations of the same input data, but encoded in a different way. A trained neural network will create representations that are better suited for its task. For example, if we imagine a task of classifying images between those which picture a dog and those with a cat, we are interested in a high level representation of characteristics such as a long tail, whiskers or sharp ears.

### 1.4.2. Convolutional networks

Convolutional networks [LeCun et al., 1989] are well-suited for tasks involving sequential (timeseries) or spatial data (images). Instead of multiplying a weight matrix with the whole input vector as in MLPs, we split this input into smaller chunks of fixed sized corresponding

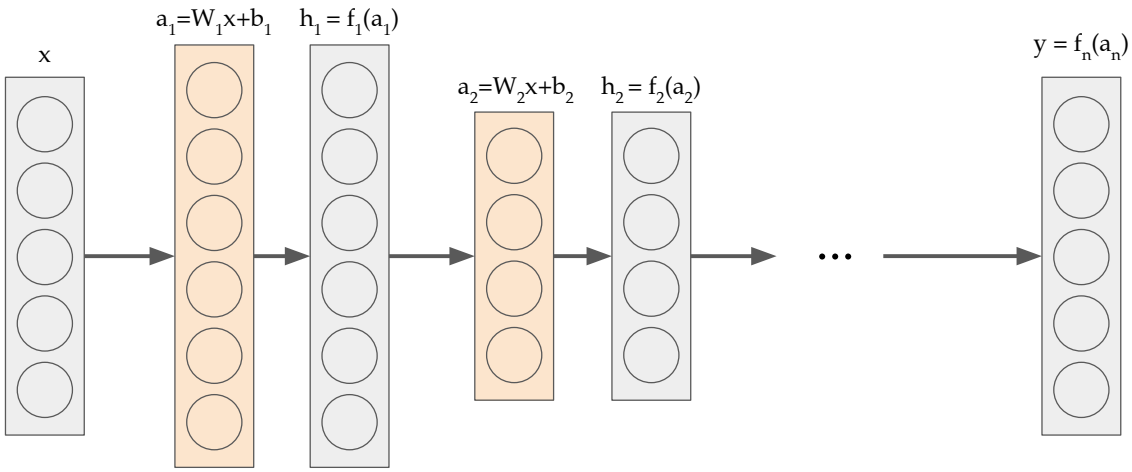


FIGURE 1.1. A multilayer perceptron consists in alternatively stacking layers of a linear transformation and a nonlinearity

to small vectors (1d case) or small matrices (2d case). The same weight matrix is applied to each of ~~this~~ these smaller vectors so as to get a result at each position. In fact it amounts to a standard MLP with sparse connectivity and with the ~~weight matrix~~ weights sharing values between corresponding positions.

### 1.4.3. Autoencoders

Autoencoders [Hinton and Salakhutdinov, 2006, Vincent et al., 2008] are neural networks that are composed of an encoder part ~~a decoder~~and a decoder part. The encoder takes the input and encodes encodes it to a new representation (often with less dimension than the input). The decoder takes the encoded input with the task of reconstructing the output. The encoded representation is often a layer with less neurons than the size of the input. The autoencoder is trained end-to-end, without manually taking care of the encoded representation. This representation is automatically created by learning from the data. ~~It is a special case for regression.~~ In order to be efficient, a well designed autoencoder will need to create a relevant representation of the data. For example, if we want to encode pictures of faces, relevant representations could be the gender, the color and size of hair, and so on.

## 1.5. More elaborated cost functions

We can often associate a task and its corresponding loss function: regression with the quadratic error loss, and classification with the cross entropy loss 1.2.1. Some more recent advances in neural networks make use of more complex cost functions.

**Neural art** [Gatys et al., 2015] and its feed-forward extension [Ulyanov et al., 2016] tackle the task of generating artwork images from a real world picture, that mimick the style

of a given painting. To this end, they create a cost function that measures how a generated image resembles both the picture and the painting:

$$\mathcal{L}_{\text{total}}(p, a, x) = \alpha \mathcal{L}_{\text{content}}(p, x) + \beta \mathcal{L}_{\text{style}}(a, x)$$

$p$ is the picture, $a$ is the artwork that we want to extract the style, and $x$ is any image. $\mathcal{L}_{\text{content}}$ is a loss function that measures how close $x$ is from $p$ in terms of contents, and $\mathcal{L}_{\text{style}}$ is a loss function that measures a distance from $a$ to $x$ in terms of artistic style. By minimizing $\mathcal{L}_{\text{total}}(p, a, x)$ with respect to $x$ for given $p$ and $a$, we obtain the desired image in $x$. $\alpha$ and $\beta$ are scalar values that control the influence of each part of the loss. In the original paper [Gatys et al., 2015] we start from a randomly initialized $x$ and we perform gradient descent on each pixel of $x$. In Ulyanov et al. [2016] we use a convolutional neural network to generate $x$, which takes the picture as input, and outputs the desired stylized image. This network is trained using $\mathcal{L}_{\text{total}}$. It has the main advantage of being very fast as generating new images once it has be trained on a specific artwork.

Another family of cost functions that becomes more and more popular is that of the discriminators in **Generative Adversarial Networks** [Goodfellow et al., 2014], that can be thought of as learned cost functions. In this setup, 2 networks are trained one against each other : the generator part takes random noise and generate a sample that tries to fool the discriminator. The discriminator also is a trained network that tries to classify whether its input is from a given data distribution, or if it was generated by the generator. Training these networks is very unstable, and is the object of many research at the time of this writing. But provided that we successfully trained both parts, we get a generator that is able to generate new samples of complex data, such as realistic images.

# Chapter 2

# OPTIMIZATION OF NEURAL NETWORKS

## 2.1. Gradient descent and backpropagation

### 2.1.1. Learning using gradient descent

Once we have chosen a model, and supposing that this model is capable of solving a given task with a dataset of examples of this task, the main challenge is now to learn the parameters of the model from the data. Some simple models have closed form solutions, this is for example the case for a linear model and a regression task. For more complex models such as neural networks, we can not derive a simple formula for getting the values of all parameters given a dataset. In this case, we start from an initialized network and iterate updates for our parameters until we get the expected results. To this end, we must find an efficient way of getting an update $\Delta\theta$ of our parameters $\theta$. Considering that we aim at finding the minimum of the empirical risk, such an update is given by the steepest direction of descent of the empirical risk, given by minus the gradient of the empirical risk, with respect to the parameters, denoted by $\nabla_\theta R$:

$$\nabla_\theta R \quad = \quad \frac{1}{n} \sum_i \nabla_\theta l\ell \left( f_\theta \left( x_i \right), y_i \right)$$

Once we have a direction, we must choose how far to move in this direction. One way of choosing this rate is by using a line search algorithm. But its requires evaluating our objective several times, which can be costly for deep networks or big datasets. We will stick to a simple fixed scalar learning rate $\lambda$, so that each iteration becomes:

$$\theta \quad \leftarrow \quad \theta - \lambda \nabla_\theta R$$

Of course the learning rate $\lambda$ plays a very important role. If we choose a value that is too small then it will take ~~several~~ many steps to reach the same performance, so it will take longer. If the value is too ~~big~~ large then we can go too far, to a point in the space

of parameters where the gradient has changed so the direction that we are following is no longer a descent direction. In this case, we can even decrease the performance. For a practical example think of a valley. We start from a side of the valley and follow the steepest descent direction. If we go too far we will pass the bottom of the valley and start going up again.

### 2.1.2. Computing the gradients using backpropagation

It might be difficult to get an exact expression for the gradient of a complex function, such as a neural network. What enabled the success of neural networks was a smart use of the chain rule for splitting the computation of the gradient, into a sequence of linear algebra operations, that is described in figure 2.1. For example we can decompose the gradient going through a layer $h_l = f_l(a_l) = f_l(W_l h_{l-1} + b_l)$ using the expression:

$$\begin{aligned} \nabla_{h_{l-1}} \underline{l}\ell &= \left(\mathbf{J}_{a_l}^{h_{l-1}}\right)^T \nabla_{a_l} \underline{l}\ell \\ &= \left(\mathbf{J}_{a_l}^{h_{l-1}}\right)^T \left(\mathbf{J}_{h_l}^{a_l}\right)^T \nabla_{h_l} \underline{l}\ell \end{aligned}$$

We denote by $\mathbf{J}_f^x$ the jacobian of the vector function $f$ with respect to $x$. It is the matrix composed of the partial derivatives $\left(\mathbf{J}_f^x\right)_{ij} = \frac{\partial f_i}{\partial x_j}$, so it has dimension $n_f \times n_x$. In the particular case of neural networks we have simple expressions for the jacobians of the backpropagated signal (red arrows in figure 2.1):

$$\begin{aligned} \mathbf{J}_{a_l}^{h_{l-1}} &= W_l \\ \mathbf{J}_{h_l}^{a_l} &= \operatorname{diag}\left(f_l'(a_l)\right) \end{aligned}$$

where diag is the operation that takes a vector and transforms it to a diagonal matrix with the values of the vector as diagonal terms. These jacobians can be thought of the gradient flow between layers.

We also have expressions for the jacobians of the activations with respect to the parameters (green arrows in figure 2.1):

$$\begin{aligned} \mathbf{J}_{a_l}^{W_l} &= \nabla_{a_l} \underline{l}\ell \left(h_{l-1}\right)^T \\ \mathbf{J}_{a_l}^{b_l} &= \nabla_{a_l} \underline{l}\ell \end{aligned}$$

### 2.1.3. Automatic differentiation tools

A key component in training neural networks is the library that we use to implement our models. The difficulty of implementing backpropagation in all kinds of neural networks inspired models, is solved using an automatic differentiation tool, such as Theano [Bastien

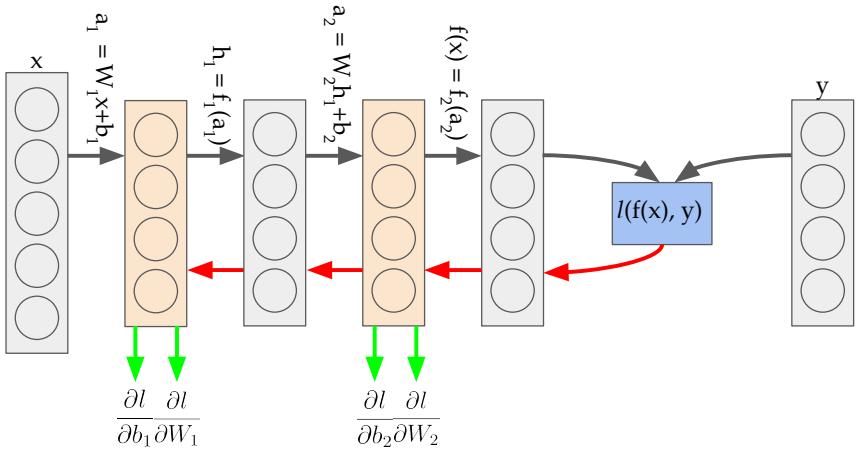


FIGURE 2.1. Forward (in black) and backward (in red) propagation of the intermediate results of the process of computing the output of the network and the gradient corresponding to this output and the desired "true" output. The green arrows represent the computation of the gradients with respect to the parameters, given the gradients with respect to the pre-activations.

et al., 2012]. Using such a tool we can define a model and a scalar cost function, then call a method `grad` that takes care of building the computational graph of all operations required to get the gradients with respect to the parameters.

## 2.2. STOCHASTIC GRADIENT DESCENT

While backpropagation is an efficient way of computing the exact gradient of the empirical risk with respect to the parameters, in practice we are not required to use its exact value, but rather we can use an estimate of the gradient, as long as this estimate will make our objective decrease. It is worth recalling at this point that even the exact gradient of the empirical risk is different of the real gradient we would like to follow, which is the gradient of the true expected risk ~~1.2.2~~(as discussed in section 1.2.2).

A good estimate is obtained by computing the gradient using a smaller subset of our dataset, called a mini-batch. Replacing the gradient descent update with this estimate is called mini-batch gradient descent, and the extreme case where we take a mini-batch size of 1 is called **stochastic gradient descent** (SGD) [Bottou, 2010]. The main benefit of using SGD instead of full gradient descent is that we can reduce the memory required to compute the gradient, in order to fit in the memory of a GPU, which is very efficient for computing vector operations such as the one required to get the values of our gradients. The memory required to backpropagate the intermediate gradients (red arrows in figure 2.1) is proportional to the size of the mini-batch, and so we face a trade-off between obtaining a noisier update direction with a small mini-batch, but very fast, and obtaining a direction

that is more precise but that takes longer. Typically it is more efficient to compute several noisy gradients using mini-batches ~~that~~ in the GPU memory and make several updates, than to compute a single more precise update on a bigger mini-batch or the full dataset.

## 2.3. Hyperparameters

In the preceding sections we have introduced the learning rate (section 2.1) and the mini-batch size (section 2.2). These values are called hyperparameters, which is another kind of parametrization of our learning procedure. Hyperparameters also include the structure of our model, such as the number of hidden layers and hidden units, the number of training iterations, the coefficients of the regularization terms. The success of our learning procedure is ~~dependant~~ dependent on the values of the hyperparameters. We do not find the optimal value using gradient descent, but instead we tune it by running several time the same experiment with different hyperparameter values, and compare the final value of the risk on a held-out set of examples called the validation set.

A difficulty in comparing optimization algorithms resides in the fact that there performances can change drastically for different values of hyperparameters. Optimization papers sometimes mention heuristics that they experimentally found provide with a sensible value for some hyperparameters. But to overcome this difficulty and provide "fair" benchmarks, we usually tune the values of the hyperparameters by trying several sets of values. Hyperparameters tuning is a research field on its own, so we will just introduce 2 existing methods and we will later motivate our use of a new technique that we call biased random search (section 4.2).

The most simple ~~hyperparamater~~ hyperparameter tuning procedure, called **grid search**, consists in selecting values at fixed length intervals, or using a logarithmic scale. A simple example would be a training procedure involving only one hyperparameter: the learning rate. We can launch several experiments for all values in $\{10^{-3}, 10^{-2}, 10^{-1}, 1\}$ for a fixed number of updates and select the learning rate for which we obtained the best value for our target criteria such as the validation loss. When generalizing to several hyperparameters, we have to select all combinations of values, which make our search space grow exponentially, and similarly for the number of experiments we will have to run.

A first extension to grid search replaces the fixed length intervals by random samples in our search space. It is called **random search**. Its main advantage over grid search shows up when any ~~hyperparameters~~ hyperparameter has no important effect on the learning algorithm [Bergstra and Bengio, 2012]. It will explore more different values for the other hyperparameters. In this case, it clearly appears that they are correlated, in the sense that the best value for one hyperparameter depends on the chosen value for the other hyperparameter.

In the rest of this work, we will use an extension of random search that we call **biased random search**, and that we present in section 4.2.

To give a more intuitive understanding of the effect of the hyperparameter values on the learning algorithm, we also introduce a graphical representation that helps making sense of the interaction between different hyperparameters, which we now describe. We launch a hyperparameter search on 2 hyperparameters and plot this point on a scatter plot, with a color scale depicting the final result of each experiment. By observing this plot, we can identify ~~2d~~ 2D patterns of the link between 2 hyperparameters. As an illustration, we use the task described in section 4.1.1, using standard stochastic gradient descent and by keeping all hyperparameters values fixed, that is we use a fixed mini-batch size, and a fixed number of parameter updates. We tune 2 hyperparameters: the learning rate and the variance of the initial random weights, and we plot the result in figure 2.2. These plots show the interaction between 2 hyperparameters. We observe that the best values lie in a region with a very particular shape that can be assimilated to a tilted valley (it is not parallel to the x-axis nor the y-axis).

These plots and this random search technique are a key component for assessing the true performance of optimization techniques that we present in section 4 and 5. Indeed it is easy to experimentally find that a new optimization technique which gives better performance than a baseline if we spend too much time tuning hyperparameters for our new technique but stick to default values for baselines.

## 2.4. Limits of (stochastic) gradient descent and some directions to overcome them

We can think of the task of training a neural network as the one of finding the minimum of a scalar field in $n$ dimensions, $n$ being the number of parameters. Each gradient descent step is a small shift in this field. We must ensure that this path in the field of the empirical risk is feasible. We now present reasons that can make this field have a pathological structure, and directions to avoid these difficulties.

### 2.4.1. Gradient magnitudes

A common issue for deep networks or recurrents networks is how to control the magnitude of the gradient flow for many layers. If the ~~jacobians $\left(\mathbf{J}^{h_{l-1}}_{a_l}\right)^T \left(\mathbf{J}^{a_l}_{h_l}\right)^T$ have a spectral norm that is too big~~spectral norm of the jacobians $\mathbf{J}^{h_{l-1}}_{a_l}\mathbf{J}^{a_l}_{h_l}$ is too large, then the ~~gradient will become bigger and bigger~~ norm of the gradient will exponentially grow for lower layers. This can happen if the weight matrices have singular values that are too ~~big~~ large compared to 1 ~~, or if~~ while the derivatives of the activation functions take ~~big~~ large values. In this case, we are in a situation of exploding gradients. This effect is amplified in recurrent networks, where the same weight matrix is repeatedly used in the backward pass. For such a ill-conditioned

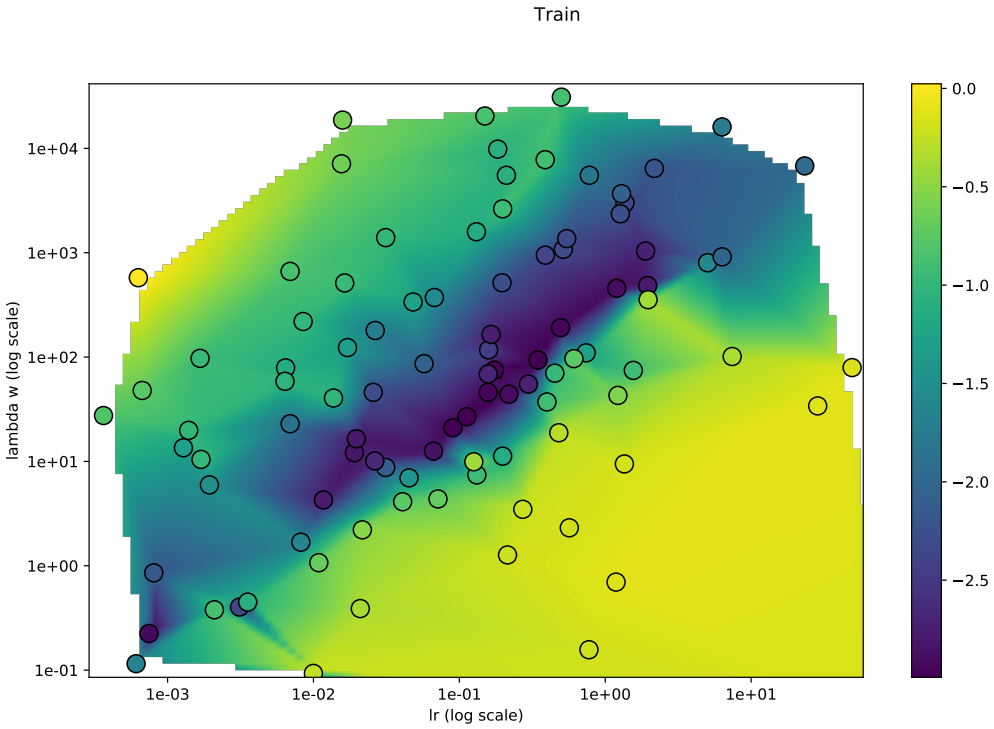


FIGURE 2.2. Relation between 2 hyperparameters: for this experiment we can clearly see that the plotted hyperparameters are not independant one from each other. The color scale represents the final performance (best performing runs in blue).

problem, gradient descent will not be effective. Indeed, in the case of exploding gradient, two layers separated by several others will have updates of different order of magnitude.

This effect can be mitigated using gradient clipping [Pascanu et al., 2013]. We can also use second order methods to compensate for gradient vanishing/exploding as proposed in [Martens and Sutskever, 2011].

### 2.4.2. Initialization

Initialization of the weight matrices is of crucial importance. In terms of our empirical risk field in the space of parameters, it controls how far we start from a minimum. A good initialization scheme must at least make sure that there is enough signal flowing in forward and backward direction, that is: the weights must be chosen not too small, otherwise the forward signal will be smaller and smaller, and in the meantime the weights must not be too ~~big~~large, so as to avoid exploding gradients in the backward pass, and saturating functions in the forward pass.

The most common initialization scheme at the time of writing are Glorot initialization [Glorot and Bengio, 2010] and He initialization [He et al., 2015]. Glorot takes care of

maintaining a training signal during the forward pass, and the backward pass, by sampling random weights from an uniform distribution with variance $\frac{\alpha}{n_{in}+n_{out}}$, while He argues that only the forward pass matters so the weights should be initialized from a distribution with variance $\frac{\alpha}{n_{in}}$. In both cases, $\alpha$ depends on the activation function~~, and the papers~~. While Glorot and Bengio [2010] propose default values for ~~sigmoid and ReLU[Glorot et al., 2011].~~ ~~In our experiments~~tanh and ReLU, we treated $\alpha$ as a hyperparameter when assessing for the performance of an optimization algorithm, and tuned it using a biased random search (section 4.2).

### 2.4.3. Gradient smoothing methods

A family of optimization tricks ~~use~~ uses geometrical considerations in the space of parameter values. In this case with some common sense we can define a simple principle to derive better updates which is that for an equivalent decrease of the empirical risk, we must follow a direction of descent that has a smaller derivative for longer in order to achieve the same improvement as for a direction that has a larger derivative. Many popular techniques use this principle, the most successful ones at the time of writing being Adam [Kingma and Ba, 2014], RMSProp [Tieleman and Hinton, 2012], Nesterov momentum [Nesterov, 1983] and so on.

# Chapter 3

# SECOND ORDER METHODS IN NEURAL NETWORKS

In section 2.1 we described how training a neural network for a given task reduces to an optimization task: the one of minimizing the empirical risk, by iteratively updating the values of the parameters of the network. We obtained the updates by following the direction given by minus the gradient. But there are more efficient updates.

In this section, we introduce the well known second order methods known as Newton's method and the less popular but very effective natural gradient descent. We then derive the updates of this methods adapted to neural networks and we contribute a unified formulation for both methods.

## 3.1. Second order methods

### 3.1.1. Newton steps

Second order methods refer to all optimization methods that make use of the second derivative or Hessian matrix of the function to be minimized. It follows from the Taylor series decomposition of the function:

$$f(x+\Delta x) = f(x) + (\nabla f)_x^T \Delta x + \frac{1}{2}\Delta x^T \left(\nabla^2 f\right)_x \Delta x + o\left(\|\Delta x\|_2^2\right)$$

$(\nabla^2 f)_x$ is the Hessian matrix of $f$, expressed at $x$. We use the little-o notation $o$ that represents an unknown function with the only property that $\lim_{x\to 0} \frac{o(x)}{x} = 0$.

In the context of neural network optimization, $f$ is the empirical risk, and $x$ are the parameters.

By constraining $\|\Delta x\|_2^2$ too stay small, we can ignore higher order terms ($o\left(\|\Delta x\|_2^2\right) = 0$) and we have a quadratic approximation for $f$. Using this approximation in a minimization problem, we get the following minimization which has a closed form solution:

$$\begin{aligned}\Delta x^* &= \text{argmin}_{\Delta x} f(x+\Delta x) \\ &\approx \text{argmin}_{\Delta x} f(x) + (\nabla f)_x^T \Delta x + \frac{1}{2}\Delta x^T \left(\nabla^2 f\right)_x \Delta x\end{aligned}$$

This expression is solved by taking the derivative with respect to $\Delta x$, and setting it to zero which yields:

$$\left(\nabla^2 f\right)_x \Delta x = -(\nabla f)_x$$

If we assume that $f$ has a minimum in $x^*$, then the Hessian will be positive definite in $x^*$, and under the supplementary assumption that the Hessian is continuous, it will also be positive definite in a neighborhood of $x^*$. In this case, it is invertible and we get the solution:

$$\Delta x = -\left(\nabla^2 f\right)_x^{-1} (\nabla f)_x \tag{3.1.1}$$

This update ~~3.1.1~~ (eq 3.1.1) is called the **Newton step**. By making several iterations of Newton, and under the assumption that we are close enough to a minimum so that $(\nabla^2 f)_x$ remains positive definite, the updates will converge to ~~it~~this minimum.

The main difficulty of this algorithm is that it does not scale well when applied to problems with many variables such as neural network optimization. In this case $f$ is the empirical risk, and the variables that we are optimizing are the parameters of the network. The limitations come from the following aspects:

1. *Getting the value of the Hessian matrix*: Using an automatic differentiation software, we can get an expression for the Hessian, by differentiating the symbolic expression of the gradient. But unlike the computation of the gradient, the graph produced to compute the Hessian will have ~~much~~ many more nodes. We will explore this question in more details in section 3.1.4 and present an approximate value of the Hessian called Gauss-Newton.
2. *Storing the Hessian matrix*: The Hessian matrix is a square matrix of size $n_{\text{parameters}} \times n_{parameters}$. As the number of parameters grows, which is the case when building deep networks, the memory required to store the Hessian will grow in $O(n^2)$. We will present an approximation of the Hessian that saves memory in section 3.1.5.
3. *Inverting the Hessian matrix*: Inverting the Hessian matrix is also costly as it grows in $O(n^3)$ with the size of the matrix. Some techniques use 2nd order information without inverting the Hessian such as *Hessian Free* [Martens, 2010]. We propose to factorize the Hessian so as to require inverting a smaller matrix while benefiting from some 2nd order information in section 6.2.2.

4. *Saddle points*: The optimization problem of minimizing a cost function over a dataset has many more saddle points than local minima [Dauphin et al., 2014]. In this case using the Hessian will fail as it will converge to a saddle point instead of escaping from it in order to find a minimum.

### 3.1.2. The learning rate

Amongst other hyperparameters, the learning rate of standard (stochastic) gradient descent plays a particular role which we will show in the following. We use the quadratic approximation for a function $f$:

$$\Delta x^* \quad = \quad \text{argmin}_{\Delta x} f(x) + (\nabla f)_x^T \Delta x + \frac{1}{2} \Delta x^T \left(\nabla^2 f\right)_x \Delta x + o\left(\|\Delta x\|_2^2\right)$$

If we replace the Hessian with a scaled diagonal matrix $\lambda \mathbf{I}$, we can simplify this expression to the following one that is often used for deriving the first order gradient descent update:

$$\Delta x^* \quad \approx \quad \text{argmin}_{\Delta x} f(x) + (\nabla f)_x^T \Delta x + \frac{\lambda}{2} \Delta x^T \Delta x$$

By solving this minimization problem we recover the update $\Delta x^* = -\frac{1}{\lambda} (\nabla f)_x$ with $\frac{1}{\lambda}$ playing the role of the usual learning rate. But of course this $\lambda$ hides second order information. In fact, LeCun et al. [1993] proposes to automatically adapt the value of the learning rate by using the biggest eigenvalue of the hessian as $\lambda$. In this case we are guaranteed that we do not go too far in the direction of greatest curvature (which is the corresponding eigenvector). But in exchange it will equivalently scale down an update in any other direction, even if an optimal step would require to go further in this direction.

### 3.1.3. Validity of Newton for non quadratic functions and Tikhonov regularization

In the previous section, we considered that our function was approximated by its second order Taylor series decomposition. While this is true in a neighborhood of $x$, the approximation becomes less precise as we move away from $x$. In particular this is the case when the Newton step provide big updates, that is when the Hessian has at least one small eigenvalue. The corresponding eigenvector points in a direction that will have a low curvature using the quadratic approximation, so the minimum following this direction will be far away. But the actual function that we are minimizing is not a quadratic, and the terms hidden in $o\left(\|\Delta x\|_2^2\right)$ will become preponderant for bigger values of $\Delta x$.

To counter this undesirable effect, we simply add a regularization term that penalizes bigger values of $\Delta x$:

$$\begin{aligned}
\Delta x^* &= \text{argmin}_{\Delta x} f(x + \Delta x) \\
&\approx \text{argmin}_{\Delta x} f(x) + (\nabla f)_x^T \Delta x + \frac{1}{2} \Delta x^T \left(\nabla^2 f\right)_x \Delta x + \frac{\epsilon}{2} \|\Delta x\|_2^2 \\
&= \text{argmin}_{\Delta x} (\nabla f)_x^T \Delta x + \frac{1}{2} \Delta x^T \left(\left(\nabla^2 f\right)_x + \epsilon \mathbf{I}\right) \Delta x
\end{aligned}$$

This gives the Tikhonov regularized version of the Newton step:

$$\Delta x = -\left(\left(\nabla^2 f\right)_x + \epsilon \mathbf{I}\right)^{-1} (\nabla f)_x$$

This new hyperparameter $\epsilon$ controls the size of the steps, and thus plays a very similar role to the learning rate.

In addition to this, we can also mention that it stabilizes the inversion when the condition number of $(\nabla^2 f)_x$ is too big, and that it can account for the estimation error when we estimate $(\nabla^2 f)_x$ using a minibatch of examples instead of using the true risk.

### 3.1.4. Gauss-Newton approximation of the Hessian

In the case of neural network optimization, the Hessian matrix we need to evaluate is the second derivative of the empirical risk, with respect to the parameters. A first remark that we can make, is that it is also composed of a sum of second order derivatives, to be computed at each example of the dataset:

$$\begin{aligned}
\mathbf{H} &= \frac{\partial^2 R}{\partial \theta^2} \\
&= \frac{\partial^2}{\partial \theta^2} \left\{ \frac{1}{n} \sum_i \underline{l}\ell \left(f_\theta(x_i), y_i\right) \right\} \\
&= \frac{1}{n} \sum_i \frac{\partial^2}{\partial \theta^2} \left\{ \underline{l}\ell \left(f_\theta(x_i), y_i\right) \right\}
\end{aligned}$$

By making use of the chain rule we can also give an expression for the second derivative of the loss, for a single example. We start with the first derivative:

$$\frac{\partial}{\partial \theta} \left\{ \underline{l}\ell \left(f_\theta(x_i), y_i\right) \right\} = \mathbf{J}_\theta(x_i, \theta)^T \left( \frac{\partial}{\partial f} \left\{ \underline{l}\ell \left(f_\theta(x_i), y_i\right) \right\} \right)^T$$

$\mathbf{J}$ is the jacobian of the output of the network $f$ with respect to the parameters $\theta$. In this notation we made the dependance in $\theta$ of both parts of the product explicit. Note that both parts also take different values for each examples $x_i$. We now derive this expression once more to obtain the Hessian:

| Loss function | $\frac{\partial^2}{\partial f_\theta^2}\left\{l\left(f_\theta\left(x_i\right), y_i\right)\right\} \frac{\partial^2}{\partial f_\theta^2}\left\{\ell\left(f_\theta\left(x_i\right), y_i\right)\right\}$ |
|---|---|
| quadratic error | $\mathbf{I}$ |
| cross entropy for binary decision | $\frac{y_i}{\left(f_\theta(x_i)\right)^2}+\frac{1-y_i}{\left(1-f_\theta(x_i)\right)^2}$ |
| cross entropy for multiclass classification | $\operatorname{diag}\left(\frac{y_i}{\left(f_\theta(x_i)\right)^2}\right)$ |

TABLE 3. I. Expressions for the Gauss-Newton approximation of the Hessian, for a single example $x_i$. For the cross entropy, all operations (division, squarred value) are elementwise, and the diag function transforms a vector into a diagonal matrix with the vector values on its diagonal. Full derivation in appendix.

$$\frac{\partial^2}{\partial \theta^2}\left\{\underline{l\ell}\left(f\left(x_i, \theta\right), y_i\right)\right\} = \underbrace{\mathbf{J}_\theta\left(x_i, \theta\right)^T \frac{\partial^2}{\partial f^2}\left\{\ell\left(f_\theta\left(x_i\right), y_i\right)\right\} \mathbf{J}_\theta\left(x_i, \theta\right)}_{G_f\left(x_i, \theta\right)} + \sum_j\left(\nabla^2 f_\theta\left(x_i\right)_j\right)\left(\frac{\partial}{\partial f_j}\left\{\underline{l\ell}\left(f_\theta\left(x_i\right), y_i\right)\right\}\right)^T$$

$G_f\left(x_i, \theta\right)$ is called the Gauss-Newton (GN) approximation of the Hessian [Schraudolph, 2002]. The remainder is proportional to $\frac{\partial}{\partial f_j}\left\{l\left(f_\theta\left(x_i\right), y_i\right)\right\} \frac{\partial}{\partial f_j}\left\{\ell\left(f_\theta\left(x_i\right), y_i\right)\right\}$. As we get closer to the ~~the~~ optimum, this part will go toward 0 as it is a first derivative, so the approximation will get more precise. At a minimum for $l\left(f_\theta\left(x_i\right), y_i\right) \ell\left(f_\theta\left(x_i\right), y_i\right)$, we will have $\frac{\partial^2}{\partial \theta^2}\left\{l\left(f_\theta\left(x_i\right), y_i\right)\right\}=G_f\left(x_i, \theta\right) \frac{\partial^2}{\partial \theta^2}\left\{\ell\left(f_\theta\left(x_i\right), y_i\right)\right\}=G_f\left(x_i, \theta\right)$ so it is a reasonable approximation to use in practice. Note that a minimum for the empirical risk $R(\theta)$ will not necessarily be a minimum for each example $l\left(f_\theta\left(x_i\right), y_i\right) \ell\left(f_\theta\left(x_i\right), y_i\right)$, especially if the capacity of the neural network is not sufficient to model the data distribution.

In terms of computational cost, we can also note that we can compute the GN part using standard backpropagation, but this time of the jacobian. The other term is much more complicated because it involves a second derivative of a composed function.

In practice, $G_f\left(x_i, \theta\right)$ presents a much more convenient expression for common loss functions, as the second derivative of the loss with respect to the ouput of the network simplifies (table 3. I).

We finally give an expression for the Gauss-Newton approximation of the Hessian for the empirical risk:

$$G_f(\theta) = \frac{1}{n} \sum_i \mathbf{J}_\theta\left(x_i, \theta\right)^T \frac{\partial^2}{\partial f^2}\left\{\underline{l\ell}\left(f_\theta\left(x_i\right), y_i\right)\right\} \mathbf{J}_\theta\left(x_i, \theta\right)$$

We will show in section 6.2 that this matrix can be factorized to design optimization algorithms adapted to the particular structure of neural networks.

### 3.1.5. Block diagonal Hessian

Apart from the issue of computing a value for the hessian matrix, a main limit is that we need to invert it. The hessian matrix has size $n_{\text{parameters}} \times n_{\text{parameters}}$, and the procedure used for numerically inverting a square matrix requires $O(n^3)$ operations so it rapidly becomes untractable for deep networks. A first approximation we make is by ignoring the interactions between the parameters of different layers. We make the hessian block diagonal, each block having the size of the number of parameters of the corresponding layer. An interesting property of block diagonal matrices is that we get the inverse by inverting every smaller block separately:

$$\left(\nabla^2 f\right)^{-1} \approx \begin{pmatrix} \mathbf{H}_1 & 0 & \cdots & 0 \\ 0 & \mathbf{H}_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \mathbf{H}_n \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{H}_1^{-1} & 0 & \cdots & 0 \\ 0 & \mathbf{H}_2^{-1} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \mathbf{H}_n^{-1} \end{pmatrix}$$

It also makes the implementation easier, as we can treat each block "locally" in the network, and use its inverse to update the gradient direction for the corresponding block (or layer) using $\theta_i \leftarrow \theta_i - \lambda \mathbf{H}_i^{-1} \frac{\partial C}{\partial \theta_i}$. We do not need to store a big $n_{\text{parameters}} \times n_{\text{parameters}}$ matrix.

## 3.2. Natural gradient methods

We now present the natural gradient. We give some context and ~~interpration~~ interpretation for the natural gradient, and we give its expression for neural networks.

### 3.2.1. Fisher Information Matrix

The Fisher information matrix (FIM) is well used in statistics. In the context of machine learning, and in particular deep learning, we use its inverse as a preconditioner for the gradient descent algorithm, similarly to the Newton algorithm (section 3.1.1). In this section, we show how the FIM can be derived from the KL divergence and how we get a better "natural" gradient using this information. Let us first write the definition of the KL divergence for 2 distributions $p$ and $q$:

$$\mathrm{KL}(p \parallel q) = \mathbb{E}_p\left[\log\left(\frac{p}{q}\right)\right]$$

From a broad view, it is a non-negative quantity that measures how much $q$ differs from $p$. In particular, $\mathrm{KL}(p \parallel q) = 0$ when $p = q$. Note that it is not symmetric, so it cannot be considered a true metric. We now use the probabilistic interpretation of neural networks, and consider that the examples from a dataset are drawn from a joint distribution

$p_\theta(x,y) = p_\theta(y|x)\,p(x)$ where $p_\theta(y|x)$ is the function that we model with the neural network, and $p(x)$ is the input distribution.

One can view natural gradient as using KL divergence as a regularizer when doing gradient descent. We will denote by $p_\theta(x,y)$ a parametric model and $\Delta\theta$ a change in its parameter values. $\mathrm{KL}\left(p_\theta(x,y) \parallel p_{\theta+\Delta\theta}(x,y)\right)$ is used as our regularizer, so that each change $\Delta\theta$ gives a desired change magnitude in the distribution space that we control using a new hyperparameter. Instead of using the full expression for $\mathrm{KL}\left(p_\theta(x,y) \parallel p_{\theta+\Delta\theta}(x,y)\right)$ we will use its second order Taylor series around $\theta$ (for full derivation see for instance Pascanu and Bengio [2013]):

$$\mathrm{KL}\left(p_\theta(x,y) \| p_{\theta+\Delta\theta}(x,y)\right) = \Delta\theta^T \mathbf{F} \Delta\theta + o(\|\Delta\theta\|_2^2)$$

$\mathbf{F} = \mathbb{E}_{p_\theta(x,y)}\left[\left(\frac{\partial \log p_\theta(x,y)}{\partial\theta}\right)^T \left(\frac{\partial \log p_\theta(x,y)}{\partial\theta}\right)\right]$ is the Fisher information matrix (FIM), which can be used directly as a regularizer as we shall see shortly. Interestingly, even if the KL divergence is not symmetric, its second order approximation is, as we also have $\mathrm{KL}\left(p_{\theta+\Delta\theta} \| p_\theta\right) = \Delta\theta^T \mathbf{F} \Delta\theta + o(\|\Delta\theta\|_2^2)$ (note that we swapped the terms in the KL).

### 3.2.2. Natural gradient descent

As noted in section 3.1.2, the parameter update vector used in ordinary gradient descent can be ~~otabined~~ obtained as the result of the following minimization problem:

$$\Delta\theta^* = \mathrm{argmin}_{\Delta\theta} \Delta\theta^T \nabla_\theta R + \frac{1}{2\lambda}\Delta\theta^T\Delta\theta$$

where $R$ is the empirical risk, as previously defined in eq 1.2.1 in section 1.2.2.

This expression can be easily solved giving the usual gradient descent update $\Delta\theta = -\lambda\nabla_\theta R$. The parameter $\lambda$ is the usual learning rate, and controls how much each parameter can change. We will now add a new regularizer using the FIM, and transform the minimization problem into:

$$\Delta\theta^* = \mathrm{argmin}_{\Delta\theta} \Delta\theta^T \nabla_\theta R + \frac{1}{2\lambda}\Delta\theta^T\Delta\theta + \frac{1}{2\epsilon}\Delta\theta^T\mathbf{F}\Delta\theta$$

We now constrain our gradient step to be small in term of change of parameter values, and also to be small in term of how much the resulting distribution changes. This expression can be solved to give $\Delta\theta^* = -\lambda\left(\mathbf{I} + \frac{\lambda}{\epsilon}\mathbf{F}\right)^{-1}\nabla_\theta R$. This expression also gives an insight for the role of $\lambda$ and $\epsilon$, which control 2 different but related quantities expressed by our constraints. This new update is called the natural gradient [Amari, 1998].

| Loss function | $D\left(f_\theta\left(x\right)\right)$ |
|---|---|
| quadratic error | $\mathbf{I}$ |
| cross entropy for binary decision | $\frac{1}{f_\theta(x)(1-f_\theta(x))}$ |
| cross entropy for multiclass classification | $\operatorname{diag}\left(\frac{1}{f_\theta(x)}\right)$ |

TABLE 3. II. Expressions for the FIM, for a single sample $x_i$. For the cross entropy, all operations (division, squarred value) are elementwise, and the diag function transforms a vector into a diagonal matrix with the vector values on its diagonal. Full derivation in appendix.

### 3.2.3. An expression for the FIM using jacobians

Using the probabilistic interpretation of neural networks, the FIM can be expressed $\mathbf{F} = \mathbb{E}_{p_\theta(x,y)}\left[\left(\frac{\partial \log p_\theta(x,y)}{\partial \theta}\right)^T \left(\frac{\partial \log p_\theta(x,y)}{\partial \theta}\right)\right]$ which simplifies in:

$$\begin{aligned}\mathbf{F} &= \mathbb{E}_{p_\theta(x,y)}\left[\left(\frac{\partial \log p_\theta\left(x,y\right)}{\partial \theta}\right)^T \left(\frac{\partial \log p_\theta\left(x,y\right)}{\partial \theta}\right)\right] \\ &= \mathbb{E}_{x\sim p(x)}\left[\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(\frac{\partial \log p_\theta\left(x,y\right)}{\partial \theta}\right)^T \left(\frac{\partial \log p_\theta\left(x,y\right)}{\partial \theta}\right)\right]\right]\end{aligned}$$

Since $\log p_\theta\left(x,y\right) = \log p_\theta\left(y|x\right) + \log p\left(x\right)$ and $p\left(x\right)$ does not depend on $\theta$ then this can be further simplified in:

$$\mathbf{F} = \mathbb{E}_{x\sim p(x)}\left[\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial \theta}\right)^T \left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial \theta}\right)\right]\right]$$

Interestingly, for the usual distributions expressed by neural networks, we can derive an exact expression for the inner expectation. The FIM takes the following simple form as shown by Pascanu and Bengio [2013]:

$$\mathbf{F} = \mathbb{E}_{x\sim q(x)\,x\sim p(x)}\left[\mathbf{J}_\theta\left(x,\theta\right)^T D\left(f_\theta\left(x\right)\right)\mathbf{J}_\theta\left(x,\theta\right)^T\right]$$

The values for $x$ are drawn from the data generating distribution ~~$q$~~ $p$. Similarly to section 3.1.4, the notation $\mathbf{J}_\theta\left(x,\theta\right)^T$ is used for the jacobian of the output of the network (i.e. the probability expressed at a given $x$ : $p\left(y \mid x\right)$), with respect to the parameters. In other words, it measures how much the output of the network $p\left(y|x\right)$ will change for a given $x$ if we change the parameters. For usual loss functions, $D$ is a diagonal matrix with non negative diagonal terms, and depends of the cost function used. For the quadratic loss it is the identity (table 3. II).

### 3.2.4. Approximating the FIM

Similarly to the Hessian, the FIM is difficult to compute because of its size ($n_{parameters} \times n_{parameters}$) and because in general we do not have an expression for $q$ but only samples from a training dataset. As for Newton, we can make the two following approximations:

- A first approximation that we can make is by ignoring the interactions between layers. In this case the FIM takes the form of a block diagonal matrix, where each block is a square matrix which has the size of the parameters of a layer. For a neural network with $n_{layers}$ layers this reduces the FIM into $n_{layers}$ smaller matrices. We will denote by $\mathbf{F}_i$ the block corresponding to layer $i$.
- A second common approximation we make in practice is to use the empirical FIM for a training dataset of $n$ examples $x_i$: $\mathbf{F} = \frac{1}{n} \sum_i \mathbf{J}_\theta (x_i, \theta)^T D (f_\theta (x)) \mathbf{J}_\theta (x_i, \theta)^T$.

## 3.3. Gauss-Newton and Fisher share a very similar structure

### 3.3.1. Relation between the FIM and the GN approximation of the Hessian

We have just shown that the Gauss-Newton of the empirical risk with respect to the parameters, and the Fisher Information Matrix share a similar structure that is composed of the jacobians of the output of the network with respect to the parameters, and a symmetric matrix:

$$\frac{1}{n} \sum_i \mathbf{J}_\theta (x_i, \theta)^T D \left( f_\theta (x_i) \underbrace{, y_i}_{\text{opt}} \right) \mathbf{J}_\theta (x_i, \theta)^T \tag{3.3.1}$$

The main difference is in this symmetric matrix $D (f_\theta (x), y)$. For Fisher methods it does not depend on any true target and it is just an intrinsic property of a neural network, associated with an input distribution. We can thus remove the $y$: $D (f_\theta (x))$. In the case of the GN matrix it depends on the true target $y$ in general, with a notable exception for the quadratic error (table 3. III).

| | Gauss-Newton $D (f_\theta (x), y)$ | Fisher $D (f_\theta (x))$ |
|---|---|---|
| quadratic error | $\mathbf{I}$ | $\mathbf{I}$ |
| cross entropy for binary decision | $\frac{y_i}{(f_\theta(x_i))^2} + \frac{1-y_i}{(1-f_\theta(x_i))^2}$ | $\frac{1}{f_\theta(x)(1-f_\theta(x))}$ |
| cross entropy for multiclass classification | $\text{diag}\left(\frac{y_i}{(f_\theta(x_i))^2}\right)$ | $\text{diag}\left(\frac{1}{f_\theta(x)}\right)$ |

Table 3. III. Expressions for the middle term $D (f_\theta (x), y)$ and $D (f_\theta (x))$ for GN and FIM

This common structure is of great interest as we will show in chapter ?? since we will derive an approximate form that applies to both matrices.

### 3.3.2. An original interpretation from the output of the network

In the cases where $D$ is a diagonal matrix (i.e. cross entropy and quadratic error, see eq 3.3.1 and table 3. III), we can rewrite both GN and FIM matrices applied to an update as a norm in the space of the output of the network:

$$\begin{aligned}
\Delta\theta^* &= \underset{\Delta\theta}{\operatorname{argmin}} \left(\nabla R\right)_\theta^T \Delta\theta + \frac{1}{2}\Delta\theta^T \frac{1}{n}\sum_i \mathbf{J}_\theta\left(x_i,\theta\right)^T D\left(f\left(x_i,\theta\right),y_i\right)\mathbf{J}_\theta\left(x_i,\theta\right)\Delta\theta + \frac{1}{2\lambda}\Delta\theta^T\Delta\theta \\
&= \underset{\Delta\theta}{\operatorname{argmin}} \left(\nabla R\right)_\theta^T \Delta\theta + \frac{1}{2}\frac{1}{n}\sum_i \left\langle D\left(f\left(x_i,\theta\right),y_i\right)\mathbf{J}_\theta\left(x_i,\theta\right)\Delta\theta, \mathbf{J}_\theta\left(x_i,\theta\right)\Delta\theta\right\rangle + \frac{1}{2\lambda}\Delta\theta^T\Delta\theta \\
&= \underset{\Delta\theta}{\operatorname{argmin}} \left(\nabla R\right)_\theta^T \Delta\theta + \frac{1}{2}\frac{1}{n}\sum_i \left\langle D\left(f\left(x_i,\theta\right),y_i\right)\Delta f_\theta\left(x_i,\Delta\theta\right), \Delta f_\theta\left(x_i,\Delta\theta\right)\right\rangle + \frac{1}{2\lambda}\Delta\theta^T\Delta\theta
\end{aligned}$$

We denoted by $\Delta f_\theta\left(x_i,\Delta\theta\right) = \mathbf{J}_\theta\left(x_i,\theta\right)\Delta\theta$ a first order approximation of the change in the value of $f_\theta\left(x_i\right)$ induced by a change $\Delta\theta$ of $\theta$ for example $x_i$. With this decomposition we can understand GN and natural gradient as being a regularizer for each example, using the metrics $D\left(f\left(x_i,\theta\right),y_i\right)$ that depends on the considered example. We regularize for several undesirable effect:

- We ensure that $\Delta f_\theta\left(x_i,\Delta\theta\right)$ cannot take a large value. This distributes the effect of the update evenly between examples, instead of having a large change in $f_\theta\left(x_i\right)$ for a single example, and smaller changes for others.
- We weight this changes using $D\left(f\left(x_i,\theta\right),y_i\right)$. For the cross entropies for instance we observe that this term grows with $\frac{1}{f_\theta(x)}$ (the vector of probabilities of each class). If this vector is not evenly distributed, that is if for one class $t$ we have a larger value of $\left(f_\theta\left(x\right)\right)_t$, all other values will be close to 0, which means that $\left(\frac{1}{f_\theta(x)}\right)_{i\neq t}$ will take a very large value. In this case we put more weight on the examples for which our model is more confident of its prediction.

## 3.4. A cheaper Gauss-Newton matrix for cross-entropy

We now present a computational trick for computing the Gauss-Newton in the case of cross-entropy. Interestingly, in the expression of Gauss-Newton for log losses we have the unexpected equivalence $\left(\frac{\partial^2 l}{\partial f_\theta^2}\right)_{tt} = -\left(\frac{\partial l}{\partial f_\theta}\right)_t^2 \left(\frac{\partial^2 \ell}{\partial f_\theta^2}\right)_{tt} = -\left(\frac{\partial \ell}{\partial f_\theta}\right)_t^2$ for the true class $t$ and $\left(\frac{\partial^2 l}{\partial f_\theta^2}\right)_{ii} = 0 = -\left(\frac{\partial l}{\partial f_\theta}\right)_t^2 \left(\frac{\partial^2 \ell}{\partial f_\theta^2}\right)_{ii} = 0 = -\left(\frac{\partial \ell}{\partial f_\theta}\right)_t^2$ when $i \neq t$:

$$\begin{aligned}
\underline{l}\ell\left(f_\theta\left(x\right),y\right) &= -\sum_i y_i \log\left(\left(f_\theta\left(x\right)\right)_i\right) \\
\left(\frac{\partial}{\partial f_\theta}\underline{l}\ell\left(f_\theta\left(x\right),y\right)\right)_t &= -\frac{1}{\left(f_\theta\left(x\right)\right)_t}
\end{aligned}$$

$$\left(\frac{\partial^2}{\partial f_\theta^2}\underline{l\ell}\left(f_\theta\left(x\right),y\right)\right)_{tt} = \frac{1}{\left(f_\theta\left(x\right)\right)_t^2} = \left(\frac{\partial}{\partial f_\theta}\underline{l\ell}\left(f_\theta\left(x\right),y\right)\right)_t^2$$

The reason is that the second derivative of the log function ($x \longmapsto -\frac{1}{x^2}$) is minus the square of its first derivative ($x \longmapsto \frac{1}{x}$). Getting back to the expression of the GN matrix (here for a single example), we can combine the second derivative with the jacobians and get a simple expression:

$$\begin{aligned} G_f\left(x,\theta\right) &= \mathbf{J}_\theta\left(x,\theta\right)^T \frac{\partial^2}{\partial f^2}\left\{\underline{l\ell}\left(f_\theta\left(x\right),y\right)\right\}\mathbf{J}_\theta\left(x,\theta\right) \\ &= \mathbf{J}_\theta\left(x,\theta\right)^T \left(\underline{\frac{\partial l}{\partial f_\theta}}\frac{\partial \ell}{\partial f_\theta}\right)^{\underset{\sim}{T}} \left(\underline{\frac{\partial l}{\partial f_\theta}}\frac{\partial \ell}{\partial f_\theta}\right)^{\underline{T}}\mathbf{J}_\theta\left(x,\theta\right) \\ &= \nabla_\theta \underline{l\ell}\left(f_\theta\left(x\right),y\right)\nabla_\theta \underline{l\ell}\left(f_\theta\left(x\right),y\right)^T \end{aligned} \tag{3.4.1}$$

This gradient in eq 3.4.1 is the exact same as the gradient used to compute the update in gradient descent. So for no additional cost we get the expression of the GN matrix. Note that we still need to invert it, which is a $O\left(n^3\right)$ operation in the size of the matrix.

This gives an explanation of the outer product metrics mentionned in Ollivier [2013]. To the best of our knowledge this result has not been published before, which is very suprising as it gives a very cheap way of computing the GN matrix.

# Chapter 4

# EXPERIMENTAL SETUP

In order to be able to assess the performance of the ideas and algorithms in the next chapters, we now present our experimental setup.

We present 2 standard tasks. We then contribute a simple method called *biased random search* which improves random search of hyperparameter values, and motivate its use in order to provide a fair comparison of optimization algorithms.

## 4.1. Benchmark tasks

### 4.1.1. A standard benchmark: Autoencoding written digits

We now describe the main benchmark that we will be using in the rest of this document. The dataset MNIST [LeCun et al., 2010] is composed of 60.000 $28 \times 28$ grayscale images of handwritten digits, and the corresponding value of the digit that is represented in the image. For this benchmark, we use an autoencoder (see section 1.4.3) with layer sizes $\{784, 1000, 500, 250, 30, 250, 500, 1000, 784\}$. The autoencoder encodes the input image into a vector of size 30, and then decodes it to reconstruct the original image. We use the reconstruction error $\ell(f(x), y) = \|f(x) - y\|_2^2$. The benchmark consists in minimizing the empirical risk over the train set after a fixed time on the same architecture.

This benchmark has a long history in the neural network optimization litterature [Hinton and Salakhutdinov, 2006, Martens, 2010, Martens and Grosse, 2015, Desjardins et al., 2015]. To assess the performance of an algorithm, we can use 2 metrics: the empirical risk after a given number of iterations of the algorithm, and the empirical risk after a fixed elapsed time for a given computer. In real world tasks, the latter is more useful. It gives a better understanding of the trade-off between a more complex update that takes longer to compute and gives a better improvement, and a fast update that gives a small improvement, but that can be iterated several times in the meantime.

The limits of the benchmark are many. In particular the fact that the state of the art papers in computer vision do not use MLPs and sigmoid activations but rather variants of

mixed convolutional networks and residual connections, and variants of ReLU activations. Another limit is in the use of the quadratic loss. Nonetheless, we still use this benchmark as it is used by several other papers which allows for a fair comparison, and because it is reasonably deep (8 layers) and wide (the biggest weight matrix has size $1000 \times 784$).

#### 4.1.2. A classification task on an image dataset

The second benchmark that we use is a multilayer perceptron with rectifier activation functions, trained to recognize images amongst 10 classes on the CIFAR-10 dataset [Torralba et al., 2008]. It is composed of 60.000 $32 \times 32$ color images, meaning that each image is composed of $32 * 32 * 3 = 3072$ pixels. The network has 8 hidden layers of size 100 making it reasonably deep but still fast to train in order to experiment with many algorithms. We train it using multiclass cross entropy.

This architecture is far from producing state of the art results for this task. In particular, it starts overfitting for a very small number of updates. Instead, we use it to compare optimization algorithms, which means that we are more interested in its performance on the train set. If we were interested in generalization performance, we could add regularization to better condition the optimization problem.

### 4.2. Biased random search

While comparing optimization techniques on real tasks, we found that it was very difficult to provide a fair benchmark, because a slight change in a hyperparameter value can drastically improve or alter its performance. Indeed, with simple hyperparameter adjustments, we were often able to improve the benchmarks reported as state-of-the-art in previous applied optimization papers.

More sophisticated approaches to automatic hyperparameter tuning exist, such as Bayesian optimization (see e.g. Snoek et al. [2012]). While hyperparameter tuning is an active research area on its own, it is not the focus of our work. We just use a simple technique that refines random search, by allocating more ~~ressource~~ ressources to explore regions in the hyperparameter space that are more likely to give a good performance. We now describe this method that we call **biased random search** (algorithm 1), and we validate its performance using a simple experiment.

During the hyperparameter tuning procedure, we create a model of our cost landscape in the space of hyperparameters. As the number of experiments grows, the cost landscape is refined. We use this estimated cost landscape to bias our random search, so that regions of the hyperparameter space that are expected to provide a better result will have higher probability of being explored. In practice, we use a simple 1-nearest neighbor regressor [Altman, 1992] to model the cost landscape. Using the estimated value of the criteria $c_{estimate}$, we decide to keep the sampled value with probability $p$, or otherwise we reject the value and sample a new

| HP tuning procedure | Average | Standard deviation |
|---|---|---|
| Grid search | 27.23 | 0.42 |
| Random search | 27.02 | 0.28 |
| Biased random search | **26.61** | **0.13** |

TABLE 4. I. Final empirical risk obtained after training 100 hyperparameter tuning procedures, each consisting of 100 experiments (lower is better)

one, and so on until we get a value that is not rejected, which will be our next experiment. We can choose the value of $p$ using different heuristics, in practice we use $p = \frac{c_{max} - c_{estimate}}{c_{max} - c_{min}}$ (in this notation, the criteria needs to be minimized) where $c_{max}$ and $c_{min}$ are defined as the current maximum and minimum value that we have obtained so far. This value for $p$ will almost surely reject values that are close to the worst experiments, and almost surely accept values that are close to the best experiments.

**Algorithm 1** Biased random search

**Require:** $\mathcal{M}$ used to model the cost landscape in the space of HP
**Require:** $\mathcal{D}$ the domain of HP that we will explore
$\mathcal{H} \leftarrow [\,]$ ▷ History of explored HP values and corresponding result
**while** not converged **do**
  rejected $\leftarrow$ true
  **while** rejected **do**
    $a \sim U(\mathcal{D})$ ▷ Sample values for HP
    $c_{estimate} \leftarrow \mathcal{M}(\mathcal{H}, a)$ ▷ Estimate $c$ for HP $a$ using history $\mathcal{H}$
    $p \leftarrow \frac{c_{max} - c_{estimate}}{c_{max} - c_{min}}$
    $x \sim U([0, 1])$
    **if** $x < p$ **then**
      rejected $\leftarrow$ false
    **end if**
  **end while**
  $result \leftarrow run(a)$ ▷ Run experiment with HP values $a$
  $\mathcal{H} \leftarrow \mathcal{H} + (a, result)$
**end while**

To assess the performance of biased random search we ran 100 searches of 100 experiments on ~~a simple task~~ the MNIST autoencoder task (section 4.1.1) where we tuned 2 hyperparameters. We observe that ~~it~~ biased random search consistently finds comparable or better results than standard random search. We report the results in ~~the following table and~~ table 4. I and we show a comparison of a single search consisting in 100 experiments in figure 4.1~~(lower is better)~~.

In figure 4.1 we can clearly see that with biased random search the majority of experiments is launched around the region with best performing hyperparameter values.

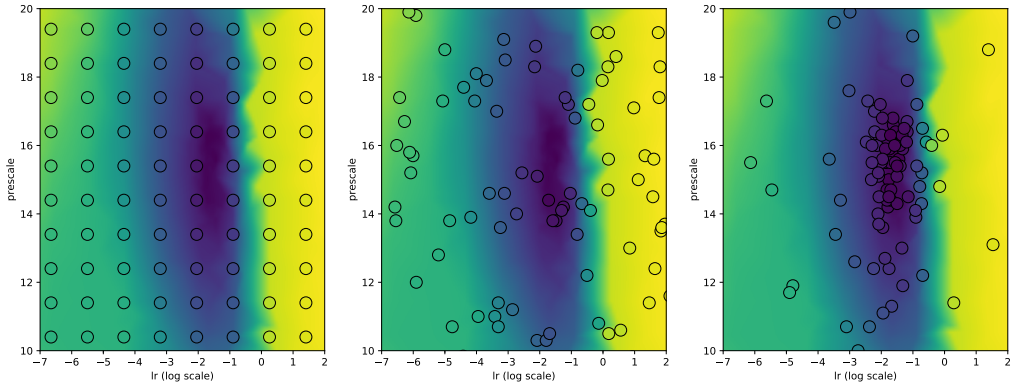


FIGURE 4.1. Comparison of hyperparameter tuning methods. On the left a grid search, in the middle a random search and on the right a biased random search. Each experiment consisted in 100 iterations of SGD from a randomly initialized network (circles). We tune 2 hyperparameters on the x and y axis (what they represent is not relevant here). The color scale represents the final loss attained after a fixed number of iterations. The best experiments are in blue, the worst experiments in yellow.

~~We now describe the main benchmark that we will be using in the rest of this document. The dataset MNIST [LeCun et al., 2010] is composed of 60.000 $28 \times 28$ grayscale images of handwritten digits, and the corresponding value of the digit that is represented in the image. For this benchmark, we use an autoencoder 1.4.3 with layer sizes $\{784, 1000, 500, 250, 30, 250, 500, 1000, 784\}$. The autoencoder encodes the input image into a vector of size 30, and then decodes it to reconstruct the original image. We use the quadratic error $l(f(x), y) = \|f(x) - y\|_2^2$. The benchmark consists in minimizing the empirical risk over the train set after a fixed time on the same architecture.~~

~~This benchmark has a long history in the neural network optimization litterature [Hinton and Salakhutdinov, 2006, Martens, 2010, Martens and Grosse, 2015, Desjardins et al., 2015]. To assess the performance of an algorithm, we can use 2 metrics: the empirical risk after a given number of iterations of the algorithm, and the empirical risk after a fixed elapsed time for a given computer. In real world tasks, the latter is more useful. It gives a better understanding of the trade-off between a more complex update that takes longer to compute and gives a better improvement, and a fast update that gives a small improvement, but that can be iterated several times in the meantime.~~

~~The limits of the benchmark are many. In particular the fact that the state of the art papers in computer vision do not use MLPs and sigmoid activations but rather variants of mixed convolutional networks and residual connections, and variants of ReLU activations. Another limit is in the use of the quadratic loss. Nonetheless, we still use this benchmark~~

as it is used by several other papers which allows for a fair comparison, and because it is reasonably deep (8 layers) and wide (the biggest weight matrix has size $1000 \times 784$).

The second benchmark that we use is a multilayer perceptron with rectifier activation functions, trained to recognize images amongst 10 classes on the CIFAR-10 dataset [Torralba et al., 2008]. It is composed of 60.000 $32 \times 32$ color images, meaning that each image is composed of $32 * 32 * 3 = 3072$ pixels. The network has 8 hidden layers of size 100 making it reasonably deep but still fast to train in order to experiment with many algorithms. We train it using multiclass cross entropy.

This architecture is far from producing state of the art results for this task. In particular, it starts overfitting for a very small number of updates. Instead, we use it to compare optimization algorithms, which means that we are more interested in its performance on the train set. If we were interested in generalization performance, we could add regularization to better condition the optimization problem.

# Chapter 5

# PROOF OF CONCEPT: EVOLUTION OF THE BACKPROPAGATED GRADIENT WHILE UPDATING THE PARAMETERS

In this section, we present a prototype technique to account for the interactions between parameters of different layers while computing updates. While we could not come up with an efficient algorithm to implement this technique, early results show that it could be useful in deep networks.

## 5.1. How is the gradient modified when changing the value of the parameters of a layer

In usual gradient descent, we compute the gradient of the empirical risk with respect to all parameters, then we update all parameters at once. But ~~since~~ it is not really clear that in doing so we will actually decrease the value of the empirical risk. The direction provided by the gradient is locally a descent direction. But how much locally? As we increase the number of parameters, we might need to use an optimal learning rate that is even too small to make any perceptible progress overall.

We experiment with a technique that aims at improving the update directions. It is a modification of the gradient, that can be computed following the same chaining of operations as computing the gradient using forward and backward propagation, but requiring more computation.

We now present the technique, and describe the derivation of this new update direction.

Since we are using backpropagation, then the process of getting the partial derivatives is sequential, that is, we get the derivatives of the top layers first, and afterwards we get the derivatives of the bottom layers. Now suppose that we apply the update for the parameters of the top layers before backpropagating through ~~it. The idea here is to estimate the~~ them. We are now optimizing an updated function. Instead of using the backropagated gradient

that we have obtained so far, we could reestimate the forward and backward pass for this updated function and get a new backpropagated signal. ~~This can also be interpreted as doing gradient descentseparately for each layer, but~~ To a certain extent it could be seen as doing coordinate descent, but instead of optimizing each parameter separately, we group them by layer, and we optimize each layer separately.

What we propose lies somewhere in-between: instead of recomputing the ~~full forward and backward propagation to the target~~ whole forward pass and the backward pass up until the current layer, we ~~just do a single updated backpropagation~~estimate how updating the parameters of the top layers will modify the backpropagated signal.

To illustrate the idea, we focus on the transformation computed by a single layer:

$$\begin{aligned} h_l &= f(a_l) \\ a_l &= W h_{l-1} + b \end{aligned}$$

Using backpropagation through this layer we get the partial derivative of the loss function $\ell$:

$$\begin{aligned} \frac{\partial l}{\partial h_{l-1}} \frac{\partial \ell}{\partial h_{l-1}} &= \frac{\partial l}{\partial f} \frac{\partial f}{\partial a_l} \frac{\partial \ell}{\partial a_l} \frac{\partial a_l}{\partial h_{l-1}} \\ &= \frac{\partial l}{\partial f} \frac{\partial f}{\partial a_l} \frac{\partial \ell}{\partial a_l} W \end{aligned} \tag{5.1.1}$$

This partial derivative is thus a function of $W$ in an explicit way. It is also a function of $W$ and $b$ through the other ~~terms $\frac{\partial l}{\partial f}$ and $\frac{\partial f}{\partial a_l}$~~term $\frac{\partial \ell}{\partial a_l}$. Can we get an update expression for ~~$\frac{\partial l}{\partial h_{l-1}}(W + \Delta W, b + \Delta b)$?~~ $\frac{\partial \ell}{\partial h_{l-1}}(W + \Delta W, b + \Delta b)$? This gradient $\frac{\partial \ell}{\partial h_{l-1}}$ is in turn used for computing the gradient of each preceding (deeper) layer using the chain rule. By obtaining a more accurate value for $\frac{\partial \ell}{\partial h_{l-1}}(W + \Delta W, b + \Delta b)$ we expect to improve all consecutive parameter updates.

## 5.2. A first order update of a first order derivative

~~A simple first order expansion gives a way of getting such an update~~ We now focus on a single layer. Suppose that we update $W \leftarrow W + \Delta W$ and $b \leftarrow b + \Delta b$. We want to estimate $\frac{\partial \ell}{\partial h_{l-1}}(W + \Delta W, b + \Delta b)$, using a first order approximation:

$$\frac{\partial l}{\partial h_{l-1}} \frac{\partial \ell}{\partial h_{l-1}}(W + \Delta W, b + \Delta b) \approx \frac{\partial l}{\partial h_{l-1}} \frac{\partial \ell}{\partial h_{l-1}}(W, b) + \left( \frac{\partial}{\partial \mathrm{vec}(W)} \left\{ \frac{\partial l}{\partial h_{l-1}} \frac{\partial \ell}{\partial h_{l-1}}(W, b) \right\} \mathrm{vec}(\Delta W) \right)^T$$

$$+\left(\frac{\partial}{\partial b}\left\{\frac{\partial l}{\partial h_{l-1}}\frac{\partial \ell}{\partial h_{l-1}}(W,b)\right\}\Delta b\right)^T$$

We used the vec operator in order to have a matrix expression for the second derivative with respect to $W$. Using the expression ~~5.1.1 for $\frac{\partial l}{\partial h_{l-1}}$~~ in eq 5.1.1 for $\frac{\partial \ell}{\partial h_{l-1}}$ we see that it requires deriving ~~3 terms. Indeed the second derivative of the jacobian of the input of the network, with respect to the preactivation $\frac{\partial f}{\partial a_l}$ is too costly to be used in a real problem. We choose not to use it. The remaining~~ 2 ~~derivatives give an update:~~ terms: $\frac{\partial \ell}{\partial a_l}$ and $W$:

$$\frac{\partial l}{\partial h_{l-1}}(W+\Delta W, b+\Delta b)\approx \left(\frac{\partial l}{\partial f}\frac{\partial f}{\partial a_l}\right)^T \Delta W + W^T\left(\frac{\partial f}{\partial a_l}\right)^T\frac{\partial^2 l}{\partial f^2}\frac{\partial f}{\partial a_l}(\Delta W h_{l-1}+\Delta b)$$

$$\begin{aligned}
\frac{\partial}{\partial \mathrm{vec}(W)}\left\{\frac{\partial \ell}{\partial h_{l-1}}(W,b)\right\}\mathrm{vec}(\Delta W) &\approx \frac{\partial}{\partial \mathrm{vec}(W)}\left\{\frac{\partial \ell}{\partial a_l}W\right\}\mathrm{vec}(\Delta W)\\
&\approx \frac{\partial}{\partial \mathrm{vec}(W)}\left\{\mathrm{vec}(W)^T\left(\mathbf{I}\otimes\left(\frac{\partial \ell}{\partial a_l}\right)^T\right)\right\}\mathrm{vec}(\Delta W)\\
&\approx \left(\left(\mathbf{I}\otimes\left(\frac{\partial \ell}{\partial a_l}\right)\right)+W^T\frac{\partial^2 \ell}{\partial a_l^2}\frac{\partial a_l}{\partial \mathrm{vec}(W)}\right)\mathrm{vec}(\Delta W)\\
&\approx \left(\left(\mathbf{I}\otimes\left(\frac{\partial \ell}{\partial a_l}\right)\right)+W^T\frac{\partial^2 \ell}{\partial a_l^2}\left(h_{l-1}^T\otimes\mathbf{I}\right)\right)\mathrm{vec}(\Delta W)\\
&\approx \frac{\partial \ell}{\partial a_l}\Delta W + W^T\frac{\partial^2 \ell}{\partial a_l^2}\Delta W h_{l-1}\\
\frac{\partial}{\partial b}\left\{\frac{\partial \ell}{\partial h_{l-1}}(W,b)\right\}\Delta b &\approx \frac{\partial}{\partial b}\left\{\frac{\partial \ell}{\partial a_l}W\right\}\Delta b\\
&\approx W^T\frac{\partial^2 \ell}{\partial a_l^2}\frac{\partial a_l}{\partial b}\Delta b\\
&\approx W^T\frac{\partial^2 \ell}{\partial a_l^2}\Delta b
\end{aligned}$$

~~In the process, we did 2 approximations: the first one in using a first order expansion, the second one in not~~ Overall we get the following form:

$$\frac{\partial \ell}{\partial h_{l-1}}(W+\Delta W, b+\Delta b) \approx \frac{\partial \ell}{\partial h_{l-1}}(W,b)+\Delta W^T\left(\frac{\partial \ell}{\partial a_l}\right)^T+(\Delta W h_{l-1}+\Delta b)^T\frac{\partial^2 \ell}{\partial a_l^2}W$$

This can be further simplified by using the chain rule for $\frac{\partial \ell}{\partial h_{l-1}}(W, b)$:

$$\begin{aligned}\frac{\partial \ell}{\partial h_{l-1}}(W + \Delta W, b + \Delta b) &\approx W^T \left(\frac{\partial \ell}{\partial a_l}\right)^T + \Delta W^T \left(\frac{\partial \ell}{\partial a_l}\right)^T + (\Delta W h_{l-1} + \Delta b)^T \frac{\partial^2 \ell}{\partial a_l^2} W \\ &= (W + \Delta W)^T \left(\frac{\partial \ell}{\partial a_l}\right)^T + (\Delta W h_{l-1} + \Delta b)^T \frac{\partial^2 \ell}{\partial a_l^2} W\end{aligned}$$

This approximate new backpropagated gradient thus decomposes into 2 terms. The first one $(W + \Delta W)^T \left(\frac{\partial \ell}{\partial a_l}\right)^T$ is very similar to the usual backpropagated gradient but accounts for the updated value of $W$. The second term $(\Delta W h_{l-1} + \Delta b)^T \frac{\partial^2 \ell}{\partial a_l^2} W$ measures a global change in the gradient. It requires using the Hessian of the preactivation. This Hessian estimates how much will the gradient $\frac{\partial \ell}{\partial a_l}$ change when the value of $a_l$ changes, which is exactly what is computed by $\Delta W h_{l-1} + \Delta b$.

The Hessian $\frac{\partial^2 \ell}{\partial a_l^2}$ is currently the limiting factor for this technique to be truly efficient. It must be computed for every example. It has the size of the number of output units for this layer, and must be computed for each example. It is thus smaller compared to the true Hessian that we use in Newton's method, which has the size $n_{parameters} \times n_{parameters}$ and also requires computing a Hessian for each example that is summed in order to get the Hessian of the empirical risk.

This Hessian $\frac{\partial^2 \ell}{\partial a_l^2}$ can also be approximated by Gauss-Newton (see section 3.1.4): $\frac{\partial^2 \ell}{\partial a_l^2} \approx \mathbf{J}_{a_l}{}^T \frac{\partial^2 \ell}{\partial f^2} \mathbf{J}_{a_l}$ where the jacobians $\mathbf{J}_{a_l} = \frac{\partial f}{\partial a_l}$ are the jacobians of the output of the network with respect to the preactivation of the current layer $a_l$. This approximation drastically reduces the computation required during backpropagation, and experimentally proves to be equally efficient as using the true ~~second derivatives, but instead just a part of them. We~~ Hessian.

## 5.3. Updated backpropagation algorithm

Starting from the usual gradient computation, we propose to replace the backpropagation step by backpropagating this updated ~~derivative~~gradient. We call this technique updated backpropagation (UBP), and we describe a simple algorithm that implements it ~~. Note that it requires backpropagating a jacobian of the output of the network, in addition to the usual gradient~~of (algorithm 2).

In addition to backpropagating the ~~loss~~gradient, we must also backpropagate the jacobians $\frac{\partial f}{\partial a_l}$ for each example. This can be done in a similar fashion than the gradient, the main difference being that the jacobians are matrices for each example, whereas the gradients are vectors. The size of the jacobians grows with the size of the output of the network.

**Algorithm 2** Updated backpropagation

**Require:** $\mathcal{D}$ a minibatch of $n$ examples
**Require:** $\lambda$ learning rate

$dh_i \leftarrow \left(\frac{\partial l}{\partial f_i}\right)^T$ ▷ Gradient of the loss w.r.t the output of the NN for examples $i$
$J_i \leftarrow \sqrt{\frac{\partial^2 l}{\partial f^2}}$ ▷ Jacobian of the loss of the NN for examples $i$
**for all** $l \in$ layers from top to bottom **do**
  $da_i \leftarrow \left(\frac{\partial f_l^{(i)}}{\partial a_l^{(i)}}\right)^T dh_i$ ▷ Derivative of the loss w.r.t the preactivation for examples $i$
  $J_i \leftarrow J_i \frac{\partial f_l^{(i)}}{\partial a_l^{(i)}}$ ▷ For elementwise functions $\frac{\partial f_l^{(i)}}{\partial a_l^{(i)}}$ is diagonal
  $\Delta b \leftarrow -\lambda \sum_i^n da_i$ ▷ bias update
  $\Delta W \leftarrow -\lambda \sum_i^n da_i h_{l-1}^T$ ▷ weights update
  $dh_i \leftarrow (W + \Delta W)^T da_i + W^T J_i^T J_i \left(\Delta W h_{l-1}^{(i)} + \Delta b\right)$ ▷ Updated backprop.
  $J_i \leftarrow J_i W$
  $b \leftarrow b + \Delta b$
  $W \leftarrow W + \Delta W$
**end for**

## 5.4. Experiments

We use the autoencoder benchmark to compare the performance of UBP with stochastic gradient descent. Our results are plotted in figure 5.1. In terms of updates, we observe that this method ~~outperforms significantly compared to~~ significantly outperforms SGD (note that this is a logarithmic scale). However it takes 10 times longer to obtain an update using UBP making it unpractical on this task.

## 5.5. Limits of this method

A first obvious limit is that it is very costly to compute the jacobians, and also to store them in memory.

A second limit which can also be very problematic is that similarly to the exploding gradient issue in deep or recurrent networks, we observe that this update can cause the backpropagated gradients to explode. Indeed, the problem resides in the fact that we have to multiply by the square matrix $\left(\frac{\partial f}{\partial a_l}\right)^T \frac{\partial^2 l}{\partial f^2} \frac{\partial f}{\partial a_l} \left(\frac{\partial f}{\partial a_l}\right)^T \frac{\partial^2 \ell}{\partial f^2} \frac{\partial f}{\partial a_l}$ at each layer. This matrix is a measure of how much will the preactivation of the current layer change when we update its value by $(\Delta W h + \Delta b)$. But by use the chain rule, we can see that we will once again multiply by this matrix when we compute the update of the next layer (which is below as backpropagation computes the derivatives backward from the top layer to the bottom layer). In the next layer, this matrix will have the expression $\left(\frac{\partial f}{\partial a_{l-1}}\right)^T \frac{\partial^2 l}{\partial f^2} \frac{\partial f}{\partial a_{l-1}} = \left(\frac{\partial a_l}{\partial a_{l-1}}\right)^T \left(\frac{\partial f}{\partial a_l}\right)^T \frac{\partial^2 l}{\partial f^2} \frac{\partial f}{\partial a_l} \frac{\partial a_l}{\partial a_{l-1}} \left(\frac{\partial f}{\partial a_{l-1}}\right)^T \frac{\partial^2 \ell}{\partial f^2} \frac{\partial f}{\partial a_{l-1}} = \left(\frac{\partial a_l}{\partial a_{l-1}}\right)^T \left(\frac{\partial f}{\partial a_l}\right)^T \frac{\partial^2 \ell}{\partial f^2} \frac{\partial f}{\partial a_l} \frac{\partial a_l}{\partial a_{l-1}}$, so it will contain the former matrix. In this case, repeating this process multiple times will

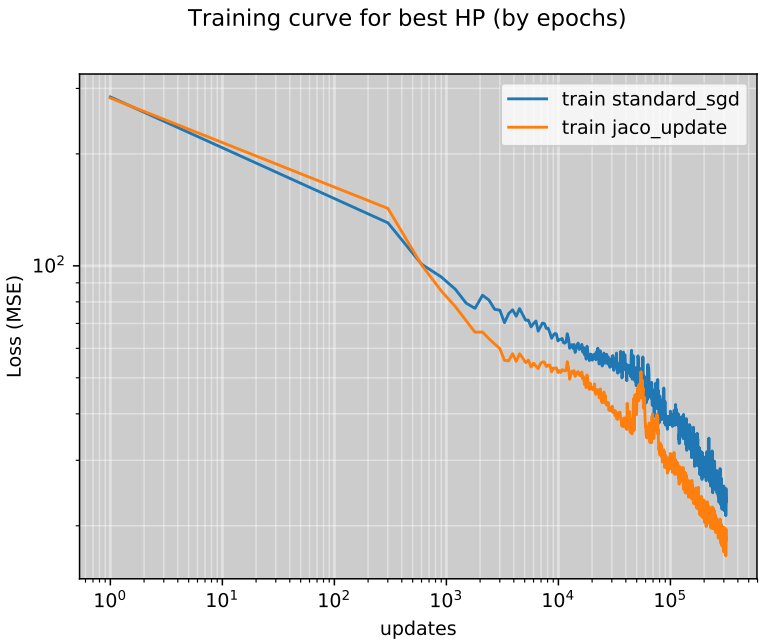


FIGURE 5.1. Training error for standard backpropagation (in blue) and updated backpropagation (in orange)

do something like a power method, which will either explode or vanish depending on the biggest eigenvalue of $\left(\frac{\partial f}{\partial a_{l-1}}\right)^T \frac{\partial^2 l}{\partial f^2} \frac{\partial f}{\partial a_{l-1}} \sim \left(\frac{\partial f}{\partial a_{l-1}}\right)^T \frac{\partial^2 \ell}{\partial f^2} \frac{\partial f}{\partial a_{l-1}}$.

## 5.6. CONCLUSION

This current implementation is certainly not statisfying as a way to accelerate optimization, because of its computational cost. Instead we just see it as a proof of concept that it is possible to act on the backpropagated signal in order to improve it. There are probably more efficient ways of doing similar things, which could improve training very deep networks, where it could account for the interactions between layers that are separated by several others.

# Chapter 6

# FACTORIZED SECOND ORDER

The expressions that we obtained for the FIM and the GN so far are generic in the sense that they could be applied to any model and any empirical risk composed of a sum of terms. We will now exploit the very particular structure of neural networks, to obtain a better understanding of how to apply these techniques for real tasks.

In an unconvenional way, we will start by presenting the local criterion that we introduce, which allowed us to get competitive results, and then we will introduce more general expressions and algorithms.

## 6.1. A local criterion and the importance of the covariance of inputs in a layer

### 6.1.1. Derivation of a new update

Neural networks are usually trained using gradients computed all the way from the loss function to the parameters. Inspired by target propagation [Bengio, 2014, Lee et al., 2015], we explored an alternative which consists in replacing the last step of the backpropagation algorithm: the one of finding updates to the parameters given a derivative on the preactivations. In figure 2.1 (page 15) we keep the usual computation for backpropagation (red lines) and replace the part in green.

We now focus on a single layer $h_l = f_l(a_l) = f_l(W_l h_{l-1} + b_l)$ and from now on we will drop the subscript $l$ for brevity and write $h' = f(a) = f(Wh + b)$. The gradients on the preactivation are given by $\nabla_a l = \left(\frac{\partial l}{\partial a}\right)^T \nabla_h \ell = \left(\frac{\partial \ell}{\partial h}\right)^T$ as usual. We formulate our local criterion as finding updates $\Delta W^*, \Delta b^*$ so that in expectation we will match the opposite gradients of preactivations times a learning rate $-\lambda \nabla_a l - \lambda \nabla_a \ell$. We call this optimization problem "local" in the sense that it is formulated locally to a single layer. We formulate our criterion as:

$$\begin{aligned}
\Delta W^*, \Delta b^* &= \operatorname{argmin}_{\Delta W, \Delta b} \frac{1}{n} \sum_i \left\| \Delta W h^{(i)} + \Delta b - \left( -\lambda \nabla_{a^{(i)}} \ell \right) \right\|_2^2 + \epsilon \left\| \Delta W \right\|_2^2 \\
&\approx \operatorname{argmin}_{\Delta W, \Delta b} \frac{1}{n} \sum_i \left\| \Delta W h^{(i)} + \Delta b + \lambda \nabla_{a^{(i)}} \ell \right\|_2^2 + \epsilon \left\| \Delta W \right\|_2^2 \\
&= \operatorname{argmin}_{\Delta W, \Delta b} \ell_C \left( \Delta W, \Delta b \right)
\end{aligned}$$

We denoted by $\ell_C \left( \Delta W, \Delta b \right)$ this new local criterion, to be solved for each layer. Instead of using gradient descent to find the optimal values for $\Delta W^*, \Delta b^*$, we directly solve this expression by taking derivatives with respect to $\Delta W$ and $\Delta b$ and setting them to 0:

$$\begin{aligned}
\nabla_{\Delta W} \ell_C &\approx \frac{1}{n} \sum_i \left( \Delta W h^{(i)} + \Delta b + \lambda \nabla_{a^{(i)}} \ell \right) h^{(i)T} + \epsilon \Delta W \\
&\approx \Delta b \frac{1}{n} \sum_i h^{(i)T} + \epsilon \Delta W + \Delta W \frac{1}{n} \sum_i h^{(i)} h^{(i)T} + \frac{\lambda}{n} \sum_i \left( \nabla_{a^{(i)}} \ell \right) h^{(i)T} \qquad (6.1.1) \\
\nabla_{\Delta b} \ell_C &\quad \frac{1}{n} \sum_i \left( \Delta W h^{(i)} + \Delta b + \lambda \nabla_{a^{(i)}} \ell \right) \\
&\approx \Delta b + \Delta W \frac{1}{n} \sum_i h^{(i)} + \frac{\lambda}{n} \sum_i \nabla_{a^{(i)}} \ell \qquad (6.1.2)
\end{aligned}$$

By solving $\nabla_{\Delta b} \ell_C = 0$ for $\Delta b$ in eq 6.1.2 we get:

$$\Delta b^* \approx -\frac{\lambda}{n} \sum_i \nabla_{a^{(i)}} \ell - \Delta W \frac{1}{n} \sum_i h^{(i)} \qquad (6.1.3)$$

By solving $\nabla_{\Delta W} \ell_C = 0$ for $\Delta W$ in eq 6.1.1 and replacing $\Delta b$ using eq 6.1.3 we get:

$$\frac{\partial l_C}{\partial \Delta W} = \Delta W^* \left( \frac{1}{n} \sum_i \Delta W h h^{(i)} + \Delta b + \lambda \nabla_{a^{(i)}} l h^{(i)T} + \epsilon \Delta W = \Delta b - \frac{1}{n} \sum_i h^{(i)T} + \epsilon \Delta W + \Delta W^{(i)} \frac{1}{n} \sum_i h^{(i)} h^{(i)T} + \epsilon$$

By putting both expressions together and simplifying ~~for $\Delta W^*$ we get~~the covariances we obtain:

$$\begin{aligned}
\Delta W^* \left( C + \epsilon \mathbf{I} \right) &= -\frac{\lambda}{n} \sum_i \left( \nabla_{a^{(i)}} l - \ell \right) \left( h^{(i)} - \frac{1}{n} \sum_i h^{(i)} \right)^T \\
\Delta b^* &= -\frac{\lambda}{n} \sum_i \nabla_{a^{(i)}} l - \ell - \frac{1}{n} \Delta W^* \sum_i h^{(i)}
\end{aligned}$$

We denoted by $C = \frac{1}{n}\sum_i \left(h^{(i)} - \frac{1}{n}\sum_i h^{(i)}\right)\left(h^{(i)} - \frac{1}{n}\sum_i h^{(i)}\right)^T$ the covariance matrix of the activation of the previous layer. This expressions can be solved by inverting the square matrix $(C + \epsilon \mathbf{I})$.

### 6.1.2. Comparison with standard SGD

The updates for standard SGD are $\Delta_{SGD} b = -\frac{\lambda}{n}\sum_i \nabla_{a^{(i)}} l$ and $\Delta_{SGD} W = -\frac{\lambda}{n}\sum_i (\nabla_{a^{(i)}} l)\, h^{(i)T}$ $\Delta_{SGD} b = -$ and $\Delta_{SGD} W = -\frac{\lambda}{n}\sum_i (\nabla_{a^{(i)}} \ell)\, h^{(i)T}$.

The **update for** $b$ gets a new term $-\frac{1}{n}\Delta W \sum_i h^{(i)}$ that permits taking into account the update of $W$. In practice, we found that it did not change much as $\Delta W$ is typically at least one order of magnitude smaller than $\frac{\lambda}{n}\sum_i \nabla_{a^{(i)}} l \frac{\lambda}{n}\sum_i \nabla_{a^{(i)}} \ell$.

The **update for** $W$ is different in 2 ways. First, it is also scaled using the inverse covariance matrix of the input $C^{-1}$. Secondly, it is centered since we substract the expectation of $h$. This is related to an old well used trick [LeCun et al., 1998, Schraudolph, 2012].

### 6.1.3. What is behind this local criteria

This new update is somewhere between usual gradient descent, and something that is inspired from target propagation. From a theoretical point of view it is not yet clear why it would provide sensible updates. Also surprising is the effectiveness of these new updates as we will see in experimental section. In the following sections, we will show that it is actually linked to second order methods applied to the particular structure of neural networks.

## 6.2. Decomposition using the Kronecker product

In this section, we will show a convenient factorization of the Gauss-Newton approximation of the Hessian, that was first applied to the Fisher Information Matrix in the litterature [Martens and Grosse, 2015]. To this end, we will use an operation called the Kronecker product that permits giving simple expressions for the GN matrix. For 2 matrices $A$ of size $m \times n$ and $B$ of size $p \times q$ it produces a new matrix $A \otimes B$ of size $mp \times nq$ defined by:

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

Its most interesting property in the context of neural networks is its relationship with the *vec* operation, that "flattens" a matrix into a vector. It is of great use for 2nd order, because the weight matrices can be vectorized using *vec*, to give matrix expressions for the Hessian, which otherwise could not be written. We will make use of the property:

$$vec\left(AXB\right) \quad = \quad \left(B^T \otimes A\right) vec\left(X\right)$$

Getting back to the expression for the Gauss-Newton matrix derived in section 3.1.4, we use the block diagonal approximation and focus on a single layer defined by the linear transformation $a = Wh + b$ and the nonlinearity $h' = f\left(a\right)$. We start from the jacobian of the output of the network, with respect to the output of the linear transformation $a$, denoted by $\mathbf{J}_a$. From this jacobian computed by backpropagation, we can get the jacobian with respect to the parameters of the layer by making use of the chain rule $\mathbf{J}_\theta = \mathbf{J}_a \mathbf{J}_\theta^a$. We use the notation $\mathbf{J}_\theta^a$ for the jacobian of $a$ with respect to $\theta$. In order to get an expression for this jacobian, we now make use of the $vec$ operator to transform $W$ into a vector:

$$\begin{aligned} a &= vec\left(a\right) \\ &= vec\left(Wh\right) + b \\ &= \left(h^T \otimes \mathbf{I}\right) vec\left(W\right) + b \end{aligned}$$

$\mathbf{I}$ is the identity, of the same size as $a$, that is the output size of the layer. We can now give an expression for $\mathbf{J}_{vec(W)}^a$ and $\mathbf{J}_b^a$:

$$\begin{aligned} \mathbf{J}_b^a &= \mathbf{I} \\ \mathbf{J}_{vec(W)}^a &= h^T \otimes \mathbf{I} \end{aligned}$$

Or, if we stack $vec\left(W\right)$ and $b$ in a vector $\theta$:

$$\mathbf{J}_\theta^a \quad = \quad \begin{pmatrix} h^T & 1 \end{pmatrix} \otimes \mathbf{I}$$

And finally by the chain rule:

$$\begin{aligned} \mathbf{J}_\theta &= \underbrace{\mathbf{J}_a}\left(\begin{pmatrix} h^T & 1 \end{pmatrix} \underbrace{\otimes \mathbf{I}}\right) \\ &= \left(\underbrace{1 \otimes \mathbf{J}_a}\right)\left(\begin{pmatrix} h^T & 1 \end{pmatrix} \underbrace{\otimes \mathbf{I}}\right) \\ &= \begin{pmatrix} h^T & 1 \end{pmatrix} \otimes \mathbf{J}_a \end{aligned} \tag{6.2.1}$$

where $\begin{pmatrix} h^T & 1 \end{pmatrix}$ is the concatenation of the row vector $h^T$ and 1.

This jacobian is a first order measure of how much the output of the network will change if we change the values of the parameters of this layer, for a single example. Let us now recall the expression of the GN matrix $G_f = \frac{1}{n}\sum_i \mathbf{J}_\theta\left(x_i\right)^T D\left(x_i\right) \mathbf{J}_\theta\left(x_i\right)$ from section 3.1.4.

We can rewrite this expression using the factorization in eq 6.2.1:

$$\begin{aligned} G_f &= \frac{1}{n}\sum_i \left[\begin{pmatrix} h_i^T & 1 \end{pmatrix} \otimes \mathbf{J}_{a_i}\right]^T D\left(x_i\right) \left[\begin{pmatrix} h_i^T & 1 \end{pmatrix} \otimes \mathbf{J}_{a_i}\right] \\ &= \frac{1}{n}\sum_i \begin{pmatrix} h_i^T & 1 \end{pmatrix}^T \begin{pmatrix} h_i^T & 1 \end{pmatrix} \otimes \left(\mathbf{J}_{a_i}^T D\left(x_i\right) \mathbf{J}_{a_i}\right) \\ &= \frac{1}{n}\sum_i \begin{pmatrix} h_i h_i^T & h_i \\ h_i^T & 1 \end{pmatrix} \otimes \left(\mathbf{J}_{a_i}^T D\left(x_i\right) \mathbf{J}_{a_i}\right) \end{aligned} \tag{6.2.2}$$

We used the property that $(A \otimes B)(C \otimes D) = AC \otimes BD$ when the sizes of the matrices $A, B, C, D$ match. This factorization is interesting because it separates the GN matrix into a contribution from the backpropagated jacobian (red arrow in figure 2.1 page 15), and a part that only uses the forward statistics and that is local to a layer. While these 2 contributions are clearly factorized for a single example, it is not clear whether the resulting sum can still be factorized using a kronecker product. As we will show in the next section, similar factorizations were exploited in KFAC [Martens and Grosse, 2015] and Natural Neural Networks [Desjardins et al., 2015] to build efficient optimization algorithms. Note that in this formulation in eq 6.2.2 we contribute an explicit distinction between the weight matrix $W$ and the bias $b$ whereas in previous work authors usually put all parameters together in a weight matrix with an extra column. As we will see in future sections, this is key to understanding the role of centering the updates.

### 6.2.1. Decomposition into 2 smaller matrices

In second order algorithms, inverting the Hessian matrix is often the limiting factor as its computational cost is $O(n^3)$. The Kronecker product ~~as~~ has the pleasing property that it turns the inversion of a big matrix into inverting 2 smaller matrices since $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. In our case, if such a decomposition existed we would reduce the computational cost from $O(n_{in}^3 n_{out}^3)$ to $O(n_{in}^3) + O(n_{out}^3)$.

Unfortunately, we can not write the GN matrix nor the FIM using 2 matrices because it is a sum of Kronecker products, so we aim at finding approximate factorizations that will have the required form.

### 6.2.2. Focus on the covariance part of the decomposition

We now suppose that we can use the following approximation:

$$\frac{1}{n}\sum_i \begin{pmatrix} h_i h_i^T & h_i \\ h_i^T & 1 \end{pmatrix} \otimes \left(\mathbf{J}_{a_i}^T D\left(x_i\right) \mathbf{J}_{a_i}\right) \approx \frac{1}{n}\sum_i \begin{pmatrix} h_i h_i^T & h_i \\ h_i^T & 1 \end{pmatrix} \otimes \alpha\mathbf{I} = G_{in} \tag{6.2.3}$$

Here $\alpha$ is the same for all examples, it does not depend on $i$. This approximation means that we ignore the interactions between the output preactivations. Instead we just focus on some statistics of the activations of the current layer.

Looking back at the use we will make of this preconditioner, ~~we can interpret it as penalizing an update~~ namely $\Delta\theta = -\lambda G_{in}^{-1}\nabla_\theta R$ or equivalently $G_{in}\Delta\theta = -\lambda\nabla_\theta R$, we can observe that this will penalize an update direction of $\theta$ if the corresponding activation has a high variance, as measured by $h_i h_i^T$. This makes sense since in this case changing the value here will change the next forward propagated signal more that if the variance of the corresponding activation were lower. This would result in a bigger expected change in the output.

The left part $A = \frac{1}{n}\sum_i \begin{pmatrix} h_i h_i^T & h_i \\ h_i^T & 1 \end{pmatrix}$ corresponds to some statistics on the input of the considered layer. It has the size $(n_{in}+1)\times(n_{in}+1)$ with the line/column corresponding to the bias. We will now derive the update that corresponds to using this matrix $G_{in}$ as a preconditioner:

We need to invert $A$. This matrix can be inverted blockwise. We denote by $C = \frac{1}{n}\sum_i \left(h_i - \frac{1}{n}\sum_j h_j\right)\left(h_i - \frac{1}{n}\sum_j h_j\right)^T$ the covariance matrix of the input vector of the linear layer. We get the inverse:

$$\begin{pmatrix} \frac{1}{n}\sum_i h_i h_i^T & \frac{1}{n}\sum_i h_i \\ \frac{1}{n}\sum_i h_i^T & 1 \end{pmatrix}^{-1} = \begin{pmatrix} C^{-1} & -C^{-1}\frac{1}{n}\sum_i h_i \\ -\frac{1}{n}\sum_i h_i^T C^{-1} & 1 + \frac{1}{n}\sum_i h_i^T C^{-1}\frac{1}{n}\sum_i h_i \end{pmatrix}$$

Applying this preconditioner to a gradient update we can get a new update for the weight matrix and the bias. Let us first recall the gradient for a minibatch of examples. In order to be able to use it with our preconditioner we put the parameters into a vector $\theta = \begin{pmatrix} \mathrm{vec}\,(W)^T & b^T \end{pmatrix}^T$:

$$\nabla_\theta R = -\frac{\lambda}{n}\sum_j \begin{pmatrix} h_j \\ 1 \end{pmatrix} \otimes \nabla_{a_j} \ell$$

The corresponding update is a Newton step using the approximate Hessian (see eq 3.1.1 page 22):

$$G_{in}^{-1}\nabla_\theta R = \frac{1}{n}\sum_j \begin{pmatrix} C^{-1} & -C^{-1}\frac{1}{n}\sum_i h_i \\ -\frac{1}{n}\sum_i h_i^T C^{-1} & 1 + \frac{1}{n}\sum_i h_i^T C^{-1}\frac{1}{n}\sum_i h_i \end{pmatrix} \begin{pmatrix} h_j \\ 1 \end{pmatrix} \otimes \frac{-\lambda}{\alpha}\nabla_{a_j}\ell$$

From this expression we can write the update for $W$:

$$\begin{aligned}\Delta \text{vec}\,(W) &= C^{-1}\frac{-\lambda}{n}\sum_j h_j \otimes \frac{1}{\alpha}\nabla_{a_j} l - C\ell - C^{-1}\frac{1}{n}\sum_i h_i \otimes \frac{-\lambda}{n}\sum_j \frac{1}{\alpha}\nabla_{a_j} l\ell \\ &= \frac{1}{\alpha}C^{-1}\frac{-\lambda}{n}\sum_j \left(h_j - \frac{1}{n}\sum_i h_i\right) \otimes \nabla_{a_j} l\ell \\ &\boxed{\Delta W = \frac{-\lambda}{\alpha}\frac{1}{n}\sum_j \nabla_{a_j}\ell \left(h_j - \frac{1}{n}\sum_i h_i\right)^T C^{-1}} \qquad (6.2.4)\end{aligned}$$

And the update for $b$:

$$\begin{aligned}\Delta b &= -\frac{-\lambda}{n}\sum_j \frac{1}{n}\sum_i h_i^T C^{-1} h_j \frac{1}{\alpha}\nabla_{a_j} l\ell + \left(1 + \frac{1}{n}\sum_i h_i^T C^{-1}\frac{1}{n}\sum_i h_i\right)\frac{-\lambda}{n}\sum_j \frac{1}{\alpha}\nabla_{a_j} l\ell \\ &= \frac{-\lambda}{n}\sum_j \frac{1}{\alpha}\nabla_{a_j} l\ell + \frac{\lambda}{n}\sum_j \frac{1}{n}\sum_i h_i^T C^{-1}\left(h_j - \frac{1}{n}\sum_i h_i\right)\frac{1}{\alpha}\nabla_{a_j} l\ell \\ &\boxed{\Delta b = -\frac{\lambda}{\alpha}\frac{1}{n}\sum_j \nabla_{a_j}\ell - \Delta W \frac{1}{n}\sum_i h_i} \qquad (6.2.5)\end{aligned}$$

Using an argument based on second order methods, we thus get back to the very same update as in eq 6.1.1.

## 6.3. Algorithms

We now present 2 algorithms. The first one is very simple and just aims at isolating the centering trick, in order to assess how much of the gain of performance comes from just centering the update, and how much comes from the whole covariance. The other one is the full update that we just derived.

### 6.3.1. Centered gradient descent

Following the update for $W$ derived in eq 6.2.4, we simply replace the usual update for the weight matrices by a centered version ~~$\Delta W = \frac{1}{n}\sum_j \nabla_{a_j} l \left(h_j - \overbrace{\frac{1}{n}\sum_i h_i}^{\text{centering}}\right)^T$~~ $\Delta W = \frac{1}{n}\sum_j \nabla_{a_j}\ell \left(h_j - \overbrace{\frac{1}{n}\sum_i h_i}^{\text{centering}}\right)^T$

The gradient, as well as the inner expectation, are computed using a minibatch. Some ~~author~~ authors refer to a very similar idea as *mean-only batch normalization* [Salimans and Kingma, 2016] where the value of $h$ is replaced by $h - \mathbb{E}[h]$ in the forward pass, with the expectation being computed using a mini-batch. The difference here is that we do not reparametrize the forward propagation, instead we just follow a slightly different direction which is not the gradient but a centered gradient, as suggested by eq 6.2.4.

**Algorithm 3** Centered gradient descent

**while** not converged **do**
  Sample a minibatch $\mathcal{D}$
  **for** all layers **do**
    $\Delta_{a_i} \leftarrow -\nabla_{a_i} l\left(f\left(x_i\right), y_i\right) \forall i \in \mathcal{D}$
    $b \leftarrow b + \lambda \frac{1}{n} \sum_i \Delta_{a_i}$
    $W \leftarrow W + \lambda \frac{1}{n} \sum_i \Delta_{a_i} \left(h_i - \frac{1}{n} \sum_j h_j\right)^T$
  **end for**
**end while**

### 6.3.2. Amortized covariance preconditioner

In the updates derived from the covariance (for $b$ eq 6.2.5 and for $W$ eq 6.2.4), we require an inverse of the matrix $C = \frac{1}{n} \sum_i \left(h_i - \frac{1}{n} \sum_j h_j\right) \left(h_i - \frac{1}{n} \sum_j h_j\right)^T$. This matrix has the size of the input of a layer $n_{in}$. While it is smaller than the full GN or FIM for a single layer of size $(n_{in} + 1) \times n_{out}$, it is still not very efficient to estimate the inverse at each iteration. Meanwhile, these statistics do not change much between iterations so a natural idea is to amortize the cost of inversion over several updates.

A question remains for the choice of $\alpha$ that we used to approximate the real GN matrix in eq 6.2.3. We adopt two approaches. The first one consists in treating it as a fixed value, so it is a hyperparameter that we tune using our biased random search (section 4.2). The second one is a very experimental heuristic, which consists in taking the maximum value of the squarred gradient ~~$\alpha = \max_{i \in \text{minibatch}, j \leq n_{out}} (\nabla_{a_i} l)^2_j$~~ $\alpha = \max_{i \in \text{minibatch}, j \leq n_{out}} (\nabla_{a_i} \ell)^2_j$. This gives a different value for each layer, and also different for each minibatch. We found it worked very well experimentally, and we justify it as being a rough heuristic estimate of the curvature of the empirical risk, with respect to the output of the layer.

For numerical stability and to account for the imprecision of $C$ between two estimates, we use Tikhonov regularization (section 3.1.3). This adds a scalar value $\epsilon$ to the diagonal of our approximate GN, which in this case is scaled by $\frac{1}{\alpha}$:

$$\frac{1}{n} \sum_i \begin{pmatrix} h_i h_i^T & h_i \\ h_i^T & 1 \end{pmatrix} \otimes \alpha \mathbf{I}_{out} + \epsilon \mathbf{I} = \frac{1}{n} \sum_i \begin{pmatrix} h_i h_i^T + \frac{\epsilon}{\alpha} \mathbf{I} & h_i \\ h_i^T & 1 + \frac{\epsilon}{\alpha} \end{pmatrix} \otimes \alpha \mathbf{I}_{out}$$

and the updated covariance matrix becomes:

$$C = \frac{1}{n} \sum_i \left(h_i - \frac{1}{n} \sum_j h_j\right) \left(h_i - \frac{1}{n} \sum_j h_j\right)^T + \frac{\epsilon}{\alpha} \mathbf{I}$$

**Algorithm 4** Amortized covariance preconditioner (ACP)

**Require:** **N** estimate statistics every **N** minibatches

$n_{updates} \leftarrow 0$
**while** not converged **do**
  **if** $n_{updates} \mod \mathbf{N} = 0$ **then** ▷ Amortization
    Sample a minibatch $\mathcal{D}$ and compute forward pass
    **for** each layer $j$ **do**
      $C^{(j)} \leftarrow \text{cov}_{\mathcal{D}}\left(h^{(j)}, h^{(j)}\right)$
      $inv\_C^{(j)} \leftarrow \text{inverse}\left(C^{(j)} + \frac{\epsilon}{\alpha}\mathbf{I}\right)$
    **end for**
  **end if**
  Sample a minibatch $\mathcal{D}$ and compute forward pass
  **for** each layer $j$ **do**
    $\Delta_{a_i^{(j)}} \leftarrow -\nabla_{a_i^{(j)}} \ell\left(f\left(x_i\right), y_i\right) \; \forall i \in \mathcal{D}$
    $b \leftarrow b + \frac{\lambda}{\alpha}\frac{1}{n}\sum_{i\in\mathcal{D}} \Delta_{a_i^{(j)}}$ ▷ eq 6.2.5
    $W \leftarrow W + \frac{\lambda}{\alpha}\frac{1}{n}\sum_{i\in\mathcal{D}} \Delta_{a_i^{(j)}} \left(h_i^{(j)} - \frac{1}{n}\sum_{k\in\mathcal{D}} h_k^{(j)}\right)^T inv\_C^{(j)}$ ▷ eq 6.2.4
  **end for**
  $n_{updates} \leftarrow n_{updates} + 1$
**end while**

## 6.4. Other related approximate second order algorithms

The 2 following techniques have been proposed using the same factorization of the FIM that we wrote in eq 6.2.2. In addition to their factorization we introduced the explicit separation of weight matrix and bias, which we use in the following notations.

### 6.4.1. KFAC

~~KFAC [Martens and Grosse, 2015] proposed to split~~ Kronecker Factored Approximate Curvature (KFAC) [Martens and Grosse, 2015] is another factorization where the sum of Kronecker products ~~into~~ is approximated by a product of sums:

$$\mathbf{F} \approx \mathbb{E}\left[\begin{pmatrix} hh^T & h \\ h^T & 1 \end{pmatrix}\right] \otimes \mathbb{E}\left[\left(\boldsymbol{J}_{\boldsymbol{y}}^a\right)^T D\left(\boldsymbol{y}\right) \boldsymbol{J}_{\boldsymbol{y}}^a\right]$$

The Kronecker product has the nice property that for 2 invertible square matrices $A$ and $B$, $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$. It follows that inverting the FIM now requires inverting 2 smaller matrices. As for the approximation we made in eq 6.2.3 section 6.2.2, we lose the coupling between both parts of the kronecker product of each example in the FIM (see eq 6.2.2 page 49). In our experiments we found a comparable performance between KFAC and ACP.

KFAC has been introduced for the natural gradient, but as we ~~shown~~ showed in the previous sections, it can also be adapted to Gauss-Newton.

### 6.4.2. Natural Neural Networks

Natural neural networks [Desjardins et al., 2015] exploit the same factorization by focusing on the input covariance part of each layer. They propose a reparametrization that makes $\mathbb{E}\left[hh^T\right]$ equal the identity. They also notice that in order for their method to work well, they have to use the centering trick. To this view, they change the original linear transformation $a = Wh + b$ to become:

$$a = VU(h - \mu) + d$$

$V$ is the new weight matrix and $d$ are the new biases. $\mu = \mathbb{E}[h]$ is the mean value for $h$ and $U$ is the square root of the inverse covariance of $h$, defined by $U^2 = \left(\mathbb{E}\left[(h-\mu)(h-\mu)^T\right]\right)^{-1}$, denoted by $U = \left(\mathbb{E}\left[(h-\mu)(h-\mu)^T\right]\right)^{-\frac{1}{2}}$. $U$ and $\mu$ are not trained using gradient descent but instead they are estimated using data from the training set.

The new parameters $V$ and $d$ are trained using gradient descent. We will denote by $h_e = U(h - \mu)$ the new "effective" input to the linear transformation induced by the weight matrix $V$. Let us first remark that $\mathbb{E}[h_e] = U(\mathbb{E}[h] - \mu) = U(\mu - \mu) = 0$, so the reparametrized input is centered on average. A second remark is that $\mathbb{E}\left[h_e h_e^T\right] = U\mathbb{E}\left[(h-\mu)(h-\mu)^T\right]U^T = \mathbf{I}$. By construction $U$ cancels out the covariance. Wrapping everything together we thus have the desired property that:

$$\begin{aligned}\mathbb{E}\left[\begin{pmatrix} h_e h_e^T & h_e \\ h_e^T & 1 \end{pmatrix}\right] &= \begin{pmatrix} \mathbb{E}\left[h_e h_e^T\right] & \mathbb{E}[h_e] \\ \mathbb{E}\left[h_e^T\right] & 1 \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{I} & 0 \\ 0 & 1 \end{pmatrix}\end{aligned}$$

The approximate FIM for the reparametrization thus has a better form. In this case the natural gradient update will be closer to the usual gradient update.

## 6.5. Experiments

### 6.5.1. Centering tricks

We compare our algorithms using the autoencoder benchmark presented in section 4.1.1. We ran all experiments on the same architecture using a Titan Black GPU.

For each experiment we plot the expected loss on the train set, and on a test set that we did not use for learning (figure 6.1). To assess the practical performance, the x-axis represents the actual time spent on each experiment. We selected the best hyperparameters using biased random search, and we only plot the best experiment. For the test experiments,

we also only plot the best result for each technique. We ran each experiment for $3 \times 10^5$ updates.

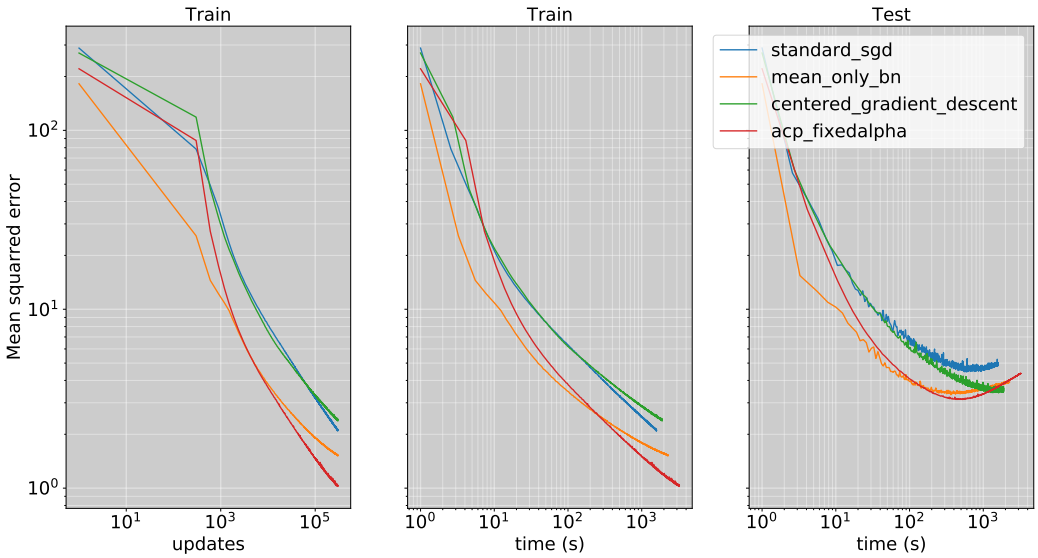


FIGURE 6.1. Comparison of centering methods on MNIST autoencoder: blue: standard SGD with fixed learning rate; orange: mean-only batch norm; green: centered updates as presented in section 6.3.1; red: ACP with fixed $\alpha$ as presented in section 6.3.2.

### 6.5.2. Comparison of 2nd order approximate methods

In the second experiment, we compare all second order approximations to a baseline using batch normalization on the autoencoder on MNIST. For all experiments we use an amortization factor of 100, that is we update the statistics and invert the corresponding matrices every 100 updates. We use a minibatch size of 200 for each update as well as for the statistics. See figure 6.2.

In a third experiment, we compare all second order approximate methods on the classification task on CIFAR10. Similarly to the autoencoder, we use a minibatch size of 200, and we recompute the inverse statistics every 100 updates. As all methods are able to overfit very rapidly, we also report the error rate on both the train set and a separated test set. See figure 6.3.

### 6.5.3. Interpretation and conclusions

6.5.3.1. *How do all second order methods compare ?*

In terms of updates, our best performer is the natural gradient approximated by KFAC on both experiments. But in the case of the autoencoder, we see that because it is faster to

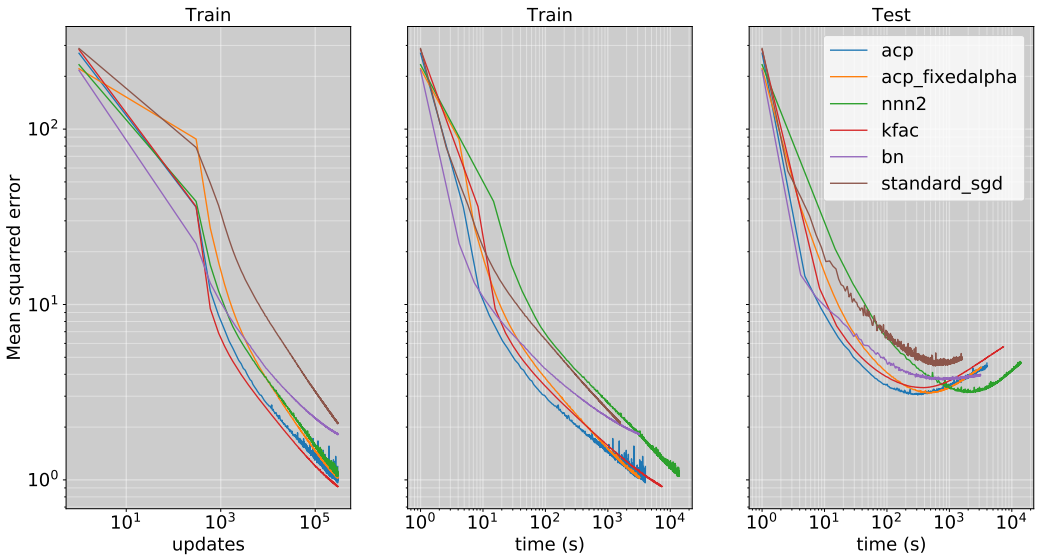


FIGURE 6.2. Comparison of second order methods on MNIST autoencoder: blue: ACP with fixed $\alpha$ as presented in section 6.3.2; orange: ACP with heuristic for $\alpha$ as presented in section 6.3.2; green: Natural Neural Network; red: KFAC; purple: batch norm with standard SGD; brown: standard SGD

compute only the forward statistics, ACP and KFAC compare similarly. Note that for the autoencoder, the best test set is attained by ACP, and also in a shorter time than all other methods.

#### 6.5.3.2. *Does the improvement come from the centering trick or from the covariances ?*

In the centering experiment, we clearly see that full ACP outperforms all centered activation methods by a large margin. Centered gradient descent does not even improve on SGD. This is in contradiction with our early experiments on a smaller number of iterations, which showed much better progress for early training. The best performing centered gradient is mean only batch norm. In addition to centering the gradients, it also probably better condition the problem by modifying the backpropagated gradients.

We can thus conclude that the improvement of ACP and other second order methods does not only come from centering the activations, but also from the full covariance.

#### 6.5.3.3. *Stability*

There is a stability issue with ACP visible on the blue curve in figure 6.3. Recall that we divide the Tikhonov regularization factor by a varying $\alpha$, proportional to the gradient incoming in the current layer. This stability issue could probably be eased with regularization by adding a small scalar value before dividing.

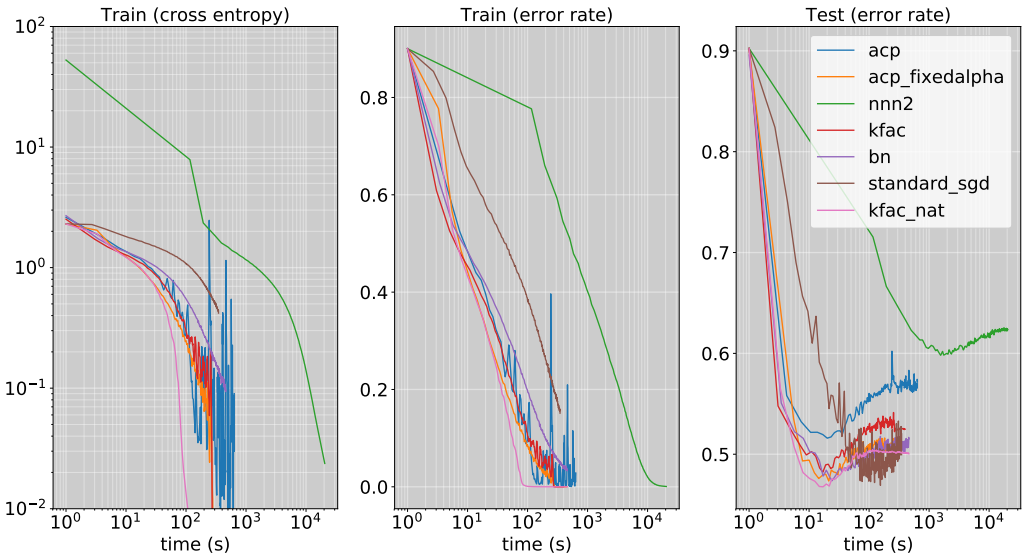


FIGURE 6.3. Comparison of second order methods on CIFAR10 classification: blue: ACP with fixed $\alpha$ as presented in section 6.3.2; orange: ACP with heuristic for $\alpha$ as presented in section 6.3.2; green: Natural Neural Network; red: KFAC for Gauss-Newton; purple: batch norm with standard SGD; brown: standard SGD; pink: KFAC for natural gradient

6.5.3.4. *How does KFAC compare to KFAC for Gauss-Newton ?*

As mentionned, the FIM and the GN are the same for the quadratic error. But this is not the case for the cross entropy. Both methods can be decomposed using the Kronecker product of 2 small matrices, so we want to compare how they perform. As can be seen in figure 6.3 by comparing the red curve for GN and the pink curve for FIM, we observe that the natural gradient version performs much better than the GN version, with respect to both the number of updates and the wallclock time. It is remarkable as the natural gradient version takes much longer to recompute stats, because it needs the whole ~~jacobians~~ jacobian of the output of the network, whereas GN only needs the gradient, so there is no additional cost for the backpropagation as we mentionned in section 3.4.

# CONCLUSIONS

~~During~~ In this work, we derived new expressions for the well known GN matrix and FIM. We showed that there is a common factor in both matrices that is composed of the covariance matrix of the activation at each layer, in the case of a block diagonal approximation. By separating the bias and the weight matrices we introduced a new mathematical explanation for the well-known centering trick. Using this new expressions we derived a new algorithm ACP that loosely resembles ~~2~~ two state of the art methods inspired by natural gradient. We benchmarked our new algorithm against these methods and showed that they all perform similarly.

We also introduced a tentative modification of backpropagation in order to obtain better derivatives. This algorithm showed promising result since it provided better updates than vanilla gradient descent. However there remains some limits to applying this technique in a real setup as it is still too computationnally expensive.

A natural follow-up to this work is to extend it to other architectures such as recurrent neural networks as formally initiated in Ollivier [2015] or convolutional networks such as in Grosse and Martens [2016]. Networks with very small outputs can also be good candidates, as computing the jacobians and thus the FIM is linear in the number of outputs. Amongst them is the very popular family of GAN networks where the output is a single unit, and where the natural gradient could be used as a way to stabilize the training by balancing the rate of change of the output from each part (generator and discriminator).

Another direction of pursuing this research is to look for better approximations of the layer FIM/GN than the one of splitting into 2 expectations. Indeed, by better we do not mean an approximation that is closer in norm to the real FIM, but instead to an approximation that will give updates that are more efficient. Amortization can also be improved, by monitoring how our inverse statistics stay close to the true statistics, and just performing updates of these ~~preconditioner~~ preconditioners when it is necessary, allowing for less updates (and less matrix inversions) for layers where the statistics do not change much.

As a last word, let us just state that second order methods have proven very powerful in a wide variety of optimization problems, but suffer from their ~~computation~~ computational complexity and the difficulty to use them in a setup with a lot of variables to optimize. We

hope that by pursuing this effort of clarifying things and finding approximate methods that are computationnally cheaper, we will carry on contributing to more efficient neural network training.

# Bibliography


Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2): 251–276, 1998.

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

Yoshua Bengio. How auto-encoders could provide credit assignment in deep networks via target propagation. *arXiv preprint arXiv:1407.7906*, 2014.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, et al. Natural neural networks. In *Advances in Neural Information Processing Systems*, pages 2071–2079, 2015.

Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576*, 2015.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Yann LeCun, Patrice Y Simard, and Barak Pearlmutter. Automatic learning rate maximization by on-line estimation of the hessian's eigenvectors. In *Advances in neural information processing systems*, pages 156–163, 1993.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010.

Dong-Hyun Lee, Saizheng Zhang, Asja Fischer, and Yoshua Bengio. Difference target propagation. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 498–515. Springer, 2015.

James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International Conference on Machine Learning*, pages 2408–2417, 2015.

James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040, 2011.

Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

Yann Ollivier. Riemannian metrics for neural networks i: feedforward networks. *arXiv preprint arXiv:1303.0818*, 2013.

Yann Ollivier. Riemannian metrics for neural networks ii: recurrent networks and learning symbolic data sequences. *Information and Inference: A Journal of the IMA*, 4(2):154–193, 2015.

Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. Technical report, DTIC Document, 1961.

Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

Nicol N Schraudolph. Centering neural network gradient factors. In *Neural Networks: Tricks of the Trade*, pages 205–223. Springer, 2012.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958–1970, 2008.

Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor S Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. In *ICML*, pages 1349–1357, 2016.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

# Appendix A

## DERIVATIONS OF THE SECOND DERIVATIVES OF COMMON LOSS FUNCTIONS

### A.1. Quadratic error

$$\begin{aligned} l(f,y) &= \|f-y\|_2^2 \\ \frac{\partial l}{\partial f} &= 2(f-y)^T \\ \frac{\partial^2 l}{\partial f^2} &= \mathbf{I} \end{aligned}$$

### A.2. Binary cross entropy

$$\begin{aligned} l(f,y) &= -(y\log(f)+(1-y)\log(1-f)) \\ \frac{\partial l}{\partial f} &= -\left(\frac{y}{f}-\frac{1-y}{1-f}\right) \\ \frac{\partial^2 l}{\partial f^2} &= \frac{y}{f^2}+\frac{1-y}{(1-f)^2} \end{aligned}$$

### A.3. Multiclass cross entropy

$f$ and $y$ are the vector (true, estimated) of probabilities of being a member of each class.

$$\begin{aligned} l(f,y) &= -y^T\log(f) \\ \frac{\partial l}{\partial f} &= -\left(\frac{y}{f}\right)^T \\ \frac{\partial^2 l}{\partial f^2} &= \operatorname{diag}\left(\frac{y}{f^2}\right) \end{aligned}$$

Here all operations (division, logarithm) are elementwise.

# Appendix B

---

# DERIVATIONS OF THE EXPRESSION FOR THE FIM

$$\mathbf{F} = \mathbb{E}_{x\sim q(x)}\left[\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(\frac{\partial \log p_\theta(y|x)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta(y|x)}{\partial\theta}\right)\right]\right]$$

## B.1. Quadratic error

In the case of the quadratic error we suppose that the samples are drawn from a gaussian with diagonal covariance matrix $\sigma^2\mathbf{I}$ centered in $f_\theta(x)$ (the output of the network). We denote by $n$ the dimension of the output.

$$p(y|x) = \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}}\exp\left(-\frac{1}{2\sigma^2}(y-f_\theta(x))^T(y-f_\theta(x))\right)$$

$$\log p_\theta(y|x) = -\frac{1}{2\sigma^2}(y-f_\theta(x))^T(y-f_\theta(x)) - \frac{n}{2}\log\left(2\pi\sigma^2\right)$$

$$\frac{\partial \log p_\theta(y|x)}{\partial\theta} = -\frac{1}{\sigma^2}(y-f_\theta(x))^T\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)$$

$$\left(\frac{\partial \log p_\theta(y|x)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta(y|x)}{\partial\theta}\right) = \frac{1}{(\sigma^2)^2}\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)^T(y-f_\theta(x))(y-f_\theta(x))^T\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)$$

$$\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(\frac{\partial \log p_\theta(y|x)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta(y|x)}{\partial\theta}\right)\right]$$

$$= \frac{1}{(\sigma^2)^2}\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)^T\mathbb{E}_{y\sim p_\theta(y|x)}\left[(y-f_\theta(x))(y-f_\theta(x))^T\right]\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)$$

$$= \frac{1}{(\sigma^2)^2}\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)^T\sigma^2\mathbf{I}\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)$$

$$= \frac{1}{\sigma^2}\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)^T\left(\frac{\partial f_\theta(x)}{\partial\theta}\right)$$

## B.2. Binary cross entropy

$$
\begin{aligned}
\log p_\theta\left(y|x\right) &= \left(y\log\left(f_\theta\left(x\right)\right)+\left(1-y\right)\log\left(1-f_\theta\left(x\right)\right)\right)\\
\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta} &= \left(\frac{y}{f_\theta\left(x\right)}-\frac{1-y}{1-f_\theta\left(x\right)}\right)\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right) &= \left(\frac{y}{f_\theta\left(x\right)}-\frac{1-y}{1-f_\theta\left(x\right)}\right)^2\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
&= \frac{\left(y-f_\theta\left(x\right)\right)^2}{f_\theta\left(x\right)^2\left(1-f_\theta\left(x\right)\right)^2}\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right)\right] &= \frac{\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(y-f_\theta\left(x\right)\right)^2\right]}{f_\theta\left(x\right)^2\left(1-f_\theta\left(x\right)\right)^2}\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
&= \frac{1}{f_\theta\left(x\right)\left(1-f_\theta\left(x\right)\right)}\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)
\end{aligned}
$$

## B.3. Multiclass cross entropy

$f$ and $y$ are the vector (true, estimated) of probabilities of being a member of each class.

$$
\begin{aligned}
\log p_\theta\left(y|x\right) &= y^T\log\left(f_\theta\left(x\right)\right)\\
\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta} &= \left(\frac{y}{f_\theta\left(x\right)}\right)\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right) &= \left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\operatorname{diag}\left(\frac{y^2}{f_\theta\left(x\right)^2}\right)\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
\mathbb{E}_{y\sim p_\theta(y|x)}\left[\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right)^T\left(\frac{\partial \log p_\theta\left(y|x\right)}{\partial\theta}\right)\right] &= \left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\operatorname{diag}\left(\frac{\mathbb{E}_{y\sim p_\theta(y|x)}\left[y^2\right]}{f_\theta\left(x\right)^2}\right)\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)\\
&= \left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)^T\operatorname{diag}\left(\frac{1}{f_\theta\left(x\right)}\right)\left(\frac{\partial f_\theta\left(x\right)}{\partial\theta}\right)
\end{aligned}
$$

Here all operations (division, logarithm) are elementwise.