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 21, 2017

SOMMAIRE

Sommaire et mots-clés français...

SUMMARY

English summary and keywords...

CONTENTS

LIST OF TABLES

LIST OF FIGURES

DÉDICACES

Vos dédicaces.

REMERCIEMENTS

Merci !

INTRODUCTION

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

This work is mostly focused on optimization in artificial neural networks. My main contributions are a deeper understanding of the very particular structure of the Fisher Information matrix and the Hessian matrix when used in the setting of neural networks. It gives new insights on some methods that have proven to be very good empirically such as batch normalization or RMSProp, but without a formal understanding of why they perform so well.

The first part sets up the basic framework of machine learning and introduces neural networks.

In the second part, we introduce the usual methods of optimization that have enabled the recent successes in machine learning and in particular vision from 2012.

The third part showcases the results of my research. It is mostly a dive in into the mathematical expressions behind second order methods, the definition of algorithms that make use of some approximations of these expressions, and the implementation and benchmarking of these algorithms.

CONTENTS

Chapter 1

NEURAL NETWORKS

In this chapter, we will progressively 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, which will

be close to 0 is 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 32×32 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 family of functions, which consists in parametrized functions $\mathcal{F}_\theta$. We will denote by f_θ such a function, parametrized by a vector of parameters θ. Adapting the value of the parameters will change the output of the function f_θ. 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 a loss function l, adapted to this task. The better our function, the lower the value of l. 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_θ for the given task is given by the risk:

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

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

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

Finding this value θ^* 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 target y.

In **regression**, the input vector x is mapped to a numerical value y. To assess the performance of f_θ, we use the loss function $l(f_\theta(x), y) = \|f_\theta(x) - y\|_2^2$, 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_θ 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* $l(f_\theta(x), y) = -\log\Big((f_\theta(x))_y\Big)$. This will push the probability of the correct category toward 1. An example classification task is proposed by the ImageNet project [6] 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:

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

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

For random values of the parameters θ, 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 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 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 [17]. 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 [25]. 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 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 biological neuron. 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.

This single perceptron is extended in a more complex model called the multilayer perceptron. 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 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.

1.4.2. Convolutional networks

1.4.3. Autoencoders

Autoencoders [13, 31] are neural networks that are composed of an encoder part a decoder. 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.

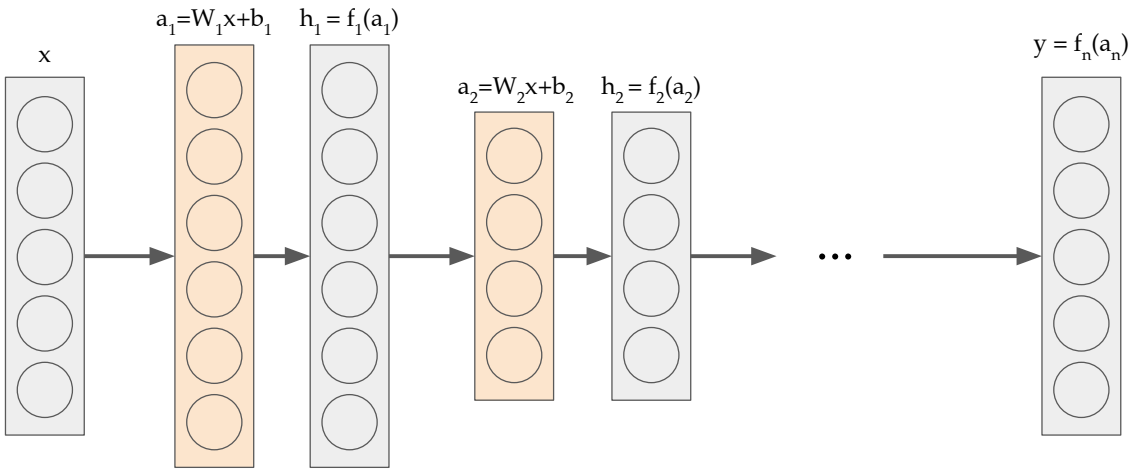

FIGURE 1.1. A multilayer perceptron consists in alternatively stacking layers of a linear transformation and a nonlinearity

1.4.4. Residual networks

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 [8] and its feed-forward extension [30] 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. α and β are scalar values that control the influence of each part of the loss. In the original paper [8] we start from a randomly initialized x and we perform gradient descent on each pixel of x. In [30] 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. We will explain how to create $\mathcal{L}_{\text{style}}$ and $\mathcal{L}_{\text{content}}$ in details in **??**.

Another family of cost functions that becomes more and more popular is that of the discriminators in **Generative Adversarial Networks** [11], 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 isalso 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 capable of solving a given 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 θ. 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 $\nabla_\theta R$:

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

Once we have a direction, we must choose how far to move in this direction. Several line search algorithms have been developed, but they require evaluating our objective several times, which can be costly for deep networks or big datasets. We will stick to a simple fixed learning rate, so that each iteration becomes:

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

Of course the scalar parameters λ plays a very important role. If we choose a value that is too small then it will take several steps to reach the same point, so it will take longer. If the value is too big 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.

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 $h_{l+1} = f_l(W_l h_l + b_l)$ using the expression:

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

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):

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

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):

$$\begin{aligned} \mathbf{J}_{a_l}^{W_l} &= \nabla_{a_l} l \left(h_l\right)^T \\ \mathbf{J}_{a_l}^{b_l} &= \nabla_{a_l} l \end{aligned}$$

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 [3]. Using

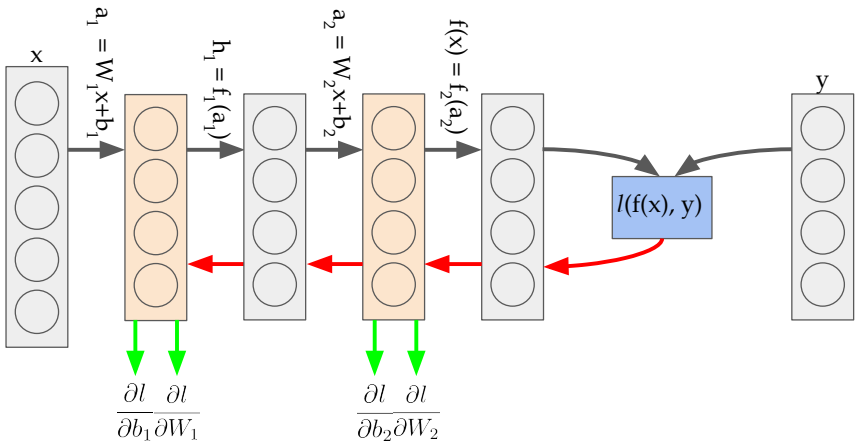

FIGURE 2.1. Forward (in black) and backward (in red) propagation of the intermediate results of the process of computing the output of the network and the gradient corresponding to this output and the desired "true" output. The green arrows represent the computation of the gradients with respect to the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

such a tool we can define a model and a scalar cost function, then call a method `grad` that takes care of the computation 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 risk.

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 **stochastic gradient descent** (SGD) [5]. The main benefit of using SGD instead of full gradient descent is that we can reduce the memory required to compute the gradient. The memory required to backpropagate the intermediate gradients (red arrows in figure 2.1) is proportional to the size of the minibatch. As datasets become bigger and bigger, while GPUs used for accelerating the computations have limited memory, adjusting the batch size is a good way of making an experiment fit on a selected computer.

2.3. Hyperparameters

In the preceding sections we have introduced the learning rate 2.1 and the minibatch size 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. We do not find the optimal value for the hyperparameters using gradient descent, but instead we tune it by running several time the same experiment with different hyperparameter values, and control

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 3 methods and motivate our use of a new technique that we call biased random search.

The most simple hyperparamater 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 value 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 has no important effect on the learning algorithm [4]. It will explore more different values for the other hyperparameters.

In the rest of this work, we will use an extension of random search that we call **biased random search**. 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 [1] 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 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). 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**
  $a \sim U(\mathcal{D})$ ▷ Sample values for HP
  rejected $\leftarrow$ true
  while rejected **do**
    $c_{estimate} \leftarrow \mathcal{M}(\mathcal{H}, a)$ ▷ 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
  $result \leftarrow run(a)$
  $\mathcal{H} \leftarrow \mathcal{H} + (a, result)$
end while

As an illustration, we ran all 3 methods on the task described in 2.5, using standard stochastic gradient descent with fixed minibatch size, for a fixed number of parameter updates. We tune 2 hyperparameters: the learning rate and the variance of the initial random weights, and plot the result in 2.2. These plots show the interaction between 2 hyperparameters.

These plots and this random search technique are a key component for assessing the true performance of optimization techniques that we present in section 3. Indeed it is easy to experimentally find that a new optimization technique gives better performance than a baseline if we spend too much time tuning hyperparameters for our new technique.

To assess the performance of biased random search we ran 100 searches of 100 experiments on a simple task where we tuned 2 hyperparameters. We observe that it consistently find comparable or better results than standard random search (lower is better):

HP tuning procedure	Average	Standard deviation
Grid search		0
Random search	27.02	0.28
Biased random search	26.61	0.13

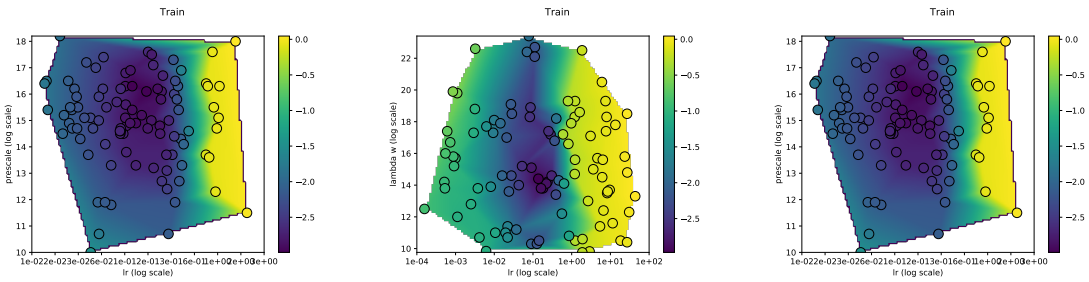

FIGURE 2.2. Comparison of hyperparameter tuning methods. On the left a grid search, in the middle a random search and on the right a biased random search. Each experiment consisted in 100 iterations of SGD from a randomly initialized network. We tune 2 hyperparameters on the x and y axis (what they represent 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.

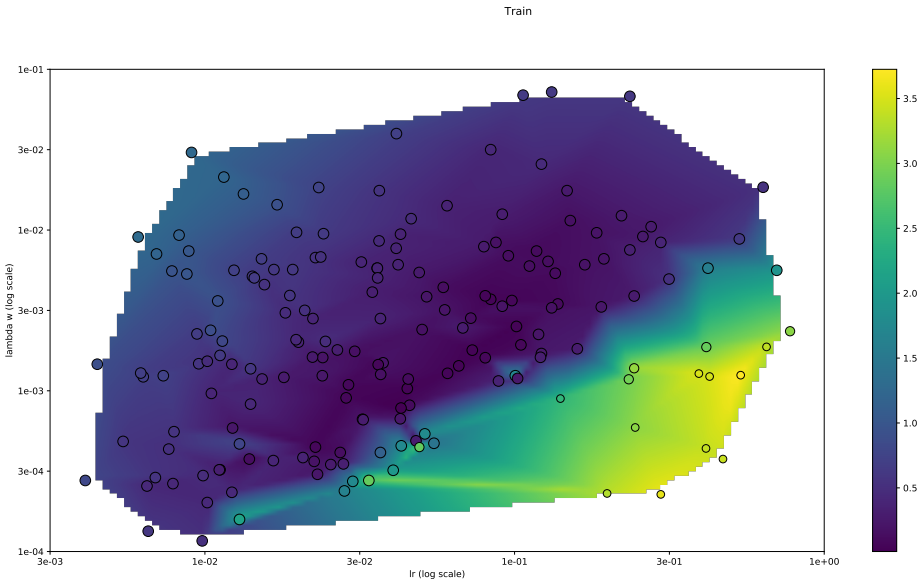

FIGURE 2.3. Relation between 2 hyperparameters: for this experiment we can clearly see that the plotted hyperparameters are not independant one from each other.

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. Every gradient update has approximately the same magnitude, controlled by the learning rate. 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 $\left(\mathbf{J}_{a_l}^{h_l}\right)^T \left(\mathbf{J}_{h_{l+1}}^{a_l}\right)^T$ have a spectral norm that is too big, then the gradient will become bigger and bigger for lower layers. This can happen if the weight matrices have singular values that are too big compared to 1, or if the derivatives of the activation functions take big 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 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 [24]. Perhaps less known is the fact that we can also use second order methods to compensate for gradient vanishing/exploding [22], that we will present in chapter 3.

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, 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 [9] and He initialization [12]. 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, α depends on the activation function, and the papers propose default values for sigmoid and ReLU [10]. In our experiments, we treated α as a hyperparameter, and tuned it using a biased random search 2.3.

2.4.3. Gradient smoothing methods

A family of optimization tricks make use of the representation of the empiriral risk as a manifold in the space of the parameters. 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 lower derivative for longer in order to achieve the same improvement as for a direction that has a greater derivative. Many popular techniques use this principle, the most successful ones at the time of writing being Adam [16], RMSProp [29], Nesterov momentum and so on.

2.5. 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 [18] is composed of 60.000 28×28 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 [13, 20, 21, 7]. 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 1000×784).

Chapter 3

ADVANCED OPTIMIZATION OF NEURAL NETWORKS

Gradient descent is a black box technique that can be applied to any kind of optimization problem. But we are interested in optimizing a very specific function, which is a sum of stacked transformations and a loss function. In this section, we introduce the well known second order methods known as Newton's method and Natural gradient descent. We then derive the updates of this methods adapted to neural networks, and show how second order methods can be approximated in order to be applied to real scale tasks.

3.1. Second order methods

3.1.1. Newton steps

Second order methods refers 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 to be minimized:

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

$\left(\nabla^2 f\right)_x$ is the Hessian matrix of f, expressed in x. We use the little-o notation o that represents an unknown function with the only property that $\lim_{x\to 0} \frac{o(x)}{x} = 0$. By ignoring higher order terms ($o\left(\|\Delta x\|_2^2\right) = 0$) we have a quadratic approximation for f. Using this approximation in a minimization problem, we get 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 Δx, and setting it to zero in order to obtain:

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

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:

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

This update 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, the updates will converge to a 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. 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 more nodes. We will explore this question in more details in 3.1.4.
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 $O\left(n^2\right)$. We will present an approximation of the Hessian that saves memory in 3.1.6.
3. *Inverting the Hessian matrix*: Inverting the Hessian matrix is also costly as it grows in $O\left(n^3\right)$ with the size of the matrix. Some techniques use 2nd order information without inverting the Hessian such as *Hessian Free* [20]. We propose to factorize the Hessian so as to require inverting a smaller matrix while benefiting from some 2nd order information in 3.3.3.

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:

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

If we ignore the second derivative and higher order terms we can simplify this expression to the following one that is often used for deriving the first order gradient descent update:

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

By solving this minimization problem we recover the update $\Delta x^* = -\frac{1}{\lambda} (\nabla f)_x$. But of course this λ "hides" second order information. In fact, [19] proposes to automatically adapt the value of the learning rate by using the biggest eigenvalue of the hessian as λ. In this case we are guaranteed to go as far as possible in the direction of greatest curvature (the corresponding eigenvector).

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 Δx.

To counter this undesirable effect, we simply add a regularization term that penalizes bigger values of Δx:

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

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 ϵ 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 $\left(\nabla^2 f\right)_x$ is too big, and that it can account for the estimation error when we estimate $\left(\nabla^2 f\right)_x$ 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:

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

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:

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

$\mathbf{J}$ is the jacobian of the output of the network f with respect to the parameters θ. In this notation we made the dependance in θ 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:

$$\begin{aligned}\frac{\partial^2}{\partial \theta^2}\left\{l\left(f\left(x_i,\theta\right),y_i\right)\right\} &= \underbrace{\mathbf{J}_\theta\left(x_i,\theta\right)^T\frac{\partial^2}{\partial f^2}\left\{l\left(f\left(x_i,\theta\right),y_i\right)\right\}\mathbf{J}_\theta\left(x_i,\theta\right)}_{G_f(x_i,\theta)} \\ &\quad + \sum_j\left(\nabla^2 f_j\left(x_i,\theta\right)\right)\left(\frac{\partial}{\partial f_j}\left\{l\left(f\left(x_i,\theta\right),y_i\right)\right\}\right)^T\end{aligned}$$

$G_f\left(x_i,\theta\right)$ is called the Gauss-Newton approximation of the Hessian [27]. The remainder is proportional to $\frac{\partial}{\partial f_j}\left\{l\left(f\left(x_i,\theta\right),y_i\right)\right\}$. 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 $l\left(f\left(x_i,\theta\right),y_i\right)$, we will have $\frac{\partial^2}{\partial \theta^2}\left\{l\left(f\left(x_i,\theta\right),y_i\right)\right\} = G_f\left(x_i,\theta\right)$ so it is a reasonable approximation to use in practice. Note that a minimum for the empirical risk $R\left(\theta\right)$ will not necessarily be a minimum for each example $l\left(f\left(x_i,\theta\right),y_i\right)$, 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

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

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 **??**

complicated because it involves a second derivative of a composed function. To illustrate this point we write the first and second derivatives of a scalar function $f\circ g\circ h:\mathbb{R}\to\mathbb{R}$:

$$\begin{aligned}(f\circ g\circ h)' &= \left(f'\circ g\circ h\right)\left(g'\circ h\right)h'\\ (f\circ g\circ h)^{(2)} &= \left(f^{(2)}\circ g\circ h\right)\left(\left(g'\circ h\right)h'\right)^2+\left(f'\circ g\circ h\right)\left(g^{(2)}\circ h\right)\left(h'\right)^2+\left(f'\circ g\circ h\right)\left(g'\circ h\right)h^{(2)}\end{aligned}$$

The second derivative requires derivating once more each term in the chain rule. For this simple example of 3 composed functions, this translates into 3 backpropagations. For bigger networks it quickly becomes very slow.

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 (3. I).

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

$$G_f\left(\theta\right) = \frac{1}{n}\sum_i \mathbf{J}_\theta\left(x_i,\theta\right)^T D\left(f\left(x_i,\theta\right),y_i\right)\mathbf{J}_\theta\left(x_i,\theta\right)$$

We will show in 3.3 that this matrix can be factorized to design optimization algorithms adapted to the particular structure of neural networks.

3.1.5. Interpretation from the output of the network

We can rewrite the GN matrix applied to an update as a norm in the space of the output of the network:

$$\begin{aligned}\Delta\theta^* &= \operatorname{argmin}_{\Delta\theta}\left(\nabla R\right)_\theta^T\Delta\theta+\frac{1}{2}\Delta\theta^T\frac{1}{n}\sum_i\mathbf{J}\left(x_i,\theta\right)^T D\left(f\left(x_i,\theta\right),y_i\right)\mathbf{J}\left(x_i,\theta\right)\Delta\theta+\frac{\epsilon}{2}\left\|\Delta\theta\right\|_2^2\\ &= \operatorname{argmin}_{\Delta\theta}\left(\nabla R\right)_\theta^T\Delta\theta+\frac{1}{2}\frac{1}{n}\sum_i\left\|\sqrt{D\left(f\left(x_i,\theta\right),y_i\right)}\mathbf{J}\left(x_i,\theta\right)\Delta\theta\right\|_2^2+\frac{\epsilon}{2}\left\|\Delta\theta\right\|_2^2\end{aligned}$$

By an abuse of notation we denoted by $\sqrt{}$ an elementwise square root. Written like this GN can be considered a first order measure of what will be the impact of a $\Delta\theta$ in terms of change of change of the output of the network as measured by some metrics $D\left(f\left(x_i,\theta\right),y_i\right)$ dependent on the example considered. It constrains all changes to induce a comparable change on the output. Interestingly, for cross entropies (binary and multiclass) we have the unexpected equivalence $\sqrt{D\left(f\left(x_i,\theta\right),y_i\right)}=-\left(\frac{\partial l}{\partial f}\right)^T$ which can be combined with the jacobian to give the simplified formulation:

$$\begin{aligned}\Delta\theta^* &= \text{argmin}_{\Delta\theta}\left(\nabla R\right)_\theta^T\Delta\theta+\frac{1}{2}\frac{1}{n}\sum_i\left\|-\left(\frac{\partial l}{\partial f}\right)^T\mathbf{J}\left(x_i,\theta\right)\Delta\theta\right\|_2^2+\frac{\epsilon}{2}\left\|\Delta\theta\right\|_2^2\\ &= \text{argmin}_{\Delta\theta}\left(\nabla R\right)_\theta^T\Delta\theta+\frac{1}{2}\frac{1}{n}\sum_i\left\|\left(\nabla l\left(f\left(x_i,\theta\right),y_i\right)\right)_\theta^T\Delta\theta\right\|_2^2+\frac{\epsilon}{2}\left\|\Delta\theta\right\|_2^2\end{aligned}$$

We will draw a link between this formulation and the well-used RMSProp [29] technique in 3.6.1.

3.1.6. 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 $O\left(n^3\right)$ 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

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 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:

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

It is a non-negative quantity that resembles a measure of how much q differs from p. In particular, $\mathrm{KL}\left(p \parallel q\right) = 0$ when $p = q$. Note that it is not symmetric, so it can not directly be used as a metric.

The idea of the natural gradient is to use the KL divergence as a regularizer when doing gradient descent. We will denote by p_θ a parametric model and $\Delta\theta$ a change in its parameter values. $\mathrm{KL}\left(p_\theta \parallel p_{\theta+\Delta\theta}\right)$ is used as our regularizer, so that each change $\Delta\theta$ gives the same change in the distribution space. Instead of using the full expression for $\mathrm{KL}\left(p_\theta \parallel p_{\theta+\Delta\theta}\right)$ we will use its second order Taylor series around θ (for full derivation see for instance [23]):

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

$\mathbf{F} = \mathbb{E}_x\left[\left(\frac{\partial \log p_\theta(x)}{\partial\theta}\right)^T \left(\frac{\partial \log p_\theta(x)}{\partial\theta}\right)\right]$ is the Fisher information matrix, which can now be used directly as a regularizer. 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. An expression for the FIM using jacobians

We use the probabilistic interpretation of neural networks. In this case the output is seen as a conditional probability $p\left(y|x\right) = f_\theta\left(x\right)$ of observing the value y given an input x. The FIM can be expressed $\mathbf{F} = \mathbb{E}_{x,y\sim p(x,y)}\left[\left(\frac{\partial \log p_\theta(x,y)}{\partial\theta}\right)^T \left(\frac{\partial \log p_\theta(x,y)}{\partial\theta}\right)\right]$ which simplifies in:

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

Loss function	$\mathbf{F}(x_i, \theta)$
quadratic error	TODO
cross entropy for binary decision	TODO
cross entropy for multiclass classification	TODO

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.

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

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

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 [23]:

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

The values for x are drawn from the data generating distribution q. The notation $\boldsymbol{J}_{\boldsymbol{y}(x)}$ is used for the jacobian of the output of the network (i.e. the probability expressed at a given $x : p(y \mid x)$), with respect to the parameters. In other words, it measures how much the output of the network will change for a given x if we change the parameters. 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.

3.2.3. Approximating the FIM

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: $\mathbf{F} = \frac{1}{n}\sum_i \boldsymbol{J}_{\boldsymbol{y}(x_i)}^T D(\boldsymbol{y}(x_i))\,\boldsymbol{J}_{\boldsymbol{y}(x_i)}$.

3.2.4. (Natural) gradient descent

The usual gradient descent algorithm can be formulated as the minimization of the following expression:

$$\Delta\theta \;=\; \text{argmin}_{\Delta\theta}\left\{\Delta\theta^T\nabla_\theta R + \frac{\epsilon}{2}\left\|\Delta\theta\right\|^2\right\}$$

This expression can be easily solved giving the usual gradient descent update $\Delta\theta = -\frac{1}{\epsilon}\nabla_\theta R$. The parameter ϵ is the inverse of the 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:

$$\Delta\theta \;=\; \text{argmin}_{\Delta\theta}\left\{\Delta\theta^T\nabla_\theta R + \frac{\epsilon}{2}\left\|\Delta\theta\right\|^2 + \frac{\lambda}{2}\Delta\theta^T\mathbf{F}\Delta\theta\right\}$$

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 $\Delta\theta = \frac{1}{\lambda}\left(\mathbf{F} + \frac{\epsilon}{\lambda}\mathbf{I}\right)^{-1}\nabla_\theta R$. This expression also gives an insight for the role of λ and ϵ, which control 2 different but related quantities expressed by our constraints. This new update is called the natural gradient [2].

3.2.5. Relation with 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 diagonal matrix. The main difference is in this diagonal matrix. 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. In the case of the GN matrix it depends on the true target except for the quadratic error.

Note that this share of structure was acknowledged in [23] but for unsupervised learning, in which case the FIM and the GN are strictly equal.

3.3. Second order: a new perspective

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.

3.3.1. 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 [21]. 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:

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

Getting back to the expression for the Gauss-Newton matrix derived in 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 $h' = f\,(a)$. 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 θ. In order to get an expression for this jacobian, we now make use of the *vec* operator to transform W into a vector:

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

$\mathbf{I}$ is the identity, of the same size than 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 $vec(W)$ and b in a vector θ:

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

And finally by the chain rule:

$$\mathbf{J}_\theta = \begin{pmatrix} h^T & 1 \end{pmatrix} \otimes \mathbf{J}_a \tag{3.3.1}$$

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 $G_f = \frac{1}{n}\sum_i \mathbf{J}_\theta(x_i)^T D(x_i) \mathbf{J}_\theta(x_i)$ from 3.1.4. We can rewrite this expression using the factorization 3.3.1:

$$\begin{aligned} G_f &= \frac{1}{n}\sum_i \left[\begin{pmatrix} h_i^T & 1 \end{pmatrix} \otimes \mathbf{J}_{a_i}\right]^T D(x_i) \left[\begin{pmatrix} h_i^T & 1 \end{pmatrix} \otimes \mathbf{J}_{a_i}\right] \\ &= \frac{1}{n}\sum_i \begin{pmatrix} h_i^T & 1 \end{pmatrix}^T \begin{pmatrix} h_i^T & 1 \end{pmatrix} \otimes \left(\mathbf{J}_{a_i}^T D(x_i) \mathbf{J}_{a_i}\right) \\ &= \frac{1}{n}\sum_i \begin{pmatrix} h_i h_i^T & h_i \\ h_i^T & 1 \end{pmatrix} \otimes \left(\mathbf{J}_{a_i}^T D(x_i) \mathbf{J}_{a_i}\right) \end{aligned} \tag{3.3.2}$$

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), and a part that only uses the forward statistics and that is local to a layer. While these 2 contributions are clearly separated for a single example, it becomes less clear as we sum up for several examples. As we will show in the next section, similar approximations were exploited in KFAC [21] and Natural Neural Networks [7] to build efficient optimization algorithms.

3.3.2. 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 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.

3.3.3. Focus on the covariance part of the decomposition

We now suppose that we can use the following approximation:

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

Here α does not depend on i. This approximation means that we ignore the interactions between the output neurons of a layer. Instead we just focus on some statistics of the activations of the current layer. We can interpret this preconditioner as penalizing an update if the corresponding activation has a high variance. 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 $\mathbf{F}_{in}$ 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 $\theta = \begin{pmatrix} \mathrm{vec}\left(W\right)^T & b^T \end{pmatrix}^T$:

$$\nabla_\theta R \;=\; \frac{1}{n}\sum_j \begin{pmatrix} h_j \\ 1 \end{pmatrix} \otimes \nabla_{a_j} l$$

The corresponding update is:

$$\mathbf{F}_{in}^{-1}\nabla_\theta R \;=\; \frac{1}{n}\sum_j \begin{pmatrix} C^{-1} & -C^{-1}\frac{1}{n}\sum_i h_i \\ -\frac{1}{n}\sum_i h_i^T C^{-1} & 1+\frac{1}{n}\sum_i h_i^T C^{-1}\frac{1}{n}\sum_i h_i \end{pmatrix} \begin{pmatrix} h_j \\ 1 \end{pmatrix} \otimes \frac{1}{\alpha}\nabla_{a_j} l$$

From this expression we can write the update for W:

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

$$\boxed{\Delta W = \frac{1}{\alpha}\frac{1}{n}\sum_j \nabla_{a_j} l \left(h_j - \frac{1}{n}\sum_i h_i\right)^T C^{-1}} \tag{3.3.3}$$

And the update for b:

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

$$\boxed{\Delta b = \frac{1}{\alpha}\frac{1}{n}\sum_j \nabla_{a_j} l - \Delta W \frac{1}{n}\sum_i h_i} \tag{3.3.4}$$

These expressions are an original contribution.

3.3.4. Comparison with standard SGD

The updates for standard SGD are $\Delta_{SGD} b = \frac{1}{n}\sum_j \nabla_{a_j} l$ and $\Delta_{SGD} W = \frac{1}{n}\sum_j \nabla_{a_j} l \left(h_j\right)^T$.

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

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 gives a new theoretical understanding of an old well used trick [17, 28].

3.4. Algorithms and experiments

3.4.1. Centered gradient descent

Following the update for W derived in 3.3.3, we simply replace the usual update for the weight matrices by a centered version $\Delta W = \frac{1}{n}\sum_j \nabla_{a_j} l \left(h_j - \frac{1}{n}\sum_i h_i\right)^T$. The gradient, as well as the inner expectation, are computed using a minibatch. Some author refer to a very similar idea as *mean-only batch normalization* [26] but 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.

Algorithm 2 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

3.4.2. Amortized covariance preconditioner

In the updates derived from the covariance 3.3.4 3.3.3, 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 α. We adopt two approaches. The first one consists in treating it as a hyperparameter and tune it using our biased random search 2.3. The second one is a very experimental heuristics, which consists in taking the maximum value of the squarred gradient $\alpha = \max_{i \in \text{minibatch}, j \leq n_{out}} \left(\nabla_{a_i} l\right)_j^2$. 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 measure of the curvature of the 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. In this case it is scaled by $\frac{1}{\alpha}$:

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

3.4.3. Experiments

3.5. Other related approximate second order algorithms

The 2 following techniques have been proposed using the same factorization of the FIM that we wrote in 3.3.2. In addition to their factorization we introduced the explicit separation of weight matrix and bias.

Algorithm 3 Amortized covariance preconditioner

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

$n_{updates} \leftarrow 0$
while not converged **do**
  if $n_{updates} \mod \mathbf{N} = 0$ **then**
    Sample a minibatch $\mathcal{D}$
    for each layer j **do**
      $C^{(j)} \leftarrow \text{cov}\left(h^{(j)}, h^{(j)}\right)$
      $inv\_C^{(j)} \leftarrow \text{inverse}\left(C^{(j)}\right)$
    end for
  end if
  Sample a minibatch $\mathcal{D}$
  for each layer j **do**
    $\Delta_{a_i^{(j)}} \leftarrow \nabla_{a_i^{(j)}} l\left(f\left(x_i\right), y_i\right) \forall i \in \mathcal{D}$
    $b \leftarrow b + \lambda \frac{1}{n} \sum_i \Delta_{a_i^{(j)}}$
    $W \leftarrow W + \lambda \frac{1}{n} \sum_i \Delta_{a_i^{(j)}} \left(h_i^{(j)} - \frac{1}{n} \sum_k h_k^{(j)}\right)^T inv\_C^{(j)}$
  end for
  $n_{updates} \leftarrow n_{updates} + 1$
end while

3.5.1. KFAC

KFAC [21] proposes to split the sum of Kronecker products into 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 section 3.3.3, we lose the coupling between the forward and backward part of the FIM. In our experiments we found a comparable performance between KFAC and ACP.

3.5.2. Natural Neural Networks

Natural neural networks [7] 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, and also they use the centering trick. To this view, they change the original linear transformation $a = Wh + b$ to become:

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

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 μ 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, which will now have the desired property. 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 FIM for the reparametrization thus has a better form.

3.6. Links with well-used methods

3.6.1. RMSProp

WIP: RMSProp is very close to diagonal GN for cross entropies.

3.6.2. Batch normalization

Batch normalization [15, 14] is a very popular way to make the learning procedure more stable in neural networks. It consists in a simple reparametrization that effectively normalize the activations using statistics (mean, variance) computed from a batch of examples, defined as follows:

$$\mathrm{BN}(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}$$

with $\mu = \mathbb{E}[x]$ and $\sigma^2 = \mathbb{E}\left[(x-\mu)^2\right]$ the empirical mean and variance computed over a minibatch in [15] or using a running average in [14]. A "batch-normalized" layer is defined by:

$$h_n(x) = f_n\left(\gamma_n \odot \mathrm{BN}\left(W_n h_{n-1}\right) + \beta_n\right)$$

After reparametrization, we train the parameters W, γ and β using a gradient based method such as stochastic gradient descent. The new gradient "goes through" the BN

operation (mean and square root of the variance) so both the forward pass and the backward pass are modified. We usually initialize the values of γ to be a vector of 1 and for β a vector of 0.

A first effect of this reparametrization, as pointed by the paper, is that it reduces the *covariate shift* of each unit. If we consider a single unit, we can observe an empirical distribution of its activation over a dataset. We have seen in 2.4.2 that carefully tuning the initial weights and biases so that each unit has a desired distribution can improve drastically the efficiency or the learning procedure. This initial distribution changes throughout the learning, and this is what is called the *covariate shift*. Batch normalization forces the distribution of activations to have zero mean and unit variance.

For some experiments, *mean-only* batch normalization can also be used to improve the learning procedure. We will cover this question in more details in **??**.

WIP

CONCLUSIONS

TODO

Bibliography

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

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

[3] 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.

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

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. *arXiv preprint arXiv:1702.03275*, 2017.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

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

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

Bibliography

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

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

[3] 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.

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

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. *arXiv preprint arXiv:1702.03275*, 2017.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

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

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.