Training of a feed-forward, fully connected Neural Network for the recognition of hand-written digits

David Idzard Svejda, Gianmarco Puleo, Henrik Modahl Breitenstein
University of Oslo
(Dated: November 18, 2022)

The MNIST hand-written digit data-set is well-known to be one of the most studied in the literature, and it is often shown as an example in many introductory machine learning courses. We train a Neural Network which can recognize digits from this data-set. We both analyze models with zero and one hidden layer, the first one being an implementation of multiclass logistic regression. Optimization of the parameters is achieved by minimizing the cross entropy with the aid of various gradient descent methods, whose performance is compared. Hyperparameters are adjusted using a test set, and two activation functions are tested for the hidden layer. The quality of our models is then assessed using a validation set. The highest accuracies are found in both cases when using RMSProp gradient descent. In particular we find 90% accuracy for the NN, whereas logistic regression only reaches 89%.

I. INTRODUCTION

In the past few decades, the increase in the performance of classical computers allowed for machine learning to become more and more spread world-wide. Nowadays, anybody can train a Neural Network on their own personal computer, which would have been inconceivable in the 1950s, when machines as big as entire rooms implemented the first examples of perceptron (as reference, see figure 4.8 in [1]). Deep learning (both supervised and unsupervised) is now more than ever employed on a large-scale by many software corporations, in order to analyze and categorize the insanely huge amount of data that they collect [2]. NNs are also widely utilized in the analysis of the signals measured by modern experiments such as the LIGO detector for gravitational waves[3], and LHC at CERN for particle physics[4]. Furthermore, deep learning finds applications in many-body physics, to find the ground state of quantum mechanical systems[5]. All these things considered, the impact and the importance of NNs in our world is crystal-clear. There are many kinds of Neural Networks, the most commonly used being (fully connected) Feed Forward Neural Networks (FFNNs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). In this project, we only focus on the first kind, which we will be simply referred to as "Neural Network" henceforth. In particular, we focus on what can be thought of as the "Hello World" of Deep Learning[6]: training a Neural Network which can recognize the hand-written digits of the MNIST dataset. We also use our model to perform a plain logistic regression.

This paper is structured as follows: in section II we describe the theoretical details of the models and the optimization techniques we use. Section III explains our optimization strategy for the many hyperparameters that can be tweaked, and summarizes the accuracy of our best models. Incidentally, we recall that in ML it is really important to try and understand the meaning of the trained model: what our model does with the data and why it gives a certain output. This is a hot-topic in today's research. In section III we will also try to deal with this issue. Our results are reproducible in the Notebook in our GitHub repository at https://github.com/DavidSvejda2507/FYS-STK4155-Project-2.

In section IV we discuss possible further steps that could be done to improve our work, and we draw the conclusion. Finally, the appendix A contains a description and a critical analysis of how we tested the proper functioning of our code.

II. METHODS

II.1. Dataset

In this report, we are studying a subset of the MNIST hand-written digits data set [7]: we have available a set of 1796 square images, each consisting in 64 pixels. Each pixel is a number between 0 and 1, so we have gray-scale images. Note that the whole MNIST dataset is way larger (a set of 70000 images can be downloaded from here). The portion of the data we use is the one made available by the ScikitLearn Python library [8]. They depict hand-written digits and are stored together with the digits to which they correspond. Thus, this data set is suitable for a supervised learning approach. Our goal is to train a classifier which, given any such image as input, will recognize the corresponding digit. In general, a classifier is a function

$$f : \mathbb{R}^n \to D, \tag{1}$$

where D is a finite set of labels, whose cardinality we denote as K. In our case, we identify D with the standard basis of $\mathbb{R}^K$. This technique is called *one-hot encoding*. Any output digit $d \in \{0, 1, \ldots, 9\}$ is thus represented by the d-th column of the 10×10 identity matrix. The input space is instead $\mathbb{R}^{64}$, which means that we input the images to the network in the format of flattened arrays. Thus, our models will never have 2-d spatial knowledge.

II.2. Multi-class logistic regression

The most common and simple approach to this problem is multi-class logistic regression. Essentially, the goal is to construct a probability mass function (PMF) $p(C_k|\mathbf{x})$, where $\mathbf{x}$ is our input image and C_k denotes one of the corresponding possible categories (digits). Once such a function is found, we let our classifier be

$$f(\mathbf{x}) = \underset{C_k \in D}{\operatorname{argmax}}\{p(C_k|\mathbf{x}).\} \tag{2}$$

In order to construct the PMF, we associate a vector of weights w_k and a bias w_{k0} to every category C_k, and we let $a_k = w_k^\intercal \mathbf{x} + w_{k0} = \mathbf{w}^\intercal \phi$, where we have set $\phi = (1, \mathbf{x})$ and $\mathbf{w} = (w_{k0}, w)$. Then, we choose the following ansatz:

$$p(C_k|\mathbf{x}) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}, \tag{3}$$

which is known in the literature as the SoftMax function. We observe the resemblance to the Boltzmann distribution from statistical mechanics. With this assumption, it can be shown [1] that maximizing the likelihood of observing our data-set $\{\mathbf{x}^{(n)}, \mathbf{t}^{(n)}\}$ is equivalent to the minimization of the *cross-entropy*:

$$C(\mathbf{x}, \{\mathbf{w}_k\}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k^{(n)} \ln\Big(y_k^{(n)}\Big), \tag{4}$$

where N is the number of examples in the data-set, $t_k^{(n)}$ is the k-th component of $\mathbf{t}^{(n)}$, and finally $y_k^{(n)} = p(C_k|\mathbf{x}^{(n)})$. The cross-entropy is the cost function which we aim to minimize, and because it is a convex function of the parameters, a local minimum is going to be a global minimum[9]. Gradient methods, which are described in the section II.6, come in really handy in the search for local minima, for this reason it is useful to write down the gradient of C:

$$\nabla_{\mathbf{w}_k} C = \sum_{n=1}^{N} \left(y_k^{(n)} - t_k^{(n)}\right) \phi^{(n)} \tag{5}$$

For the purpose of writing a back-propagation code, it is also important to recall the following:

$$\frac{\partial y_\ell}{\partial a_k} = y_\ell(\delta_{\ell k} - y_k). \tag{6}$$

While logistic regression is achieved by simply feeding an affine tranformation of the inputs to the SoftMax function, more complicated models can be constructed. Namely, we can replace the raw input vector $\mathbf{x}$ in equation (3) with the activations of the second-to-last layer of a neural network. This is a typical way of employing deep learning in classification problems. We are now going to outline the structure of a neural network.

II.3. Neural Network architecture

A Neural Network is essentially a function:

$$\mathrm{NN} : \mathbb{R}^n \to \mathbb{R}^K, \tag{7}$$

which can be defined in terms of simpler function called neurons. A *neuron* is a real-valued function:

$$\text{neuron} : \mathbb{R}^p \to \mathbb{R}.$$

. To identify it we need:

1. A vector of weights $\mathbf{w} \in \mathbb{R}^p$ and a bias w_0.
2. An activation function $\sigma : \mathbb{R} \to \mathbb{R}$.

The output a of a neuron on an input $\mathbf{x}$ is then defined by the following equations:

$$z = \sum_{j=1}^{p} w_j x_j + w_0 \ , \tag{8}$$

$$a = \sigma(z). \tag{9}$$

In the Machine Learning jargon, a is often called *activation*. In a NN, neurons are organized in layers, which are functions

$$\mathrm{L}_\ell : \mathbb{R}^p \to \mathbb{R}^{n_\ell}$$

obtained by grouping together a certain number n_ℓ of independent neurons that have inputs in $\mathbb{R}^p$. The reason for the use of the subscript ℓ will become clear later. The j-th element of the output of a layer is simply the output of its j-th neuron. Usually, neurons in a certain layer are all equipped with the same kind of activation function. The weights and biases are instead different for all the neurons. Thus, a layer is ultimately defined by

1. A matrix of weights $\mathbf{W} \in \mathbb{R}^{n_\ell \times p}$ and a vector of biases $\mathbf{b} \in \mathbb{R}^{n_\ell}$.
2. An activation function σ.

From a coding point of view, it is convenient to express the action of a layer on an input vector $\mathbf{x}$ as follows:

$$\mathrm{L}_\ell(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{10}$$

where now σ is thought to be acting element-wise. Finally, a neural network as the one mentioned in equation (7) is a composition of layers, the first having inputs in $\mathbb{R}^n$ and the last having outputs in $\mathbb{R}^K$:

$$\mathrm{NN}(\mathbf{x}) = \mathrm{L}_M \circ \mathrm{L}_{M-1} \circ \cdots \circ \mathrm{L}_2 \circ \mathrm{L}_1(\mathbf{x}) \tag{11}$$

We label as M and n_ℓ the total number of layers and the number of neurons in layer ℓ, respectively. These are hyperparameters, which determine the complexity of our model. A multi-class classifier can be constructed by choosing the SoftMax as the activation function of the last layer. In particular, if we build such a NN with just one layer, we get exactly the multi-class logistic regression model which was described in section II.2. Now that we laid out the architecture of a neural network, it is apparent how large is the number of weights and biases that we need to optimize, in order to train our model. For the sake of clarity, we denote the whole set of parameters as $\boldsymbol{\beta}$. Also, the dependence of the output $\tilde{\mathbf{y}} = \mathrm{NN}(\mathbf{x})$ on such parameters is at first glance *really* intricate. Therefore, if we want to optimize a cost function $C(\boldsymbol{\beta}, \tilde{\mathbf{y}})$, such as a mean square error, we clearly see two elephants in the room:

1. Computing a closed form of the gradient ∇C is challenging, due to the nested structure of equation (11)
2. Solving the equation $\nabla C = 0$ and ensuring that it yields a minimum (and not, for example, a maximum or a saddle point) is even more challenging.

These two issues have been dealt with in the literature. The former is model-dependent, and in the case of NNs it is solved by the back-propagation algorithm. The latter is more general, and the most studied solution is the employment of gradient descent methods. These are the topics of sections II.6 and II.4

II.4. Back-propagation algorithm

The main reference for this section is [9]. Here, we spell out the back-propagation algorithm, which allows to compute the gradient of a cost function C with respect to the weights and the biases of a neural network. A detailed proof is available in the appendix REF. First, we introduce the following notation: w^ℓ_{jk} is the weight of the input k, for the j-th neuron of layer ℓ, whereas b^ℓ_j is the bias of the same neuron. Its activation a^ℓ_j is therefore defined by

$$a^\ell_j = \sigma(z^\ell_j), \tag{12}$$

where we have set

$$z^\ell_j = \sum_{k=1}^{n_{\ell-1}} w^\ell_{jk} a^{\ell-1}_k + b^\ell_j.$$

Finally, we introduce the quantity:

$$\Delta^\ell_j = \frac{\partial C}{\partial z^\ell_j} = \frac{\partial C}{\partial a^\ell_j}\sigma'(z^\ell_j), \tag{13}$$

where in the last equality the chain rule has been used. This is an auxiliary quantity which is going to be very helpful in a moment. Note that we have tacitly assumed that C is defined explicitly in terms of the activation of the last layer, a^M_j. Also, we implied that the activation function and its derivative are known for all layers. With this in mind, we are now ready to describe and understand the back-propagation algorithm:

1. **feed-forward step.** The activation of every neuron is computed, taking advantage of the layered structure of the network (equation 11).
2. Compute Δ^ℓ_j for the last layer ($\ell = M$), using equation (13). This is easily done thanks to our knowledge of the cost and the activation functions. A list of the activation function we implement is available in section II.5
3. **back-propagation step.** Compute Δ^ℓ_j for all layers using the following recursive relation:

$$\Delta^{\ell-1}_j = \left(\sum_{r=1}^{n_\ell} \Delta^\ell_r w^\ell_{rj}\right) \sigma'(z^{\ell-1}_j) \tag{14}$$

4. Compute the derivatives with respect to all parameters:

$$\frac{\partial C}{\partial w^\ell_{jk}} = \Delta^\ell_j a^{\ell-1}_k, \quad \frac{\partial C}{\partial b^\ell_j} = \Delta^\ell_j \ . \tag{15}$$

Remarkably, the proof of this algorithm relies only on the chain rule. The tools from real analysis made feasible what was seemingly an insanely hard task at a glance.

II.5. Activation functions

Here, we provide a list of the activation functions that we implemented in our codes, together with the corresponding derivatives. They have been widely studied in the literature, and more details about them can be found in [9].

1. **Sigmoid.** It is the function

$$\sigma(z) = \frac{1}{1+e^{-z}}; \tag{16}$$

after some algebraic manipulation it is possible to show that its derivative is

$$\frac{d\sigma}{dz} = \sigma(z)(1-\sigma(z)), \tag{17}$$

which allows to spare some FLOPS (remember that activation function are computed thousands of times during a typical training procedure).

2. **Linear.** This function is just the identity. Its derivative is equal to 1 regardless of the input.
3. **R**ectified **L**inear **U**nit (**ReLU**). It is the function

$$\sigma(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}, \tag{18}$$

its derivative being the Heaviside step function $\theta(z)$.

4. **Leaky ReLU.** It is defined as:

$$\sigma(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}, \tag{19}$$

α is now a small parameter.

5. **SoftMax.** This was described in equations (3), and (6). Notably it is a function that can not be applied element-wise, which means that the derivative is not a vector but a matrix.

II.6. Optimization algorithms

We hereby motivate the gradient descent method, and later we describe the different ways it can be implemented. Let $C : \mathbb{R}^n \to \mathbb{R}$ be a well-behaved cost

function, of which we want to find a local minimum. Consider its Taylor expansion around a point β_0, truncated to the first order:

$$C(\boldsymbol{\beta}) \approx C(\boldsymbol{\beta}_0) + \nabla C(\boldsymbol{\beta}_0) \cdot (\boldsymbol{\beta} - \boldsymbol{\beta}_0). \tag{20}$$

Let $\boldsymbol{\beta} = \boldsymbol{\beta}_0 + \eta\hat{u}$, where $\hat{u}$ is unit vector which forms a certain angle θ with the gradient and η is a real number. We can then write the difference

$$C(\boldsymbol{\beta}) - C(\boldsymbol{\beta}_0) = \nabla C(\boldsymbol{\beta}_0) \cdot \eta\hat{u} = \eta \left|\nabla C(\boldsymbol{\beta}_0)\right| \cos(\theta). \tag{21}$$

Once η and $\boldsymbol{\beta}_0$ are fixed, this difference has negative sign and maximum absolute value for $\theta = \pi$, that is, when $\hat{u}$ points in a direction opposite to the gradient. In other words, $-\nabla C(\boldsymbol{\beta}_0)$ is the direction along which our cost function decreases most rapidly. It is therefore natural to search for a local minimum by the following iterative scheme: a guess $\boldsymbol{\beta}_0$ is made, a tolerance ε is set, and then we compute

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \eta \nabla C(\boldsymbol{\beta}_i), \tag{22}$$

iterating while

$$\left|C(\boldsymbol{\beta}_{i+1}) - C(\boldsymbol{\beta}_i)\right| > \varepsilon. \tag{23}$$

This algorithm is called gradient descent (GD). More details about what we discuss next can be found in [9] and [10]. There are two main computational issues which arise naturally when we try to implement it:

1. Computing gradients can be computationally expensive, especially when the number of training examples is very large. Is there a way of compute gradients more efficiently?

2. The smaller the learning rate, the slower is the convergence of the algorithm. On the other hand, too large a learning rate could result in our parameters diverging. Can we speed up the convergence in some way, without divergence occurring?

The first question is tackled by the introduction of the stochastic gradient descent (SGD) method, which we now outline. It is very often the case in ML that our cost function is defined as a sum of individual loss functions over the training set $\mathcal{D}$:

$$C(\mathcal{D}, \beta) = \sum_{x_i \in \mathcal{D}} c(x_i, \beta), \tag{24}$$

It is always possible to partition $\mathcal{D}$ into a family of disjoint subsets $\{B_k\}_{k=1}^{M}$, each roughly of the same size and using a stochastic procedure. These sets are referred to as "mini-batches", and serve us to compute an estimate of the gradient:

$$\nabla C = \sum_{x_i \in B_k} \nabla c(x_i). \tag{25}$$

At every descent step, B_k is changed. A whole loop through all mini-batches is called an *epoch* We remark that by definition, regardless of the batch size, every epoch has the same computational cost. Thus, the number of epochs N_{epochs} taken to reach convergence is a rough measurement of the number of FLOPs required by the descent. Also, we note that, instead of using a tolerance criterion for convergence like equation 23, it is common practice to let N_{epochs} be an hyperparameter of the training, which needs tuning. The computational advantage of this technique is clear: for a given number of FLOPs, we increase the number of gradient descent steps by a factor of M, though they will be less accurate than the ones taken by computing the gradient of the cost over all data points. The latter technique is referred to as "plain GD" henceforth. In order to answer our second question, we can rewrite equation (22) as

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \Delta\boldsymbol{\beta}_i. \tag{26}$$

Where we define $\Delta\boldsymbol{\beta}_i$ as follows:

$$\Delta\boldsymbol{\beta}_i = \eta \nabla C(\boldsymbol{\beta}_i). \tag{27}$$

We now look at various ways of correcting the term $\Delta\boldsymbol{\beta}_i$, with the aim of improving the convergence speed. Note that each of these methods can be used independently of whether we use stochastic gradient descent or not. Also, when we do not use any of these methods we say that we perform a "raw" gradient descent.

Gradient descent with momentum

Momentum-based GD modifies equation (27) by replacing

$$\Delta\boldsymbol{\beta}_i = \gamma\Delta\boldsymbol{\beta}_{i-1} + \eta \nabla C(\boldsymbol{\beta}_i) \tag{28}$$

In this way we have included a running average of the gradients which were computed at all previous steps. γ satisfies $0 \leq \gamma < 1$ and determines the rate of decay of this sort of "memory" of the previous gradients. This becomes apparent if we expand the recursive relation (28) in the following way:

$$\Delta\boldsymbol{\beta}_t = \eta \left[\sum_{k=0}^{t} \gamma^k \nabla C(\boldsymbol{\beta}_{t-k})\right], \tag{29}$$

and we clearly see that the term in square brackets is indeed a weighted average of all of the gradients computed previously. In the jargon of probability theory, any sample average is an estimate of the first order moment of the corresponding random variable. This is why we say that this is a first-order method. We note that as t approaches infinity the total contribution of $\nabla C(\boldsymbol{\beta}_i)$ approaches a value of

$$\nabla C(\boldsymbol{\beta}_i) \cdot (1 + \gamma + \gamma^2 + \gamma^3 + \dots) = \nabla C(\boldsymbol{\beta}_i) \cdot \frac{1}{1-\gamma}. \tag{30}$$

We re-parametrize this method in terms of the momentum variable μ, which is related to γ:

$$\gamma = 1 - \frac{1}{\mu}, \tag{31}$$

such that the total contribution of $\nabla C(\boldsymbol{\beta}_i)$ approaches a value of $\mu \nabla C(\boldsymbol{\beta}_i)$.
In the following, we describe three more approaches, which require us to compute a sort of "running average" of the squared elements of the gradient: $(\partial_{\beta_j} C)^2$. Thye are respectively called AdaGrad, RMSprop and ADAM. For the sake of clarity we introduce the following notation for the gradient:

$$\nabla C(\boldsymbol{\beta}_t) = \mathbf{g}_t, \tag{32}$$

and we will denote its individual elements as $g_{t,i}$.

AdaGrad

Adaptive **Grad**ient requires us to keep track of the following auxiliary quantities at every step of the descent:

$$G_{t,i} = \sum_{t'=0}^{t} (g_{t',i})^2, \tag{33}$$

which form a vector $\mathbf{G}_t$, which by definition can be written recursively as

$$\mathbf{G}_t = \mathbf{G}_{t-1} + \mathbf{g}_t \circ \mathbf{g}_t \tag{34}$$

. The parameters update is then computed as follows:

$$\Delta \boldsymbol{\beta}_t = \eta \frac{1}{\sqrt{\mathbf{G}_t + \epsilon}} \circ \mathbf{g}_t \tag{35}$$

where every operation involving vectors is intended to be performed element-wise, and in particular $\circ$ denotes the Hadamard product. ϵ is a small parameter which is introduced to avoid divergence, for $\mathbf{G}_{ti} = 0$ at the first iteration.

RMSprop

Root **M**ean **S**quare **prop**agation corrects the estimate of the second order moment of the gradient in the following way:

$$\mathbf{G}_t = \gamma \mathbf{G}_{t-1} + (1-\gamma) \mathbf{g}_t \circ \mathbf{g}_t. \tag{36}$$

with this new definition of $\mathbf{G}_t$, the update to the parameters is computed as prescribed by equation (35). In this method, γ is usually set to 0.9[10].

ADAM

ADAptive **M**oment estimation keeps track of both the first and the second order moment of the gradient, in a manner which is identical to RMSprop for the latter, and analogous for the former. In particular, at every step, we update the following quantities, which are initialized to zero:

$$\begin{aligned} \mathbf{M}_t &= \gamma_1 M_{t-1} + (1-\gamma_1)\mathbf{g}_t \\ \mathbf{G}_t &= \gamma_2 G_{t-1} + (1-\gamma_2)\mathbf{g}_t \circ \mathbf{g}_t \end{aligned} \tag{37}$$

where again γ_1, γ_2 are parameters chosen in the interval $[0, 1)$. They are not directly used in the computation of the step, because, especially at the beginning of the descent, they will be biased towards zero. To counteract this behaviour, the following corrections are introduced:

$$\begin{aligned} \hat{\mathbf{M}}_t &= \frac{\mathbf{M}_t}{1-(\gamma_1)^t} \\ \hat{\mathbf{G}}_t &= \frac{\mathbf{G}_t}{1-(\gamma_2)^t} \end{aligned},$$

Finally, the update to the parameters is computed as follows:

$$\Delta \boldsymbol{\beta}_t = \frac{\mathbf{M}_t}{\sqrt{\mathbf{G}_t} + \epsilon} \eta. \tag{38}$$

Usual values of the parameters are $\gamma_1 = 0.9$ and $\gamma_2 = 0.999$.

A visual comparison between the effects of all these methods is available in appendix A. There we also compare the effect of SGD, opposed to plain GD.

II.7. Learning rate schedules

In every expression of $\Delta \boldsymbol{\beta}_t$ in the last section, we assumed a fixed learning rate η. It is also possible to set a schedule for it, so that it decreases as long as the descent goes on. We list here some of the most common choices, which we also implement in our libraries.

- **Hyperbolic scheduler.** We replace η at the time step k with

  $$\eta_k = \frac{\tau}{\tau + k} \eta \tag{39}$$

  τ is a parameter which sets the amount of steps required to decrease the learning rate by a factor of 2.

- **Linear.** We can set the learning rate to decrease linearly with the number of steps taken:

  $$\eta_k = \left(1 - \frac{k}{\tau}\right) \eta, \tag{40}$$

  here τ represents number of steps after which the learning rate becomes negative. This type of schedule has to be used carefully for this reason.

- **Exponential decay.** Last, but not least, we can set:

  $$\eta_k = \eta e^{-k/\tau}, \tag{41}$$

  so that after $k = \tau$ steps the learning rate decreases by about 63%.

III. RESULTS AND DISCUSSION

Once our library for the various gradient descent methods has been tested, we use it to train a Neural Network with N layers on the MNIST dataset, for $N \in \{0, 1\}$. As we already argued, the case $N = 0$ reduces to a simple, multi-class logistic regression. For this reason we sometimes refer to this model as a network, even if it is a trivial instance. In every case, we let our cost function to be the cross entropy (equation 4), to which we add an L_2 regularisation term. This amounts to add the term $\lambda\boldsymbol{\beta}_k$ to the gradient when it is computed at any given step k. Apparently, we need to tweak many hyperparameters which enter the training procedure. To do so, we split the data-set into three parts, which we term *train*, *test* and *validation*[1] set, their proportion being 60%/20%/20%. Whenever we want to optimize a generic hyperparameter h that enters the training procedure, we scan a grid of values h_{grid} and we select the best one as described in algorithm 1.

Algorithm 1 Optimizing any hyperparameter h

for $h \in h_{\text{grid}}$ **do**

perform Gd using the train set;

evaluate and store score on the test set;

display the results of the grid search;

If we test a small set of values, we just pick the one which scores best, whereas for fine scans the plots are noisy, so we look at them and choose the best hyperparameter by eye.

Note that we use both the cross-entropy and the accuracy to assess the quality of a model, the latter being defined as the fraction of points that are properly classified

$$\text{accuracy} = \frac{N_{\text{correct}}}{N_{\text{tot}}}, \tag{42}$$

We always split our training set into 22 batches to perform SGD, and we deem that the most relevant hyperparameters are λ, η and N_{epochs}. For them, our tuning strategy is the following:

1. we optimize the hyperparameters $(\eta, N_{\text{epochs}})$ together, which means that we use algorithm 1 looping through all possible combinations $(\eta_j, N_{\text{epochs},k})$ belonging to a proper pre-defined grid.

2. with the selected pair $(\eta, N_{\text{epochs}})$, we optimise the penalty λ.

3. with this value of λ, we perform again a grid-search for the learning rate and the number of epochs, finally choosing the (new) best pair $(\eta, N_{\text{epochs}})$

We run this for all different optimisers, namely we compare:

1. Raw SGD with a learning rate schedule

2. Momentum SGD with a learning rate schedule

3. AdaGrad SGD without a learning rate schedule

4. RMSprop SGD without a learning rate schedule

5. ADAM SGD without a learning rate schedule

We have chosen not to use a learning rate schedule with the approaches that divide the gradient by an accumulating value. [2]

Before optimizing the main parameters with the strategy we have just outlined, we try to choose a suitable learning rate schedule, and we also want to select the corresponding values of τ. Moreover, we search for the best parameter μ of the momentum optimiser.
We do this only in the case of the logistic regression classifier, which is the first model we train. Later on, when moving to the neural network with one hidden layer, we keep these parameters fixed, although this might not be the optimal choice.

III.1. Logistic regression

We now go into the details of the results which we find while optimizing the logistic regression.

Momentum

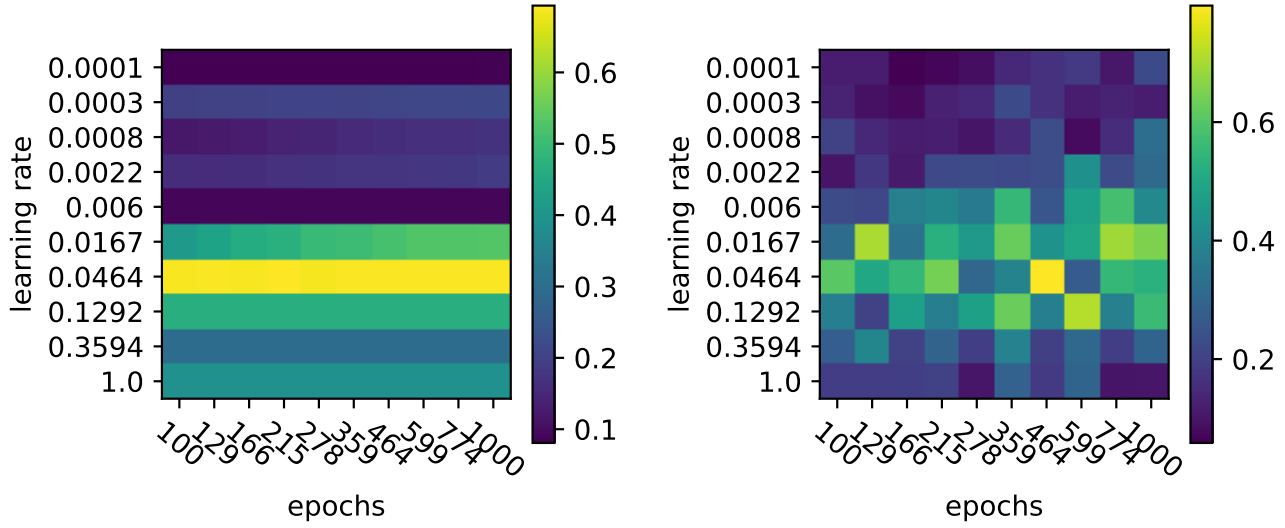
We scan values of the momentum μ ranging from 1 to 3.5 in steps of 0.5. For each μ_i train networks with learning rates and numbers of epochs in the ranges $\left[10^{-4}, 1\right]$ and $[10, 100]$ respectively. At this stage, we use no learning rate schedule and we set $\lambda = 10^{-4}$. We select the momentum $\mu = 1.5$, which yields the smallest cross entropy, over all combinations of η and N_{epochs} tested. We decide to use this value for the rest of the optimisations.

Learning rate schedule

Similarly we also optimise the value of τ for each of the learning rate schedules and then we select the

[1] Please note that, as opposed to the common terminology, we call "test" set what we use to tune the hyperparameters, and "validation" set what we *only* use to check the performance of the model gotten *after* selecting *all* hyperparameters.

[2] We have since realised that for RMSprop and ADAM the value of $\mathbf{G}$ ($\hat{\mathbf{G}}$) doesn't keep increasing but instead it reaches a mostly constant magnitude somewhere close to $\mathbf{g} \circ \mathbf{g}$. This means that they too should have had a learning rate schedule.



(a) Momentum GD

(b) AdaGrad

Figure 1: Plots that help us tuning the learning rate and the number of epochs, in the case of a logistic classifier. In both figures, the accuracy on the test set is shown using a color scale. We select the values which give highest accuracy. Interestingly, the outcome of momentum GD (figure a) seems to be way less affected by the number of epochs, if compared to the performance of the AdaGrad (figure b). This is not so unreasonable if we remind that we have fixed the parameter τ of the hyperbolic learning rate schedule.

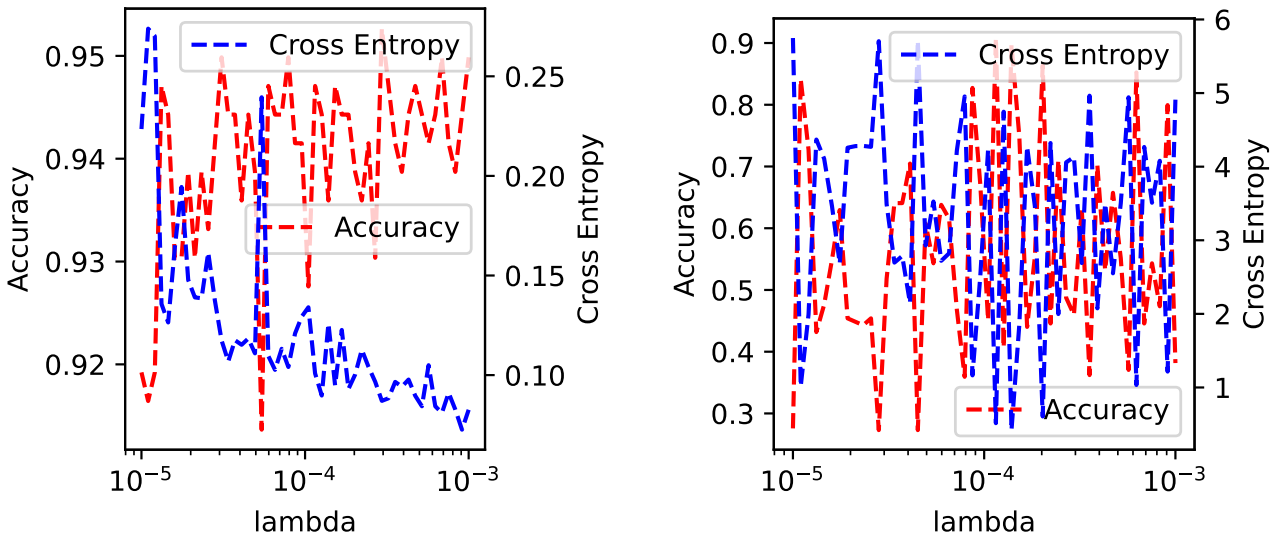
schedule and the value of τ which give best results. We use the same ranges as before for the learning rate and the number of epochs, and we use the same value of λ. We find that for both approaches the hyperbolic optimiser works best, and we find values of $\tau = 215$ and $\tau = 464$ for the raw and momentum SGD respectively.

Learning rate, penalty and number of epochs

We are now ready to start the core of our optimization strategy. We report here only some results, but all of them can be found in the notebook. After the first tuning of η and N_{epochs}, we have 5 colormaps of the kind reported in figure 1, which specifically depict Momentum GD and AdaGrad. In both of them, we found $\eta = 0.05$ to be the best learning rate and the best N_{epochs} equal to 100 and 470. Analogous plots and choices are made for all the other optimisers. Having done that, we optimize lambda, and again in figure 2 we report the result of the grid search, now showing both the cross-entropy and the accuracy. While for the RMSprop we clearly see a pattern, in the AdaGrad case, the "signal" in the plot is quite small compared to the noise due to the stochasticity of the algorithm. This makes it quite hard to understand the effect of regularization in this case. Nevertheless, we try look at the trend and make an educated guess about which λ is best. We choose $\lambda = 2 \times 10^{-4}$ in both cases. Again, analogous plots and choices are made for all optimisers. Finally, we tune η and N_{epochs} again, now with the optimal value of λ that we found. Figure 3 shows the grid-search results for AdaGrad and RMSprop. The final choices we make are $\eta = 0.06$, $N_{\text{epochs}} = 200$ for AdaGrad and $\eta = 0.002$, $N_{\text{epochs}} = 950$ for the RMSprop. With the model we've trained, we finally assess its quality by measuring the accuracy on the *validation set*, which has *never* entered neither the training procedure nor the choices of the hyperparameters in any way. The results are described in section III.3.

III.2. Neural network

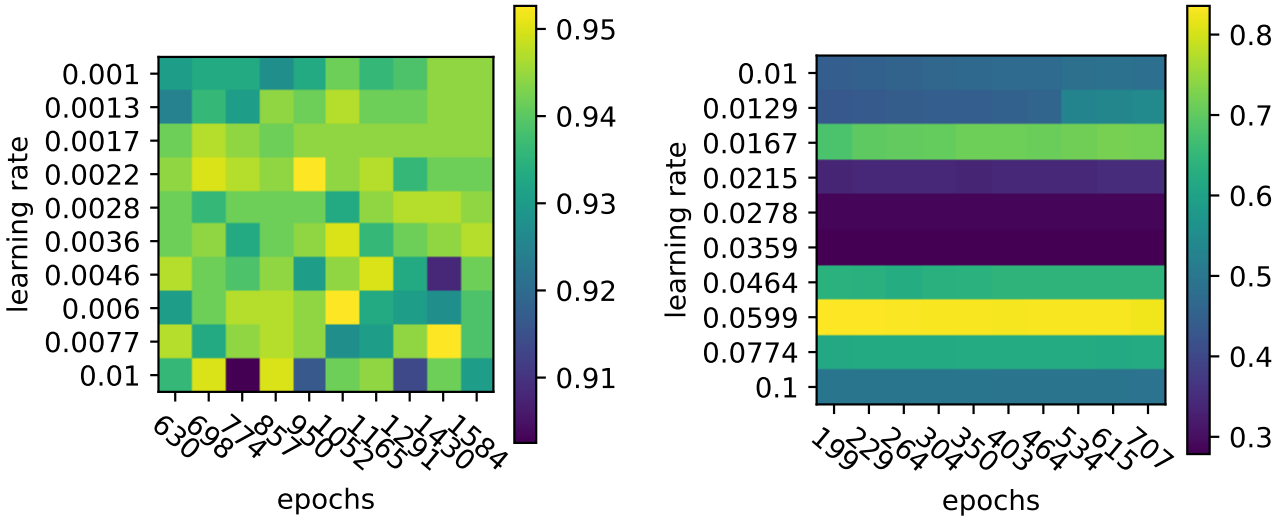
We compare the logistic regression to a neural network, so after optimising the former we now also have to optimise the latter.We always consider a NN with one hidden layer, made of 25 neurons. For the momentum and the learning rate schedules, we use the same values that we found with the logistic regression.
Before we get started with all of the other optimisations we consider the activation function. We test both ReLU and sigmoid for the hidden layer, but we choose to use sigmoid as ReLU seemingly suffers from numerical instability. When scanning a range of learning rates and regularisation parameters the network would at some point reach a state where the cost function returns NaN values which then corrupt all of the weights. We are unsure about why this happens and our best guess is currently that the inputs of the softmax function get either too big or too small. We select the sigmoid activation function, for which we had no such issues. We



(a) RMSprop

(b) AdaGrad

Figure 2: Scores of the logistic regression classifier on the test set, for different values of the L_2 regularisation parameter λ, in the case of RMSprop and AdaGrad. Observe that the accuracy is on average higher for the former than for the latter. The plot in the AdaGrad case is quite noisy, so that here we try to look at the general trend, rather than to the exact value that gives best scores.



(a) RMSprop

(b) AdaGrad

Figure 3: Tuning η and the number of epochs for the second time, after the choice of regularisation λ. The plots are analogous to the ones in figure 1. We now zoomed into smaller intervals for the two parameters. Despite the RMSprop plot looks more noisy, note the range of the color scale is way narrower in this case, and it shows a much better accuracy.

repeat the core of our optimization strategy. After the first tuning of η and N_{epochs}, we again have 5 colormaps of the kind reported in figure 4, which specifically depict Momentum GD and AdaGrad. The pattern for large learning rates in raw SGD can be explained by the learning rate schedule which brings the learning rate down to the optimum if the initial learning rate is too high, while for initial learning rates that are too small the learning rate schedule prevents the accuracy from increasing as much for larger values of N_{epochs}.

For raw SGD we choose $\eta = 4$ and $N_{\text{epochs}} = 500$ and for RMSprop we choose $\eta = 0.01$ and $N_{\text{epochs}} = 500$. Analogous plots and choices are made for all the other optimisers. Then in the next step we optimize lambda, and again in figure 5 we report the result of the grid search, now showing both the cross-entropy and the accuracy. While for the raw SGD we clearly identify the trend, in the RMSprop case the plot is quite noisy. It seems like for RMSprop the value of λ does not play an important role. We choose $\lambda = 10^{-3}$ for raw SGD and $\lambda = 4 \times 10^{-4}$ for RMSprop. Again, analogous plots and choices are made for all optimisers. Finally, we tune η and N_{epochs} again, now with the optimal value of λ that we found. Figure 6 shows the grid-search results for AdaGrad and RMSprop. The final choices we make are $\eta = 4.6$, $N_{\text{epochs}} = 1900$ for Raw SGD and $\eta = 0.003$, $N_{\text{epochs}} = 500$ for RMSprop. With the parameters we've chosen, we finally assess the networks quality by measuring the accuracy on the *validation set*, which has *never* entered either the training procedure or the choices of the hyperparameters in any way.

III.3. Comparison of models and explainability

We first compare the performance on the validation set for the two models we have trained: the logistic regression and the Neural Network with one hidden layer. In figure 7 we show a summary of their scores. It is interesting to note that the accuracies were as high as 97% on the test set which was used to select the best hyperparameters. The models perform about 7% worse when they predict the labels of never seen data. The NN with a hidden layer has better predictions in all cases, so it is clearly the best of the two models. We also note that raw GD, momentum GD and AdaGrad are performing a lot worse in the logistic regression case, with accuracies as low as 40%. We now try to explain what our model does by plotting the weights of the output layer in a gray-scale. The result for logistic regression is shown in figure 8. The results clearly shows that the shapes of any given digit k tend to have more important weights in the neuron that is supposed to output $p(C_k|\mathbf{x})$. Remarkably, this happens even if the model has no $2d$ spatial knowledge, since we input the images as flattened arrays. In the case of the NN with one hidden layer, it results to be way harder to intepret the weights of the hidden layer. They can be visualized in appendix B

IV. CONCLUSION

There are apparently many things which can be improved in this study, which only scratched the surface of a wide ocean of possibilities, especially regards the tuning of the hyper-parameters. First, we note that we could improve our search for τ, schedule and momentum parameters, because we used the same parameters for both models, even if they were optimized only for the logistic regression. Secondly, we could ask: when choosing the best hyperparameters, is it better to consider the accuracy or the cross entropy? This is an open question: their trends were strongly correlated, so it's reasonable to assume that this choice does not make that much of a difference. However, we reckon that often the results which gave the best cross-entropy did not exactly coincide with what yielded highest accuracy on the test, so studying the consequence of different choices might be interesting. Moreover, when the plots of the scores are noisy, instead of guessing the best hyperparameters by eye as we did, a better approach would be to repeat the stochastic gradient descent many times, and then average the results. Furthermore, we saw that the ReLU activation function did not produce satisfying gradient descent convergence, maybe because its outputs became too large. Further investigation are needed to settle this question. At the end, after training our models, we conclude that the NN with a hidden layer performs a bit better than the logistic regression. This suggests that increasing the complexity of the network, for example by adding another layer, might lead to a further improvement of the model. It would be interesting to investigate this in further research. We tried to interpret what our model does on the input images to find the output, finding that this can be intuitively explained for a logistic regression, whereas it is more obscure for the NN. Another possible way of analysing would be to try and make images of the derivatives of the output with respect to the variables in input. We also note that CNNs would probably be more suitable for the task of labelling images, since they can recognize spatial patterns in 2 or more dimensions. In that case, the prediction might be easier to understand. Taking everything into consideration, we may claim that we had a deep insight on the cornerstones of modern Machine Learning: gradient optimization and Neural Networks.

Appendix A: Code Testing

We implement the various gradient descent methods described in section II.6 in a library of classes. We test this code in the case of a linear regression to a straight line. A toy data-set is generated in the following manner: a set of real numbers $\{x_i\}_{i\in I}$ is sampled from a uniform distribution in the interval $[0, 1]$ and the corresponding noisy targets are then generated:

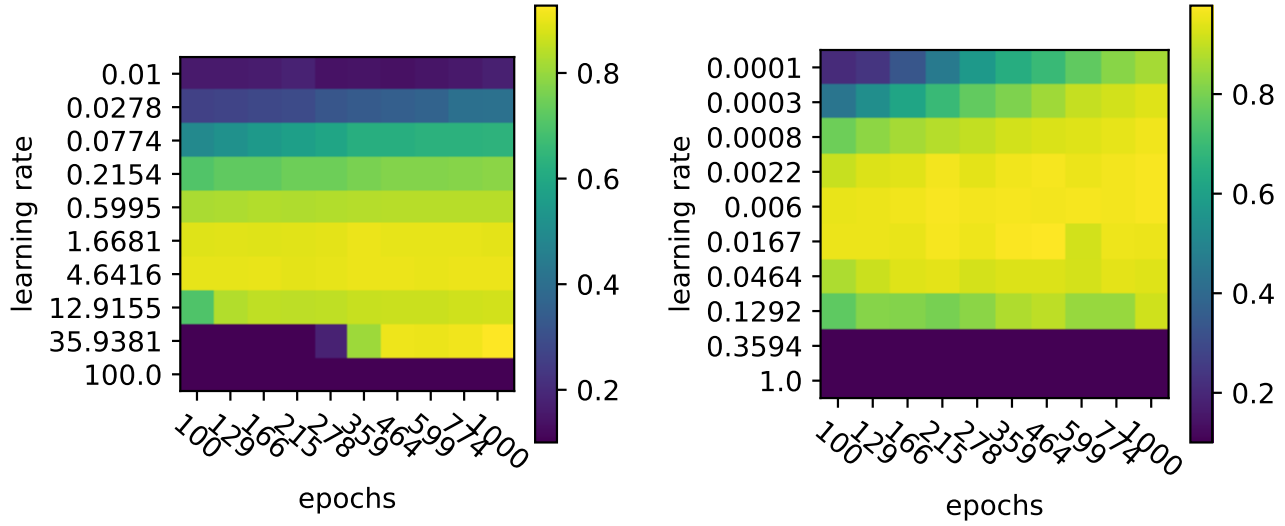
$$y_i = \beta_0 + \beta_1 x_i + \alpha\epsilon_i, \tag{A1}$$

where α is a fixed parameter and ϵ_i is a gaussian uncorrelated noise, with mean 0 and variance 1. We fit a linear model to this dataset:

$$\tilde{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i, \tag{A2}$$

and we search for the parameters which minimise the mean square error:

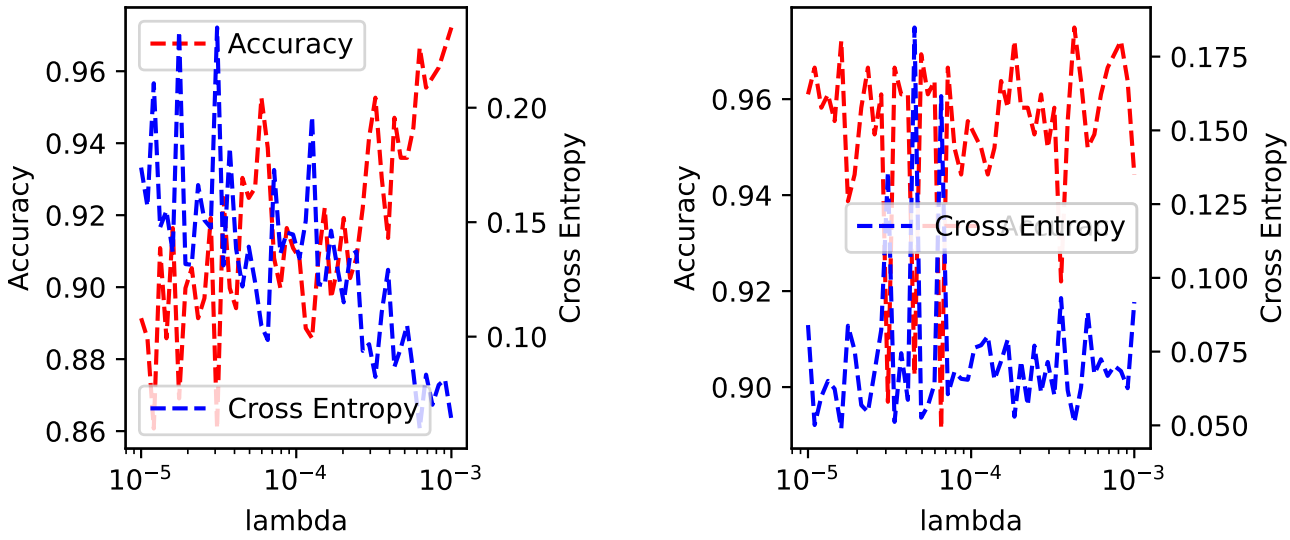
$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \tilde{y}_i)^2. \tag{A3}$$



(a) Raw SGD

(b) RMSprop

Figure 4: Plots that help us tuning the learning rate and the number of epochs, in the case of a neural network with one hidden layer. In both figures, the accuracy on the test set is shown using a color scale. We select the values which give highest accuracy. We notice that with raw SGD the number of epochs doesn't seem to make a large difference for learning rates smaller than the optimum while for learning rates that are too large the convergence is similar but happens at a later epoch. For RMSprop this pattern seems to be somewhat reversed.

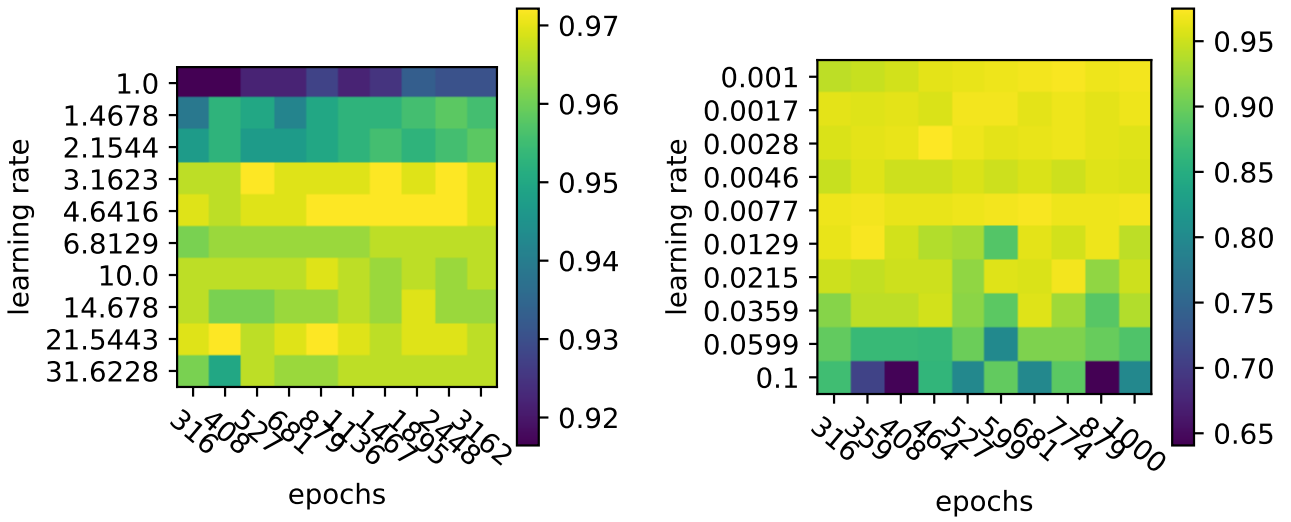


(a) Raw SGD

(b) RMSprop

Figure 5: Accuracy and cross entropy of the neural network with one hidden layer on the test set, for different values of the L_2 regularisation parameter λ, in the case of Raw SGD and RMSprop.

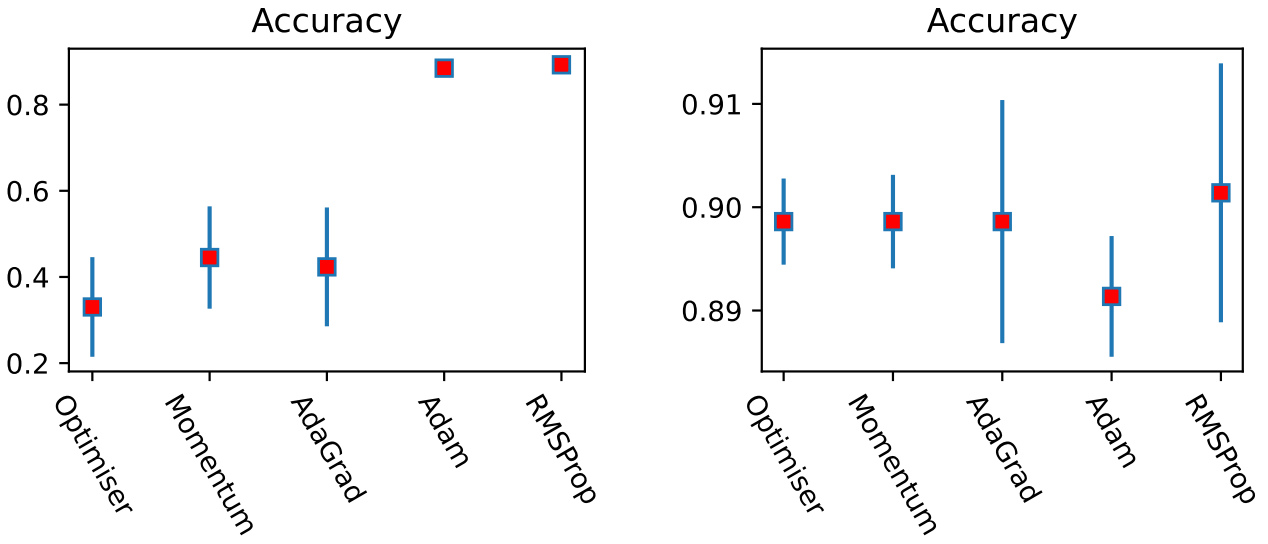
Instead of using the analytical solution to this problem, which amounts to a simple Ordinary Least Squares regression, we employ gradient descent to search for the optimal parameters. We start with an initial guess for the parameters of our model, namely, with the same notation as in section II.6, $\hat{\boldsymbol{\beta}}_{\mathbf{0}} = (0, 0)$. Then we use all of the different gradient descent methods we described, and check that they actually converge to the right parameters. Doing this check is not a difficult task, for we expect the optimal parameters to be roughly equal



(a) Raw SGD

(b) RMSprop

Figure 6: Tuning η and the number of epochs for the second time, after the choice of regularisation λ. The plots are analogous to the ones in figure 4. We now zoomed into smaller intervals for the two parameters. Note that the range of the color scale is much smaller in this case.



(a) Logistic regression

(b) NN with one hidden layer

Figure 7: Accuracies of the two trained models, with the different optimisers. The error bars show the standard deviation of the accuracies gotten in 5 different iterations of the training procedure.

to β_0, β_1, which we used to generate the data set. [3] Having defined a cost function as a sum of individual loss functions allows for a stochastic gradient descent approach. Moreover, the low complexity of our model makes it easy to visualize an exhaustive summary of the results in a 2D plot, which shows the trajectory of our descent in the space of parameters. These results for both stochastic and plain GD are visible in figure 9, for all the methods we tested. In all cases, we see that our parameters converge to the right values. The effect of stochasticity is apparent: when using

[3] Because of the noise in the dataset, this is not exactly the analytical minimum, but since $\mathbb{E}(\hat{\boldsymbol{\beta}}) = \boldsymbol{\beta}$, we expect it to be very close to it for a large enough dataset.

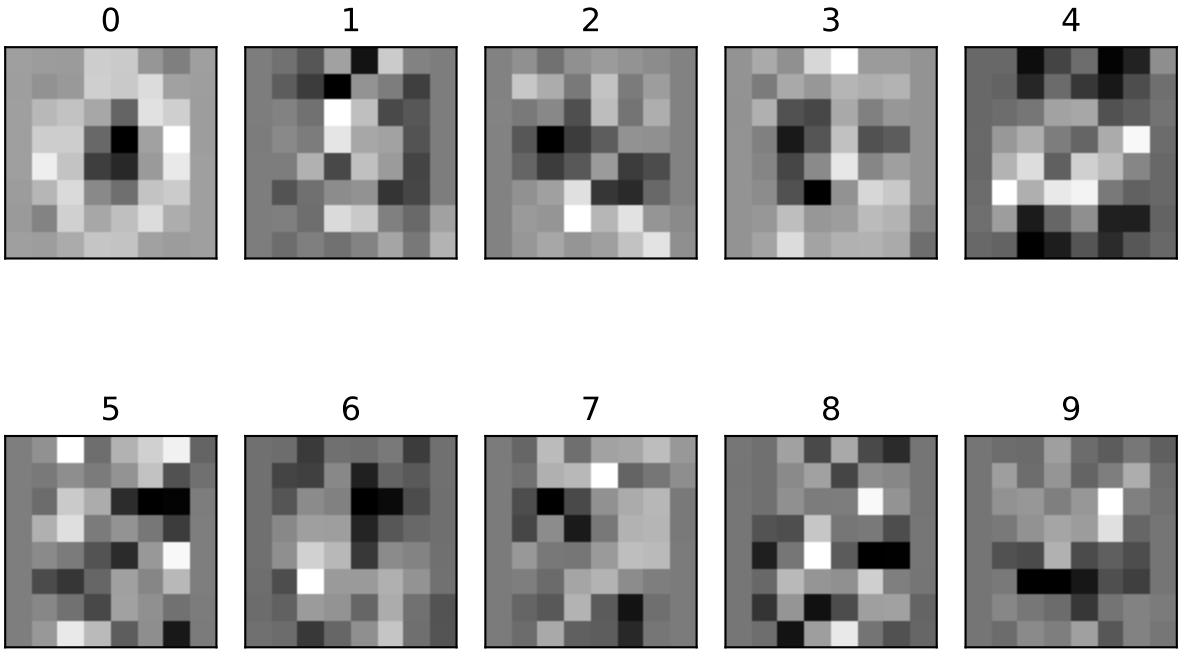
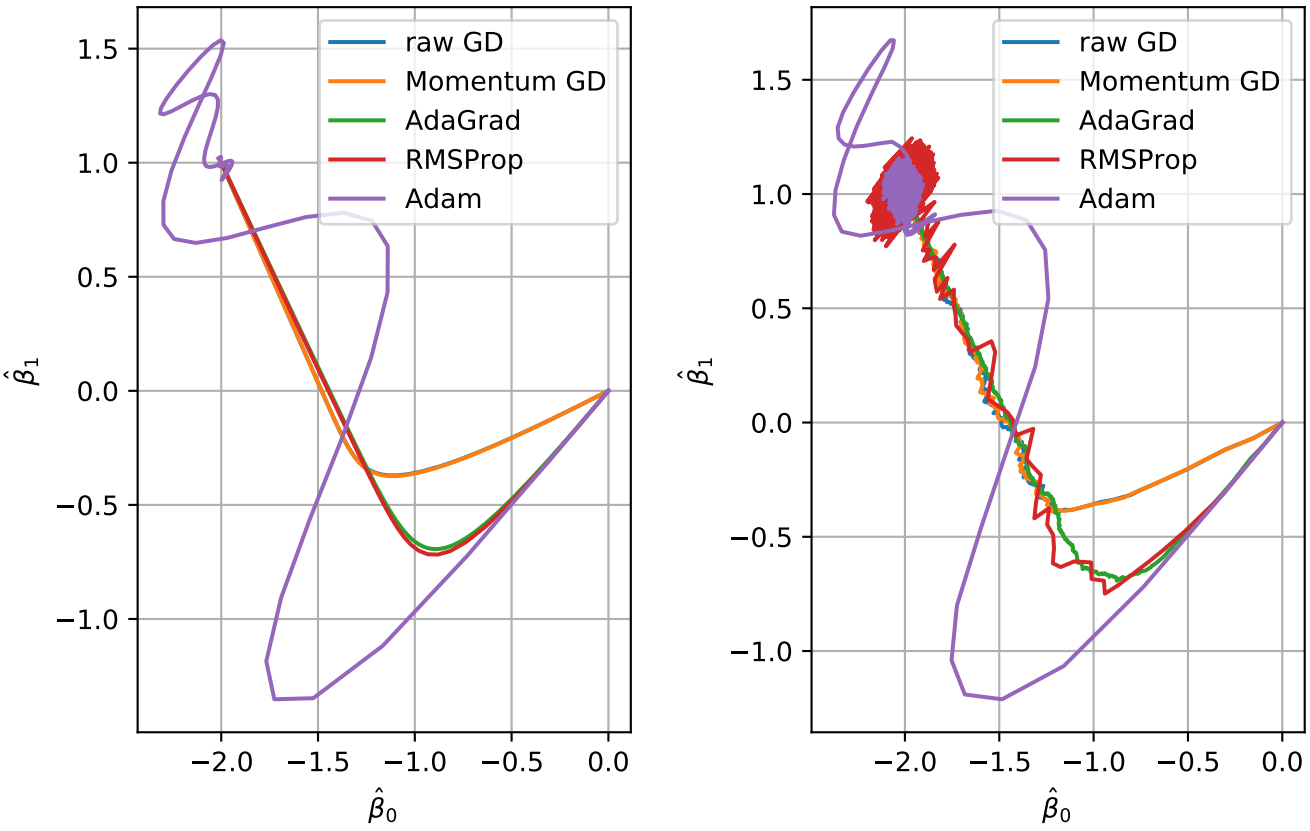


Figure 8: Weights of the output layer of the logistic regression classifier. This result is analogous to the one obtained by Mehta et al.[9]. We observe that the plots resemble the handwritten digits.

less data points to compute the gradient, the descent path becomes more butchered. Interestingly, the shape of the trajectory gotten by the Adam method seems to be only one which is almost unaffected. This could be explained if we look at equation (37) that Adam is the only method which replaces the gradient computed at every step with a weighted, running average of the most recent gradients. Despite being less precise at every step, SGD offers a gain in terms of computational cost. We can have a confirmation of this by looking at table I, where we roughly estimate the computational speed-up gotten by SGD as the ratio

$$\Xi = \frac{N_{\text{FLOPs}}(\text{plain GD})}{N_{\text{FLOPs}}(\text{SGD})}, \tag{A4}$$

where the notation is self-explanatory and the way we get this estimate of Ξ is described in the table caption. RMSProp is curiously the only method which yields $\Xi < 1$, so it seems to perform worse when we introduce stochasticity in the training. Considerable fluctuations are observed at the end of the descent, both for RMSProp and Adam. This could be explained by looking carefully at the way these methods compute the update to the parameters, namely equations (35) and (38), respectively. We first focus on the denominator, and in particular we observe that $\mathbf{G}_t$ is a running average of all the gradients computed during the training, the least recent ones having smaller and smaller weights in this average. Therefore, when we get close to the minimum of the cost function, the denominator is supposed to shrink, with a lower bound defined by the parameter ε. On the other hand, the numerator of both expressions is expected to decrease, and it is going to be roughly of the same magnitude of the denominator. In short, ignoring ε for a moment, the update to the parameters is gotten by multiplying the learning rate η by something which is roughly of order 1. And in fact, the observed fluctuations in figure 9 *are* of the order of the learning rate, which was always set to $\eta = 0.1$.



(a) Non stochastic gradient descent. At each step, the whole dataset was used to compute the gradient of the cost function.

(b) Stochastic gradient descent. The data-set was split into 100 batches of size 10, to compute the gradient at every step.

Figure 9: Comparison of the trajectories of parameters, both in stochastic and non-stochastic gradient descent methods. In all cases we set the following parameters, with the same notation of section II and of this appendix. $\eta = 0.1$, $\mu = 4$ for the momentum GD, $N = 1000$ data points in the training set, and finally the parameters of the model that generates data $\beta_0 = -2$, $\beta_1 = 1$, $\alpha = 0.1$. In both cases, the parameters start from the origin and converge to the proper values. As expected, we clearly see that the trajectory of the descent looks a lot smoother in the plain GD than in SGD.

method	plain GD	stochastic GD	Ξ
raw GD	602	213.88	2.8
momentum	562	146.96	3.8
AdaGrad	1538	62.29	24.8
RMSProp	61	247.46	0.25
Adam	93	70.88	1.3

Table I: Number of GD steps required for convergence, divided by the number of batches N_b. In plain GD we have obviously $N_b = 1$, whereas in SGD we have $N_b = 100$. The numbers displayed in the should be therefore roughly proportional to the number of FLOPs that are required for the training. We used a tolerance of $\varepsilon = 10^{-6}$ for the change in the cost function at the last step. The corresponding trajectories of the parameters are depicted in figure 9

Appendix B: Explainability of hidden layer

For the sake of completeness, we include in figure 10 a plot in gray-scale of the weights associated with the elements of the hidden layer of the trained Neural Network. It is very hard to find any meaningful pattern in these plots.

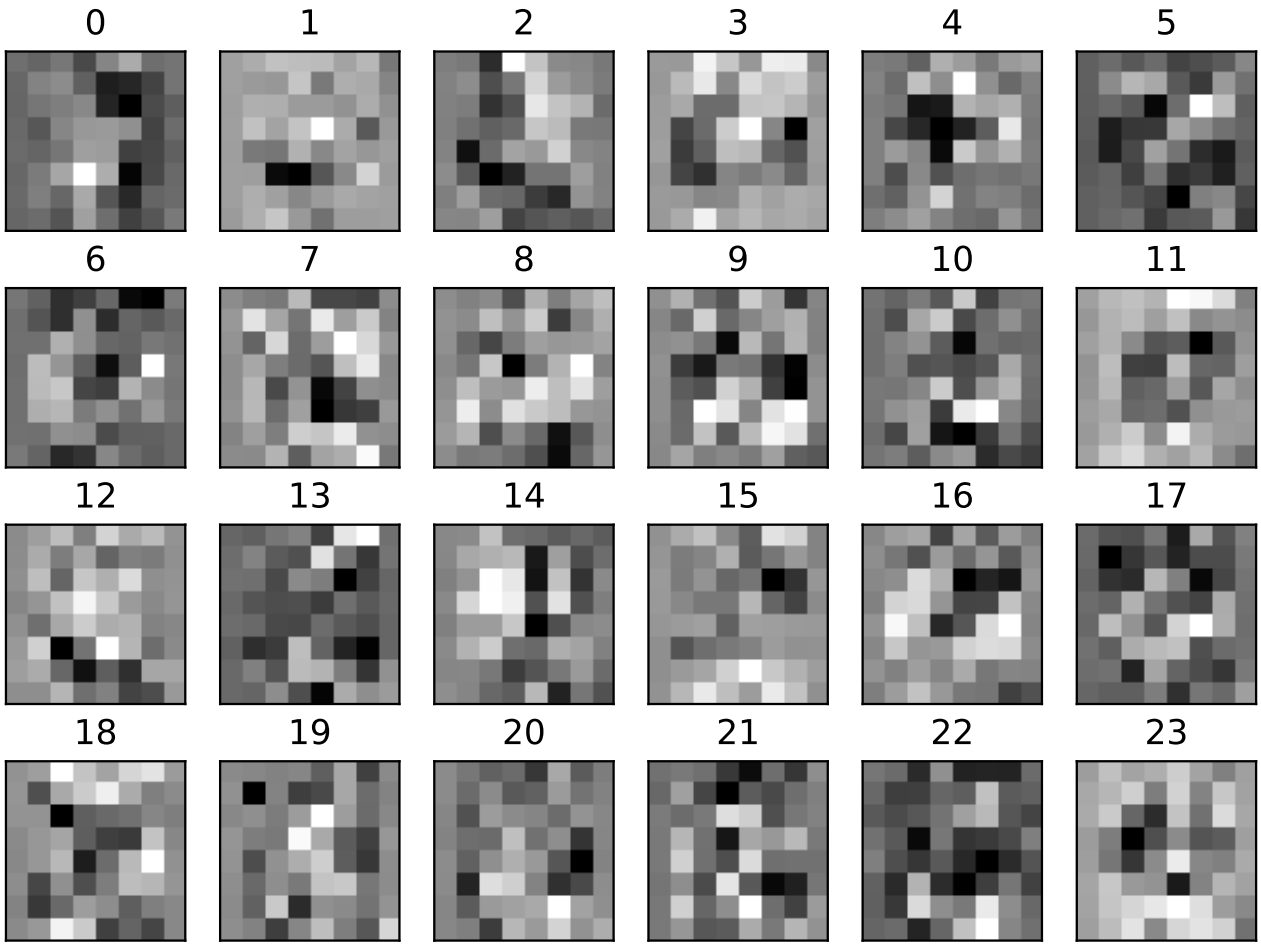


Figure 10: Weights of the best trained network.

[1] C. Bishop, *Pattern Recognition and Machine Learning*, ser. Information Science and Statistics. Springer, 2006. [Online]. Available: https://books.google.no/books?id=qWPwnQEACAAJ

[2] Q. V. Le, M. Ranzato, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," 2011. [Online]. Available: https://arxiv.org/abs/1112.6209

[3] G. Baltus, J. Janquart, M. Lopez, H. Narola, and J.-R. Cudell, "Convolutional neural network for gravitational-wave early alert: Going down in frequency," *Physical Review D*, vol. 106, no. 4, aug 2022. [Online]. Available: https://doi.org/10.1103%2Fphysrevd.106.042002

[4] A. A. Pol, G. Cerminara, C. Germain, M. Pierini, and A. Seth, "Detector monitoring with artificial neural networks at the CMS experiment at the CERN large hadron collider," 2018. [Online]. Available: https://arxiv.org/abs/1808.00911

[5] M. Rigo, B. Hall, M. Hjorth-Jensen, A. Lovato, and F. Pederiva, "Solving the nuclear pairing model with neural network quantum states," 2022. [Online]. Available: https://arxiv.org/abs/2211.04614

[6] G. Sanderson. [Online]. Available: https://www.3blue1brown.com/lessons/neural-networks

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[9] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to machine learning for physicists," *Physics Reports*, vol. 810, pp. 1–124, may 2019. [Online]. Available: https://doi.org/10.1016%2Fj.physrep.2019.03.001

[10] S. Ruder, "An overview of gradient descent optimization algorithms," 2016. [Online]. Available: https://arxiv.org/abs/1609.04747