UNIVERSITETET
I OSLO

PROJECT 2: Diagnosing permafrost state in Pangnirtung (Nunavut, Canada)
An analysis from satellite and ground-based thermic measurements

Advanced Remote Sensing - Natural Hazards and Climate Change
GEO4520 - University of Oslo

Pietro PERRONE
pietrope@uio.no

November 20, 2025

GitHub link to the project repository:
https://github.com/p-perrone/UiO_MachineLearning/tree/main/ml_project2
Link to DeepSeek chat:
https://chat.deepseek.com/share/i38u41hr9ln9ruwnxh

Abstract

In this study, a custom feedforward neural network (NN) was developed and tested for regression and classification tasks. The performance of the NN was compared with traditional linear Ordinary Least Squares (OLS) regression, showing superior approximation with two hidden layers, despite slightly lower R^2 values compared to high-degree polynomial OLS. Three Stochastic Gradient Descent (SGD) optimizers—plain SGD, RMSProp, and ADAM—were evaluated for convergence speed and stability. RMSProp and ADAM converged faster and were more robust across learning rates, though they required longer computation times than plain SGD. The custom NN was benchmarked against `scikit-learn` 's `MLPRegressor` and autograd-based gradient computation, demonstrating comparable performance but significantly longer execution time. L1 and L2 regularizations were applied, improving model generalization when penalty parameters were below critical thresholds. Finally, the NN was employed for multi-class classification on the MNIST-784 dataset, achieving an overall accuracy of 0.92 using Leaky ReLU and Softmax activations with ADAM optimization. The study demonstrates that a well-tuned NN, combined with appropriate optimizer selection and regularization, can outperform traditional regression methods and achieve strong classification performance.

Contents

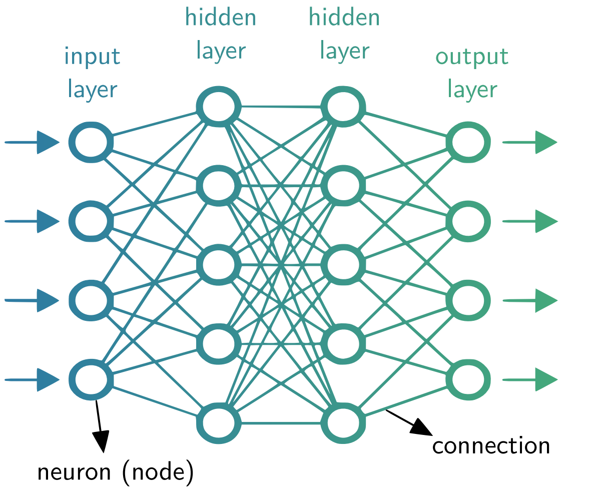

Figure 1: Neural Network | A typical structure.

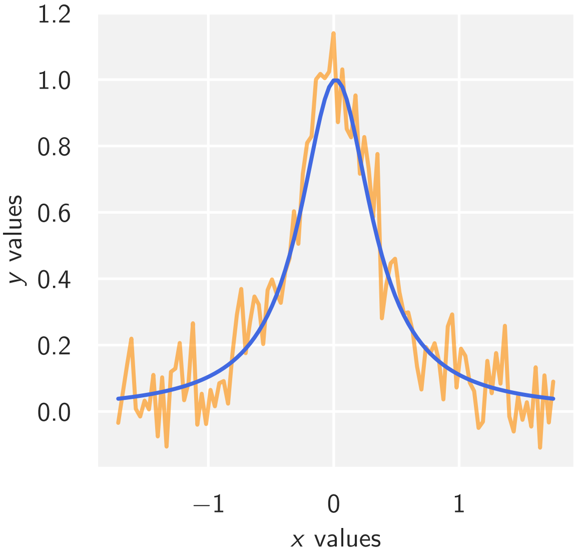

Figure 2: Runge function | In orange with an added noise of normal distribution $\mathcal{N}(0,\ 0.1)$. The x values have already been scaled.

1 Introduction

An Artificial Neural Network is a computational model that emulates the functioning of the human brain, in the way it can process several pieces of information in parallel, resulting in a form of "intelligence" (Wang, 2003).

A typical Neural Network (NN) is represented in Figure 1. It generally consists of connected units, called *nodes* or *neurons*. Every node receives a specific *signal*, *i.e.*, a real number, from its connected nodes. Nodes can be grouped into specific layers, and the signal travels from the input layer to the output layer, passing through an arbitrary number of *hidden layers* (Bishop, 2006). Before being transmitted from one layer to the following one, the signal is modulated by a non-linear function, called the *activation function*, which can be specified for each layer. Formally, an NN transforms an input $\mathbf{X} \in \mathbb{R}^{n \times d}$ into an output vector $\mathbf{y}$ through successive linear transformations, where every node is multiplied by *weights*, and activation functions are applied (Goodfellow et al., 2016). The weights are the parameters used in an NN for approximating the target function or dataset. The multiplication of each node's value is usually followed by the addition of a real number (a *bias*), in order to avoid the transmission of zero signals throughout the network. A cost function is usually computed at the end, in order to assess the agreement between the target and output prediction of the NN.

The simplest possible neural network is given by the *perceptron model*, conceived by Rosenblatt (1958). It is a type of linear classifier that takes binary inputs, weights them by real numbers, and yields a binary output. More complicated neural networks can generally support a higher number of layers and nodes, accept a multidimensional input, and yield a non-binary output. In this project, in particular, we focus on the so-called *multilayer perceptrons* (MLPs). Another type of neural network can be the *feed-forward* neural network (FFNN). In this model, the signal is transmitted always in a single direction, forward through the layers, from the input to the output. If each node in a layer is connected to all the other nodes in the following layer, this corresponds to a *fully connected* FFNN (Haykin, 1994). The operation by which the weights and biases are optimized is called *backpropagation*, which consists of calculating the gradients of the cost function with respect to weights and biases starting from the last layer; training can then be performed in order to minimize these gradients and find the optimal values of these parameters, using optimization algorithms such as gradient descent (Goodfellow et al., 2016).

The aim of this project is to build NN code in Python that can perform both forward feeding and backpropagation. We will test this model initially on a simple one-dimensional dataset, given by a cluster of samples computed with the Runge function for an input array $\boldsymbol{x} \in [-1, 1]$ (Figure 2). Several types of activation functions and layer depths will be tested.

A more complicated fit will then be performed on a dataset of handwritten digits, the MNIST-784. This consists of 70 000 digit samples, each of size 28×28 pixels. The objective will be to train the coded NN to make it recognize and classify these digits correctly. An analysis of the goodness of such classification will also be performed.

2 Theory and methods

2.1 The universal approximation theorem

The cornerstone of Neural Network theory in machine learning is the *universal approximation theorem*. The formulation by (Cybenko, 1989) states that:

Theorem 2.1: Universal approximation

Let σ be any continuous sigmoidal function such that:

$$\lim_{z \to \infty} \sigma(z) = 1 \quad \text{and} \quad \lim_{z \to -\infty} \sigma(z) = 0$$

Given a continuous function $F(\boldsymbol{x})$ on the unit cube $[0,1]^d$ and $\epsilon > 0$, there exists a one-hidden-layer neural network $f(\boldsymbol{x};\boldsymbol{\Theta})$ with parameters $\boldsymbol{\Theta} = (\boldsymbol{W}, \boldsymbol{b})$ where $\boldsymbol{W} \in \mathbb{R}^{m\times m}$ and $\boldsymbol{b} \in \mathbb{R}^m$, such that

$$|F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})| < \epsilon \quad \forall \boldsymbol{x} \in [0,1]^n$$

In other words, any continuous function $y = F(\boldsymbol{x})$ supported on the unit cube in d dimensions can be approximated by a one-layer sigmoidal network to arbitrary accuracy and errors.

A generalization of this theorem to any non-constant, bounded activation function was established by (Hornik, 1991). Dealing with the definition of the expectation value $\mathbb{E}[|F(\boldsymbol{x})|^2]$:

$$\mathbb{E}[|F(\boldsymbol{x})|^2] = \int_{\boldsymbol{x}\in D} |F(\boldsymbol{x})|^2 p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} < \infty. \tag{1}$$

it was concluded that

$$\mathbb{E}[|F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})|^2] = \int_{\boldsymbol{x}\in D} |F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})|^2 p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} < \epsilon. \tag{2}$$

2.2 Structure of a Neural Network

2.2.a The Feed-forward algorithm

Simple perceptron model with 1-D input

A simple FFNN will consist of an input vector, a single hidden layer, and an output vector. We call:

- $\boldsymbol{x}^\intercal = (x_1, x_2, \dots x_n) \in \mathbb{R}^n$ the input;
- $\boldsymbol{z}^\intercal = (z_1, z_2, \dots z_m) \in \mathbb{R}^m$ the weighted vector representing the hidden layer;
- $\tilde{\boldsymbol{y}}^\intercal = (\tilde{y}_1, \tilde{y}_2, \dots \tilde{y}_m) \in \mathbb{R}^m$ the activated, output layer.

The activation y_i of each i-th neuron is a weighted sum of inputs, passed through an activation function f. The weights and biases correspond to the parameters of the model:

$$\boldsymbol{\Theta} = (\boldsymbol{W}, \boldsymbol{b}) \tag{3}$$

where, in this case, $\boldsymbol{W} \in \mathbb{R}^{m\times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$. In the case of the simple perceptron model we have

$$\boldsymbol{z} = \sum_{j=1}^{m}\sum_{i=1}^{n} w_{ij} x_i \tag{4}$$

$$\tilde{\boldsymbol{y}} = f(\boldsymbol{z}) \tag{5}$$

where f is the activation function, x_i represents the input from neuron i in the input layer, and w_{ij} is the weight to input i. Since this is a simple perceptron, then $f(\boldsymbol{z})$ directly corresponds to the output of the NN, which will be passed to the chosen cost function $\mathcal{C}(\boldsymbol{\Theta})$. This simple architecture is shown in Figure 3 (Goodfellow et al., 2016).

Generalization to a multi-layer network with multiple inputs

A typical neural network takes as input a matrix $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, where n is the number of samples and d is the number of features. For example, a network that approximates a function $\boldsymbol{t} = u(\boldsymbol{x}, \boldsymbol{y})$ from 100 samples takes as input a matrix $[\boldsymbol{x}, \boldsymbol{y}] \in \mathbb{R}^{100\times 2}$.

The network can have an arbitrary number of layers $l = 1, 2, \dots, L$. We denote by

$$(\boldsymbol{a}^{(l)})^\intercal = (a_1^{(l)}, a_2^{(l)}, \dots, a_{m_l}^{(l)})$$

the vector of activations at layer l, where m_l is the number of neurons in that layer. These activations serve as input to the next layer $l+1$.

For a given neuron j in layer l, the pre-activation value is

$$z_j^{(l)} = \sum_{i=1}^{m_{l-1}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}, \tag{6}$$

where m_{l-1} is the number of neurons in the previous layer. In matrix form, this can be written compactly as

$$\boldsymbol{Z}^{(l)} = \boldsymbol{A}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}, \tag{7}$$

Here, we have

- $\boldsymbol{A}^{(l-1)} \in \mathbb{R}^{n\times m_{l-1}}$: activations (outputs) from the previous layer, where n is the number of samples and m_{l-1} is the number of neurons in layer $l-1$;
- $\boldsymbol{W}^{(l)} \in \mathbb{R}^{m_{l-1}\times m_l}$: weight matrix connecting layer $l-1$ to layer l;
- $\boldsymbol{b}^{(l)} \in \mathbb{R}^{1\times m_l}$: bias vector for layer l, added to the n samples during computation;
- $\boldsymbol{A}^{(l)} \in \mathbb{R}^{n\times m_l}$: activations of layer l, obtained after applying the activation function elementwise to $\boldsymbol{Z}^{(l)}$:

$$\boldsymbol{A}^{(l)} = f^{(l)}(\boldsymbol{Z}^{(l)})$$

Implementation

A possible pseudocode for such an algorithm is Algorithm 1

2.2.b The Backpropagation algorithm

A consistent result from the NN calculations can only be achieved with properly trained parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. As in every machine learning algorithm, the optimization of such parameters can be performed by minimizing the gradient of the cost function with respect to these parameters. *Backpropagation* is the operation of calculating

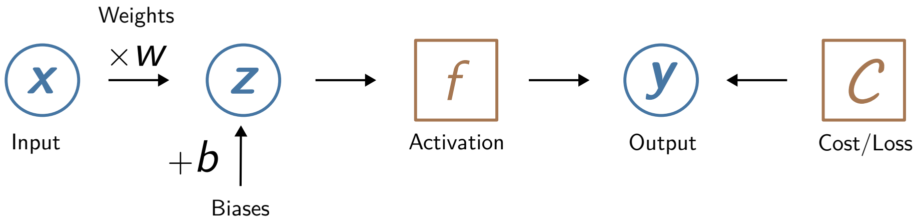

Figure 3: Simple perceptron model | In blue the different layers, in orange the functions.

Algorithm 1 Feedforward

```
Require: Input $\boldsymbol{X}$, weights $\{\boldsymbol{W}^{(l)}\}_{l=1}^{L}$, biases $\{\boldsymbol{b}^{(l)}\}_{l=1}^{L}$,
    activation functions $\{f^{(l)}\}_{l=1}^{L}$
Ensure: Network output $\boldsymbol{A}^{(l)}$
  $\boldsymbol{A}^{(0)} \leftarrow \boldsymbol{X}$
  for $l = 1$ to $L$ do
      $\boldsymbol{Z}^{(l)} \leftarrow \boldsymbol{W}^{(l)}\,\boldsymbol{A}^{(l-1)} + \boldsymbol{B}^{(l)}$      ▷ Linear weighting
      $\boldsymbol{A}^{(l)} \leftarrow f^{(l)}(\boldsymbol{Z}^{(l)})$      ▷ Activation
  end for
  return $\boldsymbol{A}^{(l)}$
```

these gradients. For the layer l:

$$\boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta}^{(l)}) = \begin{bmatrix} \partial_{\boldsymbol{W}^{(l)}}\mathcal{C} \\ \partial_{\boldsymbol{b}^{(l)}}\mathcal{C} \end{bmatrix} \tag{8}$$

The output of this calculation is given by a number of gradients that correspond to the total number of layers in the NN:

$$\left[\boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta}^{(1)}), \boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta}^{(2)}), \dots, \boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta}^{(L)})\right]$$

From Equation (6), we know that every signal at the j-th node of layer l depends on the output value of the activation function computed for the signal at layer $l-1$, *i.e.* $\boldsymbol{A}^{(l-1)}$. In turn, $\boldsymbol{A}^{(l-1)}$ is given by the activation of $\boldsymbol{Z}^{(l-1)}$, which is a function of $\boldsymbol{A}^{(l-2)}$, and so on. The dependence of each layer on the previous ones, in fully connected NNs, allows computing the gradients of the cost function with respect to the parameters of any layer by simply applying the chain rule, starting from the output layer and propagating to all the hidden layers in reversed order (*backpropagating*, namely).

Formalism of backpropagation

Let's consider a generic cost function, such as the *Mean Squared Error*. For the last layer L, we have:

$$\mathcal{C}(\boldsymbol{\Theta}^{(L)}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(a_i^{(L)} - y_i\right)^2 \tag{9}$$

We now want to derive the expression for the derivative of the cost function with respect to weights of the last layer, $\dfrac{\partial\mathcal{C}(\boldsymbol{\Theta}^{(L)})}{\partial\boldsymbol{W}^{(L)}}$, and thus we can apply the chain rule:

$$\frac{\partial\mathcal{C}}{\partial w_{ij}^{(L)}} = \frac{\partial\mathcal{C}}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}}$$

The two derivatives respectively correspond to:

$$\frac{\partial\mathcal{C}}{\partial a_j^{(L)}} = a_j^{(L)} - y_j \tag{10}$$

$$\frac{\partial a_j^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_j^{(L)}\left(1 - a_j^{(L)}\right)a_i^{(L-1)}. \tag{11}$$

The overall derivative, then, reads:

$$\frac{\partial\mathcal{C}(\boldsymbol{\Theta}^{(L)})}{\partial w_{ij}^{(L)}} = \left(a_j^{(L)} - y_j\right)a_j^{(L)}\left(1 - a_j^{(L)}\right)a_i^{(L-1)}. \tag{12}$$

In order to simplify the formalism, we can define an auxiliary factor $\boldsymbol{\delta}^{(L)}$. First, we have that, for the j-th node:

$$\delta_j^{(L)} = a_j^{(L)}\left(1 - a_j^{(L)}\right)\left(a_j^{(L)} - y_j\right) = f'\left(z_j^{(L)}\right)\frac{\partial\mathcal{C}}{\partial a_j^{(L)}} \tag{13}$$

and, using vector notation again, we have:

$$\boldsymbol{\delta}^{(L)} = f'(\boldsymbol{Z}^{(L)}) \odot \frac{\partial\mathcal{C}}{\partial\boldsymbol{A}^{(L)}} \tag{14}$$

where $\odot$ represents the element-wise (Hadamard) product. Finally, we can write the derivative of the cost function with respect to the weights of the last layer:

$$\frac{\partial\mathcal{C}(\boldsymbol{\Theta}^{(L)})}{\partial w_{ij}^{(L)}} = \delta_j^{(L)}a_i^{(L-1)} \tag{15}$$

For the biases, the expression can be similarly derived:

$$\frac{\partial\mathcal{C}(\boldsymbol{\Theta}^{(L)})}{\partial b_j^{(L)}} = \delta_j^{(L)} \tag{16}$$

since the only difference from the derivative with respect to the weights is the term $\dfrac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$, which is, in this case, equal to 1. We then have the expression of the gradient of the cost with respect to the parameters of the last layer.

In order to obtain the expression of backpropagation at a generic layer, we can first generalize the auxiliary factor

$\boldsymbol{\delta}$ at layer l. Expanding Equation (14), we have:

$$\boldsymbol{\delta}^{(l)} = \left(f^{(l)}\right)' \odot \left(\boldsymbol{W}^{(l+1)}\right)^{\intercal} \cdot \left(f^{(l+1)}\right)' \odot \cdots$$
$$\cdots \odot \left(\boldsymbol{W}^{(L-1)}\right)^{\intercal} \cdot \left(f^{(L-1)}\right)' \odot \left(\boldsymbol{W}^{(L)}\right) \cdot \left(f^{(L)}\right)' \odot \boldsymbol{\nabla}\mathcal{C}(\boldsymbol{A}^{(L)})$$

And, specifically,

$$\delta_j^{(l)} = \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

which basically corresponds to the application of the chain rule recursively from layer $l+1$ to layer l. k refers to the index of any node in layer $l+1$. Recalling Equation (6), we can finally derive a general expression for $\delta^{(l)}$:

$$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} \, w_{kj}^{(l+1)} \, f'\left(z_j^{(l)}\right) \qquad (17)$$

and in vector notation:

$$\boldsymbol{\delta}^{(l)} = \boldsymbol{\delta}^{(l+1)} \boldsymbol{W}^{(l+1)} f'(\boldsymbol{Z}^{(l)})$$

Finally, the gradient of the cost function with respect to weights and biases at layer l can be expressed as follows:

$$\boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta}^{(l)}) = \begin{bmatrix} \partial_{\boldsymbol{W}^{(l)}} \mathcal{C} \\ \partial_{\boldsymbol{b}^{(l)}} \mathcal{C} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\delta}^{(l)} \boldsymbol{A}^{(l-1)} \\ \boldsymbol{\delta}^{(l)} \end{bmatrix}. \qquad (18)$$

Implementation

Note that, in order to run backpropagation, the feedforward algorithm A possible pseudocode for the backpropagation algorithm is shown in the Algorithm 2.

Algorithm 2 Backpropagation

```
Require: Feedforward activations $\{\boldsymbol{A}^{(l)}\}_{l=0}^{L}$, pre-activations $\{\boldsymbol{Z}^{(l)}\}_{l=1}^{L}$, weights $\{\boldsymbol{W}^{(l)}\}_{l=1}^{L}$, cost $\mathcal{C}$
Ensure: Gradients $\boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta}^{(l)})$
Initialize $\boldsymbol{\delta}^{(l)} = 0$ for all $l$
$L \leftarrow$ number of layers
                                        ▷ Compute output layer error
$\boldsymbol{\delta}^{(L)} \leftarrow \frac{\partial \mathcal{C}}{\partial \boldsymbol{A}^{(L)}} \odot f^{(L)\prime}(\boldsymbol{Z}^{(L)})$
for $l = L - 1$ down to 1 do
    $\boldsymbol{\delta}^{(l)} \leftarrow (\boldsymbol{\delta}^{(l+1)} \boldsymbol{W}^{(l+1)\intercal}) \odot f^{(l)\prime}(\boldsymbol{Z}^{(l)})$
end for
                          ▷ Compute gradients for weights and biases
for $l = 1$ to $L$ do
    $\frac{\partial \mathcal{C}}{\partial \boldsymbol{W}^{(l)}} \leftarrow (\boldsymbol{A}^{(l-1)})^{\intercal} \boldsymbol{\delta}^{(l)}$
    $\frac{\partial \mathcal{C}}{\partial \boldsymbol{b}^{(l)}} \leftarrow \sum_i \boldsymbol{\delta}_i^{(l)}$
end for
return $\left\{ \frac{\partial \mathcal{C}}{\partial \boldsymbol{W}^{(l)}}, \frac{\partial \mathcal{C}}{\partial \boldsymbol{b}^{(l)}} \right\}_{l=1}^{L}$
```

2.2.c *Training the Neural Network*

Once the gradients have been computed, then the NN can be trained. The optimization can be performed by minimizing the gradients iteratively with machinery such as the gradient descent algorithm (Hastie et al., 2009). This method has been more deeply explained in project 1's report. However, we will recall its basis and apply its formalism to NNs.

At each iterations of the gradient descent, the backpropagation (and, implicitly, the feedforward) algorithms must be called. The update rule is then given by, for each node j at layer (l):

$$\begin{aligned} w_{ij}^{(l)} &\leftarrow w_{ij}^{(l)} - \eta \delta_j^{(l)} a_i^{(l-1)} \\ b_j^{(l)} &\leftarrow b_j^{(l)} - \eta \delta_j^{(l)} \end{aligned}$$

with η the learning rate.

In the frame of this project, we will use Stochastic Gradient Descent, as it allows a much faster learning with bigger input datasets and multi-layer networks (Goodfellow et al., 2016). This method consists in performing the learning on a different subset of the whole dataset (*batch*) at each iteration, randomly defined and of constant size m. The iterative training scheme is summarized in the following pseudocode. Training is performed over all batches, which are defined by a shuffled index, and repeated for a number of iterations referred to as *epochs*.

Algorithm 3 Stochastic Gradient Descent training

```
Require: training data $\boldsymbol{X}, \boldsymbol{y}$, cost $\mathcal{C}$, learning rate $\eta$
Ensure: optimized parameters
for each epoch do
    shuffle indices for batches
    for each batch $(\mathbf{x}_b, \mathbf{y}_b)$ do
        backpropagate $\boldsymbol{g} \leftarrow \boldsymbol{\nabla}\mathcal{C}(\boldsymbol{\Theta})$
        update parameters $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} - \eta \boldsymbol{g}$
    end for
end for
return $\boldsymbol{\Theta}$
```

We will test three different approach to the definition of the learning rate:

- constant learning rate throughout the iterations;
- learning rate update with the root means of squared gradients: RMSProp (Tieleman & Hinton, 2012);
- learning rate update with a combination of gradients moving averages and momentum: ADAM (Kingma & Ba, 2017).

2.3 Activation functions

So far, we addressed the activation function of the specific layer (l) with a general expression $f^{(l)}$. Depending on the objective of the model and the types of used data, different activation functions can actually be employed. We will present five of these, that will be used in our

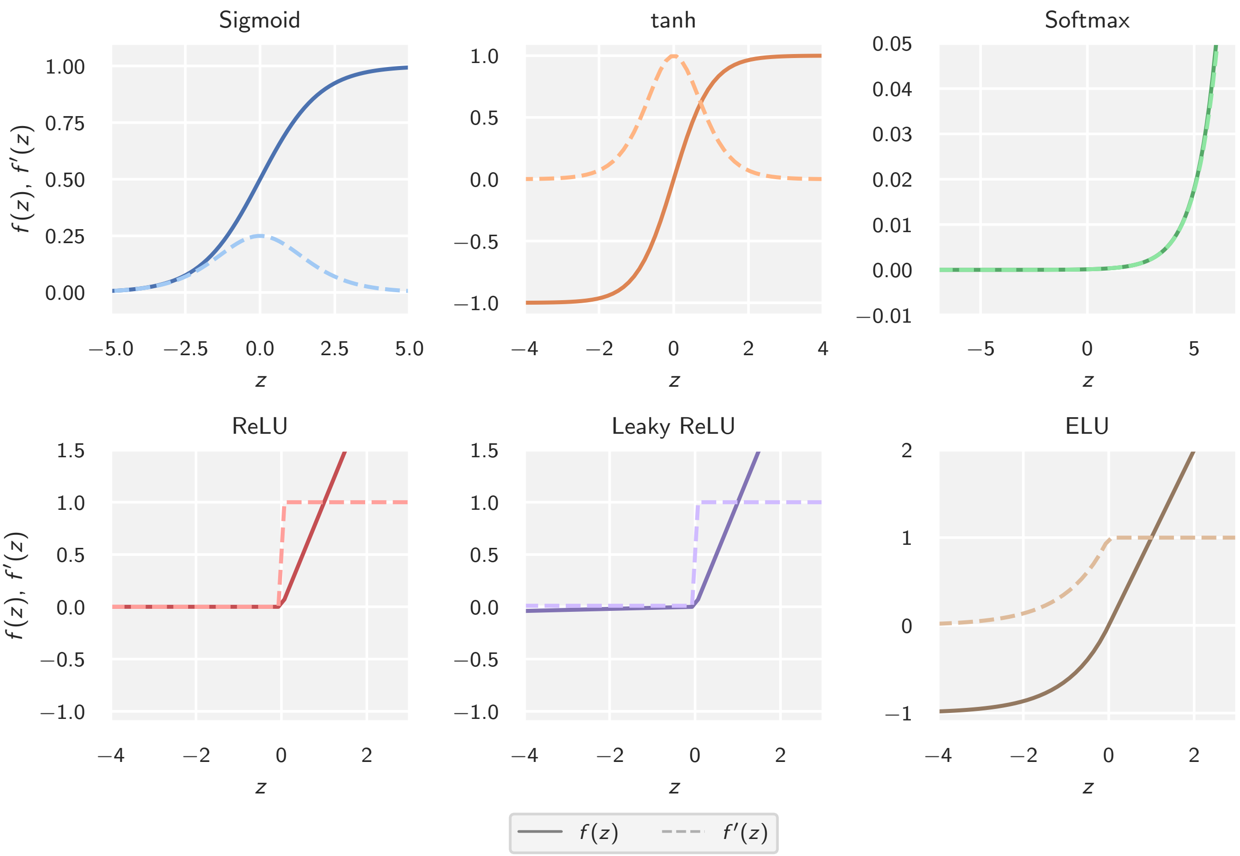

Figure 4: Activation functions | A representation of each function.

code. In Figure 4 all these activation functions have been plotted, to compare their trends.

2.3.a Activation functions and numerical instabilities

Some of this functions have been found to cause numerical instabilities. During backpropagation, the gradients can get smaller and smaller as the algorithm progresses down to the first hidden layers. As a result, the gradient descent update leaves the gradients of the lower hidden layers basically unchanged, causing the algorithm to never converge to a consistent solution. This is known in literature as *vanishing gradients problem* (Hochreiter, 1991). The opposite problem can also occur if the gradients exponentially grow from the last layer to the lower ones; in this case, we talk about *exploding gradients* (Philipp et al., 2018). Recently, a study from Glorot and Bengio, 2010 stated that most of this numerical instabilities were caused by an improper random initialization of the weights at the beginning of backpropagation.

2.3.b Employed activation functions

Rectified Linear Unit (ReLU)

The ReLU activation is defined as:

$$\mathrm{ReLU}(z) = \max(0, z) \tag{19}$$

and its derivative reads

$$\mathrm{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{20}$$

ReLU avoids vanishing gradients for positive inputs and is computationally efficient (Nair & Hinton, 2010). However, it can suffer from ”dying ReLU” problems where neurons become inactive (Maas et al., 2013).

Leaky ReLU

The Leaky ReLU is defined as:

$$\mathrm{LeakyReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases} \tag{21}$$

and its derivative reads

$$\mathrm{LeakyReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ \alpha & \text{if } z \leq 0 \end{cases} \tag{22}$$

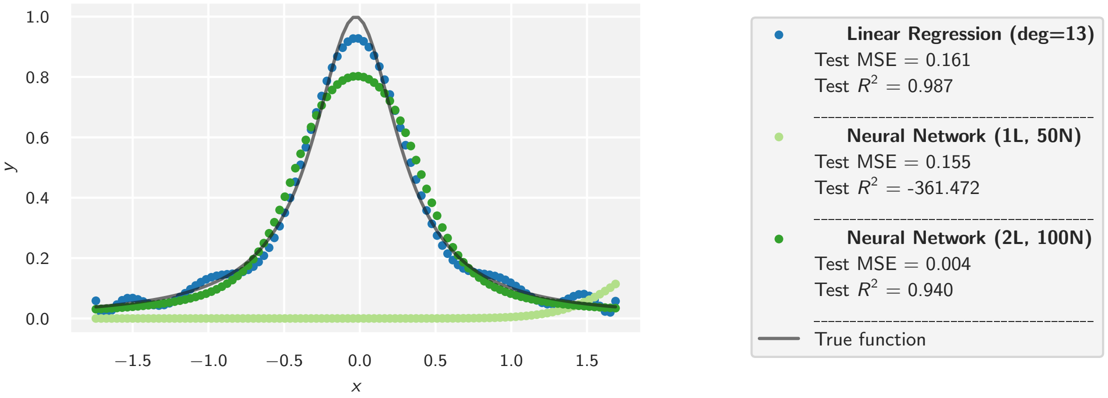

Figure 5: Linear OLS and Neural Networks | Three different models have been tested for a simple regression. A linear ordinary least squares model with degree 13 showed good test MSE and R^2. The NN with two hidden layers of 100 nodes performed even better with an MSE 50 times smaller than the previous one. The NN with one hidden layer only shows serious underfitting and it is not able to properly represent the function, with a strongly negative R^2.

where α is a small positive constant (typically 0.01). This activation prevents dead neurons by allowing a small gradient when $z \leq 0$, thanks to the modulation of parameter α (Maas et al., 2013).

Exponential Linear Unit (ELU)

The ELU activation is defined as:

$$\mathrm{ELU}(z) = \begin{cases} z & \text{if } z > \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases} \tag{23}$$

and its derivative reads

$$\mathrm{ELU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ \mathrm{ELU}(z) + \alpha & \text{if } z \leq 0 \end{cases} \tag{24}$$

where α is now typically set to 1.0. ELU smooths the transition around $z = 0$ and helps with vanishing gradients while maintaining negative values to push mean activations closer to zero (Clevert et al., 2015).

Logistic sigmoid

The S-shaped sigmoid (also known as *logit*) is given by:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{e^z}{1 + e^z} \tag{25}$$

and its derivative reads

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z)) \tag{26}$$

This activation function shrinks the signal to the range $(0, 1)$ (Mira & Sandoval, 1995). Historically, it has been used for binary classification. It mainly suffers from vanishing gradients for large values of $|z|$ (Glorot & Bengio, 2010).

Hyperbolic tangent

The hyperbolic tangent is given by:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{27}$$

and its derivative reads

$$\tanh'(z) = 1 - \tanh^2(z) \tag{28}$$

This activation function squashes the signal to the range $(-1, 1)$ (LeCun et al., 1998). It is zero-centered, which helps with gradient flow during training, but still suffers from vanishing gradients for large $|z|$ (Glorot & Bengio, 2010).

Softmax

The Softmax function for a vector $\boldsymbol{z} = (z_1, \ldots, z_K)$ is defined as:

$$\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{29}$$

and its derivative for the Jacobian matrix elements is:

$$\frac{\partial \mathrm{Softmax}(z_i)}{\partial z_j} = \mathrm{Softmax}(z_i)(\delta_{ij} - \mathrm{Softmax}(z_j)) \tag{30}$$

where δ_{ij} is the Kronecker delta. Softmax converts the classical logit to a probability distribution and is primarily used in the output layer for multi-class classification (Bridle, 1990), as in our case, namely.

2.4 Cost functions

For this work, two different types of cost functions have been employed, depending on the type of model (simple regression or classification).

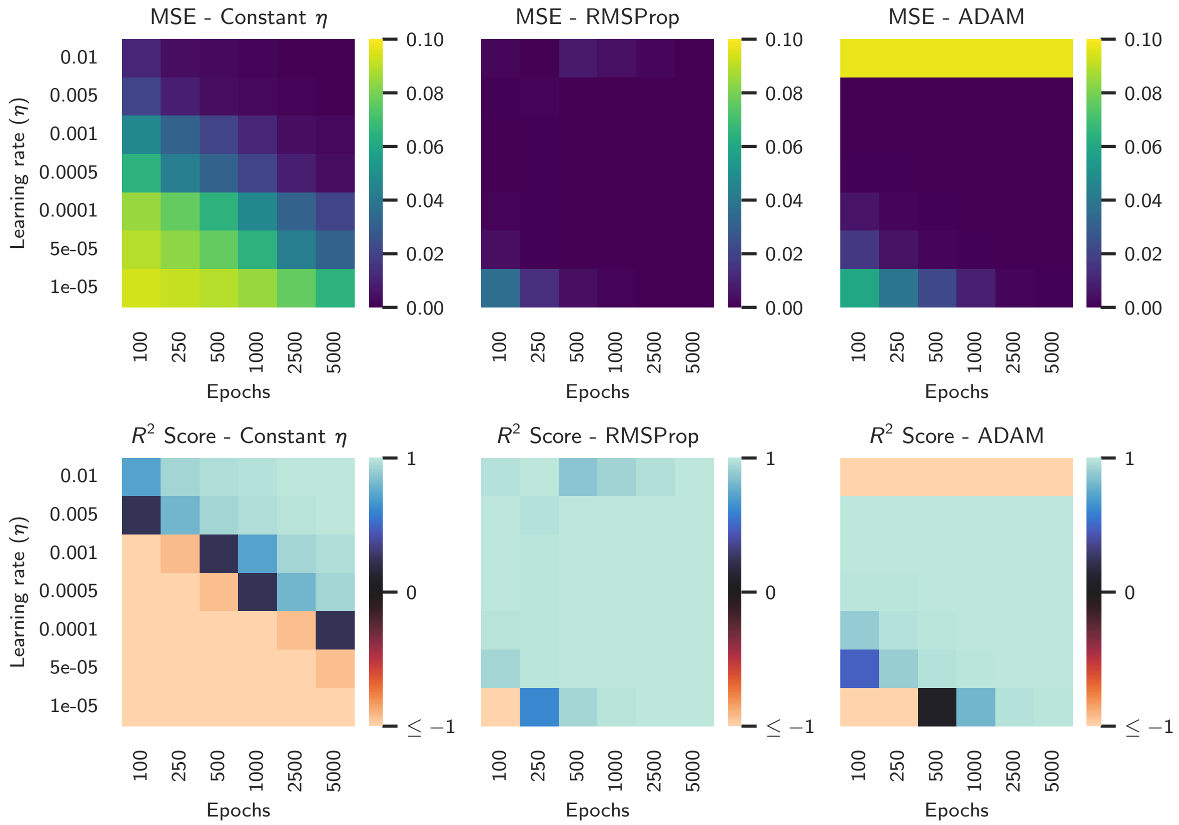

Figure 6: Performance of different SGD solvers | RMSProp and ADAM converge to acceptable results with fewer epochs, showing robust performance across learning rates and iteration counts. In contrast, plain SGD performs better with lower rates and higher numbers of epochs.

Mean Squared Error

The Mean Squared Error (MSE) loss function for regression tasks is defined as:

$$\text{MSE}(\boldsymbol{y}, \tilde{\boldsymbol{y}}, \boldsymbol{\Theta}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{31}$$

with derivative:

$$\frac{\partial \text{MSE}}{\partial \hat{y}_i} = \frac{2}{n}(y_i - \hat{y}_i) \tag{32}$$

MSE is general convex and has smooth derivatives, therefore is usually a standard choice for machine learning problems that include gradient descent (Hastie et al., 2009).

Cross-Entropy

Cross-Entropy class functions are use for classification-oriented NNs, as they express the cost in term of a probability of belonging to a given class (Bishop, 2006). The simplest case is given by the Binary Cross-Entropy (BCE):

$$\text{BCE}(\boldsymbol{y}, \tilde{\boldsymbol{y}}, \boldsymbol{\Theta}) = \\ = -\frac{1}{N}\sum_{i=1}^{N}[y_i \, \log(p_i) + (1 - y_i) \, \log(1 - p_i)]$$

with derivative:

$$\frac{\partial \text{BCE}}{\partial p_i} = \frac{p_i - y_i}{p_i(1 - p_i)} \tag{33}$$

For multi-class models, the formulation is modified as follows:

$$\text{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij} + \epsilon) \tag{34}$$

with gradient:

$$\frac{\partial \text{CE}}{\partial \hat{y}_{ij}} = \frac{\hat{y}_{ij} - y_{ij}}{n} \tag{35}$$

where ϵ is a small constant for numerical stability. Here, K represents the total number of classes. In this case, furthermore, the output $\hat{y}_{ij}$ is *one-hot encoded*. This type of encoding sets each class lab as a binary vector with values 1 for the true-class and 0 for the others (Bishop, 2006; Goodfellow et al., 2016).

Regularization

The cost function can be regularized by applying a penalty factor, in order to perform a more generalized model. This can be done by using the Ridge and LASSO methods: the first adds an L2-type norm to the cost function, the second one an L1-type (Goodfellow et al., 2016). Applying this penalties to the MSE leads to a shrinkage of the regression towards a constant model. Further details can be found in Project 1 report.

2.5 Classification

The classification effectiveness can be investigated through a confusion matrix. This is a widely used tool for evaluating the performance of classification models. It shows the relationships between predicted and true class labels in a tabular form, from which it is possible to compute scores such as accuracy and Cohen's κ (Kohavi & Provost, 1998; Powers, 2011). Each cell in the matrix represents the number of instances belonging to a specific combination of predicted and actual classes.

The accuracy score is given by

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(t_i = y_i)}{n}, \tag{36}$$

where I is an indicator function, 1 if $t_i = y_i$ and 0 otherwise if we have a binary classification problem. Here t_i represents the target and y_i the outputs of our NN code and n is simply the number of targets t_i.

2.6 Set up of the code

2.6.a Implementation

For this project, a Python module has been created, containing all the necessary functions and classes for the analysis. The core is a Neural Network class, that can perform all the main algorithms (feedforward, backpropagation, training) and adapt to different types of input datasets and regression/classification frames.

Since we deal with learning machinery, Python libraries such as `scikit-learn` have also been employed, mainly for assessing the relative behavior of the own coded NN with respect to external modules. More insights on this module's functionality can be found here.

The first part of the code is meant to implement all the activation and cost functions. The computation of the derivatives was approached in two different ways

— analytical calculation;

— automatic differentiation with `autograd`.

After that, an abstract class `Scheduler` has been created, in order to manage all the three different gradient descent algorithms (plain SGD, RMSProp and ADAM) independently of NN implementation.

The structure of `NeuralNetwork` class is shown in Algorithm 4.

As said previously, it supports all the different types of cost and activation functions. Plus, an external function `set_activations` allows choosing between automatic and manual differentiation. The cost functions and their respective derivatives can also be declared (MSE or Cross Entropy); their derivative are handled in the `cost_der` function and it is again possible to choose between analytical solutions or `autograd`. If the chosen cost is the MSE, one can also decide to apply an L1 or L2 norm, by default set to 0.1.

Algorithm 4 Neural Network Class Structure

```
class NeuralNetwork

procedure __init__
end procedure
procedure _create_layers
end procedure
procedure cost
end procedure
procedure cost_der
end procedure
procedure _feedforward
end procedure
procedure _backpropagate
end procedure
procedure _define_scheduler
end procedure
procedure _update_weights
end procedure
procedure _train
end procedure
procedure _predict
end procedure
procedure _accuracy
end procedure
procedure _reset_weights
end procedure

```

The class also accept a flexible number of layers, each with a freely definable number of neurons.

2.6.b Testing

Testing the custom NN was carried out in a Python Jupyter Notebook. The analysis consisted in:

1. Loading data, preprocessing

A dataset based on Runge function was created, and the MNIST collection was imported from `scikit-learn`.

Both the datasets have been scaled; for the Runge dataset we employed `StandardScaler` from `scikit-learn.utils`. For the MNIST data, since we were dealing with pixel values in a gray-scale range [0, 255], we normalized all the pixels to the interval [0, 1], dividing by 255.

Both have then been split in train and test sets, with a test size of 0.2.

2. Comparing with linear OLS regression

The performance of the NN in a simple linear regression has been compared to the ordinary least squares code from Project 1. Two neural network objects have been created, one with one hidden layer of 50 nodes an the other with two hidden layers of 100 nodes. Only the sigmoid has been used as activation function.

3. Testing different Stochastic Gradient Descent machineries

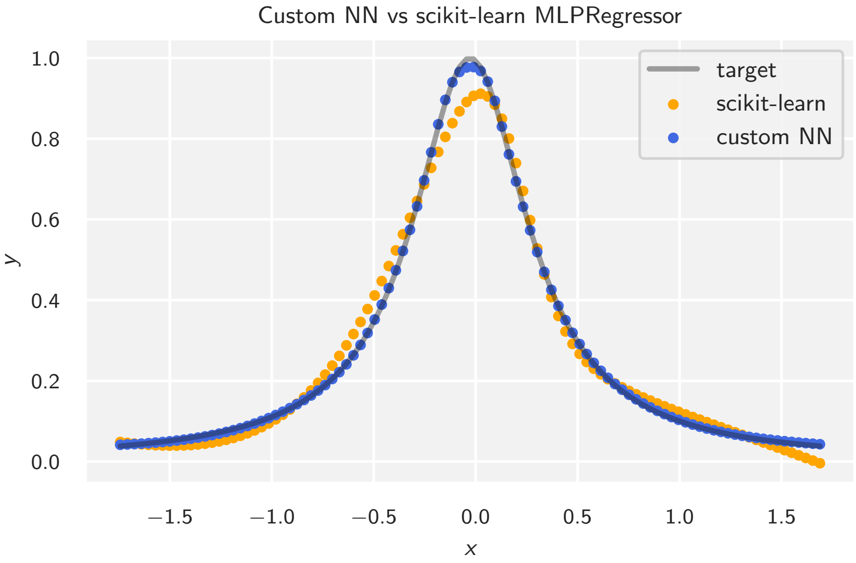

Figure 7: Custom NN vs `scikit-learn` | Both algorithms produce similar results, though our NN is 29 times slower.

Plain SGD, RMSProp and ADAM have been tested on a range of different learning rates $10^{-5} \leq \eta \leq 10^{-2}$ and for a number of epochs between 100 and 5000. This step was important in the understanding of which algorithm could perform better and faster among the three.

4. Comparing the own custom NN to `scikit-learn`, `autograd` and `torch` routines The performance of our NN has been compared to the two machine learning modules, in order to be able to assess the effectiveness of our algorithm relatively to well known algorithms. We also ensured that the gradients calculated with `autograd` were the same as the manual computed ones.

5. Applying L1 and L2 norms to MSE The effects of Ridge and LASSO regularizations have been explored for penalties $10^{-5} \leq \lambda \leq 1$ and for different learning rates.

6. Classification analysis with MNIST The NN has been employed for a multi-class classification task. Its performance has been assessed with an accuracy score and a confusion matrix.

3 Results

3.1 Comparison with linear OLS regression

In Figure 5, the results of a regression performed with both the `NeuralNetwork` and the `LinearRegression_own` classes are shown. The latter was set with a polynomial degree of 13, chosen according to the Bias-Variance tradeoff results from analyses carried out in Project 1.

In general, the best performance is achieved by the NN with two hidden layers, which shows the smallest deviation from the targets. Despite this, its R^2 is generally slightly lower than that of the linear OLS model.

Scheduler	Mean elapsed time (s)	$\pm\sigma_{\text{mean}}$ (s)
Constant η	1.673	0.025
RMSProp	2.379	0.020
ADAM	2.811	0.032

Table 1: Execution time for different SGD algorithms | ADAM and RMSProp are slower than plain SGD. A standard deviation of the mean is also shown.

3.2 Testing of different Stochastic Gradient Descent solvers

First, the speed of each algorithm was tested by repeating 50 training runs with a NN consisting of 2 hidden layers of 100 nodes each, all using a sigmoid activation. For all algorithms, an initial learning rate of 5×10^{-3} and 500 epochs were set. For each iteration, execution time was recorded, and the average elapsed time over the 50 iterations was calculated for each scheduler, as shown in Table 1. RMSProp was on average 1.4 times slower than plain SGD, and ADAM 1.7 times slower.

Second, the overall effectiveness of the three schedulers was investigated as a function of learning rate and number of epochs. The results, shown in Figure 6, indicate that RMSProp and ADAM, although slower, reach acceptable MSE and R^2 values even with relatively few epochs, *i.e.*, they converge faster to an optimal solution.

3.3 Custom NN and external libraries

The speed and performance of our NN were compared with `MLPRegressor` from `scikit-learn`. Us-

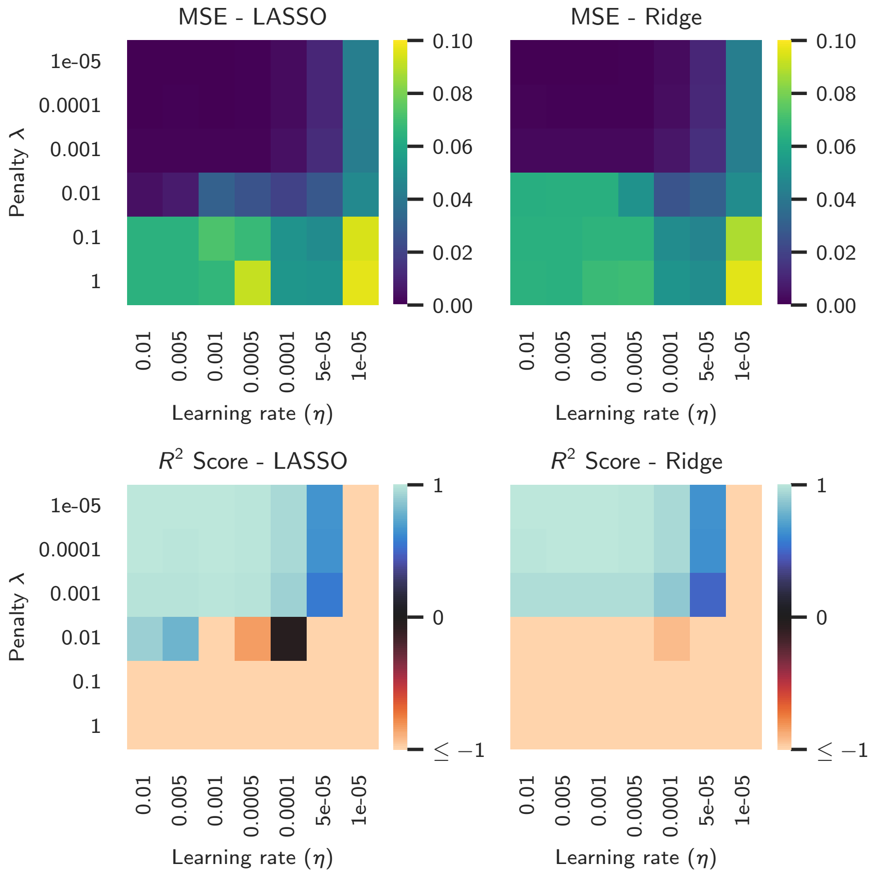

Figure 8: L1 and L2 regularization | Applying a penalty $\lambda \leq 10^{-3}$ can improve model generalization.

ing the same setup—two hidden layers with tanh activation, initial learning rate of 1×10^{-3}, and small L2 regularization—the execution time difference was significant. Both used ADAM as optimizer. Our NN completed the task in ~ 11.5 s, whereas `scikit-learn` completed it in $\sim$ 0.4 s, making it 29 times faster. The resulting fits are similar, although our NN produced a slightly better approximation on the test set ($R^2_{\text{own}} = 0.9995$, $R^2_{\text{sk-learn}} = 0.9932$). The fits are shown in Figure 7.

We also confirmed that gradients computed with `autograd` (by setting `autodiff` to True) matched the manually computed ones. Two NNs with identical setups but different differentiation methods yielded identical fits.

3.4 Adding L1 and L2 regularizations

The results of the regularized fits are shown in Figure 8. Acceptable results were obtained for $\lambda \leq 10^{-3}$ and $\eta \leq 5 \times 10^{-5}$. The quality of MSE and R^2 does not decay gradually; instead, there is a sharp transition at these thresholds.

3.5 Classification of MNIST data

MNIST-784 was correctly loaded, scaled, and split into train/test sets. Four sample images are shown in Figure 9.

Classification was performed using a NN with two hidden layers of 128 and 64 nodes, respectively. Leaky ReLU was used for hidden layers, and Softmax for the output layer. ADAM was set as optimizer with 1000 epochs and learning rate 1×10^{-3}, chosen according to previous results (Section 3.2).

Classification was successful, as shown in the confusion matrix in Figure 10. A high percentage of samples fall along the diagonal, indicating correct classification

The classification accuracy was 0.92, which is considered very good.

4 Discussion and conclusions

The custom NN performed overall as expected. It was able to correctly approximate simple functions and classify handwritten digits.

When it comes to simple regressions, the choice of a linear regressor such as Ordinary Least Squares (OLS) remains a solid option due to its relatively simple implementation and fast execution. The equations that define the model are usually straightforward and are characterized by a limited number of parameters (Wooditch et al., 2021). However, the main limitation of such methods is that they assume a linear relationship between the dependent and independent variables. For functions such as the

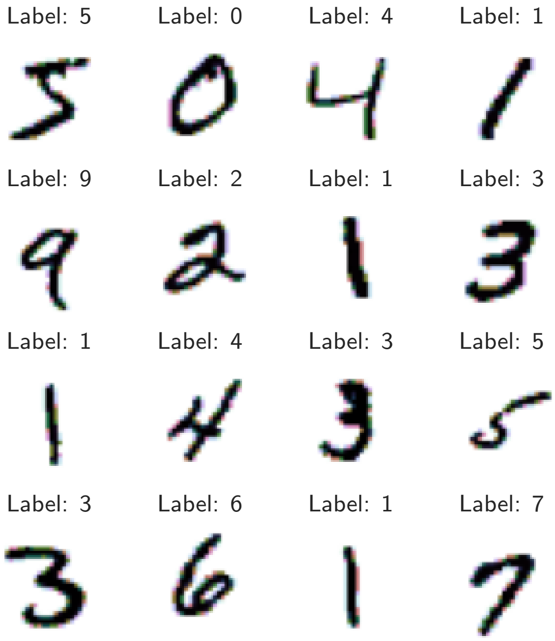

Figure 9: Samples from MNIST-784 | Handwritten digits are 28×28 pixels with gray scale values from 0 to 255.

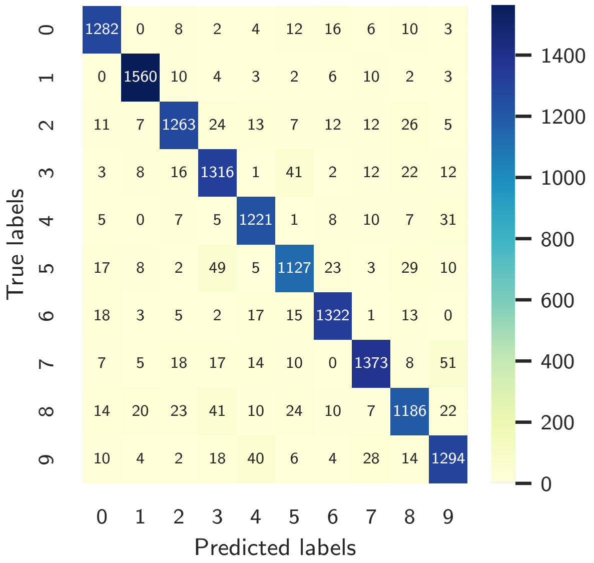

Figure 10: Confusion matrix of MNIST classification | Overall accuracy of 0.92 indicates good performance.

Runge one, this can represent a problem, since a polynomial approximation will not be able to properly capture the nonlinearities of the relationship. NNs, thanks to the nonlinear activation functions they include, are potentially a better choice for this type of system (de Ridder et al., 1999; Waterworth & Lees, 2000). This can be directly observed in Figure 5: even a complex polynomial OLS model cannot perform as well as a relatively simple two-hidden-layer NN, which correctly identifies the nonlinearities in the given function. Moreover, modern training techniques such as stochastic gradient descent and regularization strategies mitigate overfitting while maintaining model effectiveness (LeCun et al., 1998).

Choosing an NN over a linear regressor can still have drawbacks: it can take some time to find the best trade-off between complexity and efficiency, and very deep networks require a significant amount of computation time. In our case, the OLS regression fitted the Runge dataset in 0.0011 s, whereas the two-hidden-layer NN with 100 nodes and 1000 epochs required 4.58 s, which is about 4000 times slower.

Nevertheless, our results showed that highly efficient algorithms from external libraries are able to tackle the same tasks within a much smaller time interval, and this highlights one of the main weaknesses of our custom code: despite the overall consistent results, the code is still far from being considered fine-tuned.

For classification purposes, NNs are the only reasonable option, as linear regression assumes a continuous output space and does not constrain predictions to lie within class probability bounds (e.g., between 0 and 1). As a result, linear regression can yield invalid probability estimates and poorly defined decision boundaries (Bishop, 2006; Hosmer et al., 2013). In our case, *i.e.* a ten-class classification, NNs were able to correctly identify all patterns in the digits, resulting in high accuracy.

We can finally discuss the performance of our NN with regard to the choice of complexity, parameters, and optimization mechanisms.

In the context of this analysis, shallow networks with two or at most three layers and no more than 150 nodes per layer have proven to be a solid choice for tasks such as simple nonlinear regressions and the classification of very small images (on the order of 10–10^2 squared pixels), even with a relatively large input dataset (70 000 for MNIST-784). Nevertheless, choosing a single hidden layer network can easily lead to excessive simplification and yield unacceptable results, as shown in Figure 5.

The choice of activation functions is also crucial. We carried out several experiments to determine which functions could handle different problems most effectively. The sigmoid and tanh functions appear to offer a relatively good trade-off for regressions, particularly thanks to their nonlinearity. In contrast, the LeakyReLU and ReLU were not able to properly represent the curvature of the Runge function, resulting in an oversimplified approximation characterized by sharp corners. However, the ReLU family showed optimal results for discrete systems, such as classification. This can be explained by the fact that ReLU does not saturate for positive values and is also easier to compute (Glorot & Bengio, 2010).

Regarding optimization, plain SGD with a constant learning rate is a good option if the priority is a simple implementation coupled with fast execution for each epoch. However, adaptive learning rates represent a more robust choice that can yield better results almost *independently* of the number of epochs and the initial learning rate value, according to our findings (Figure 6). In fact, ADAM and RMSProp showed faster convergence, returning optimal weights and biases after only a relatively small number of epochs (250). Furthermore, these methods are generally more numerically stable, as they can produce acceptable results even for higher learning rates ($\eta \sim 10^{-3}$), overcoming the typical fixed-learning-rate divergence issues. These behaviors are also widely recognized in previous studies (Goodfellow et al., 2016; Kingma & Ba, 2017; Tieleman & Hinton, 2012).

The application of an L1 or L2 norm can also be considered and can certainly improve the model, particularly where a non-regularized network would fail or when working with a very small input dataset. However, our findings suggest avoiding penalties $\lambda \geq 0.01$, as this can lead to serious underfitting and loss of useful complexity. This is also confirmed in existing literature (Ng, 2004).

To conclude, we can state that the objectives of our analysis have been largely achieved. A general understanding of which hyperparameters, levels of complexity, and activation functions should be chosen depending on the system type is now provided. To evaluate more consistently how to tune such parameters, cross-validation and bias–variance trade-off analyses could be carried out, as previously done in Project 1. Finally, we conclude this work with a fairly low-level and non-optimized NN. Higher efficiency could certainly be reached by taking inspiration from well-performing modules such as `pytorch` and `tensorflow` .

References

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Springer-Verlag.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. *Neurocomputing: Algorithms, Architectures and Applications*, 227–236.

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint, arXiv:1511.07289*.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

de Ridder, D., Duin, R., Verbeek, P., & van Vliet, L. (1999). The Applicability of Neural Networks to Non-linear Image Processing. *Pattern Analysis & Applications*, *2*(2), 111–128. https://doi.org/10.1007/s100440050022

Glorot, X., & Bengio, Y. (2010–May 15). Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh & M. Titterington (Eds.), *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256, Vol. 9). PMLR. https://proceedings.mlr.press/v9/glorot10a.html

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press. https://www.deeplearningbook.org/contents/optimization.html

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer. https://doi.org/10.1007/978-0-387-84858-7

Haykin, S. (1994). *Neural networks: A comprehensive foundation* (1st ed.). Prentice Hall PTR.

Hochreiter, S. (1991). *Untersuchungen zu dynamischen neuronalen Netzen* [Diplom thesis]. Institut für Informatik, Technische Universität München. Retrieved on January 1, 2024, from https://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, *4*(2), 251–257.

Hosmer, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). *Applied logistic regression*. John Wiley & Sons.

Kingma, D. P., & Ba, J. (2017). ADAM: A method for stochastic optimization. https://arxiv.org/abs/1412.6980

Kohavi, R., & Provost, F. (1998). Confusion matrix. In *Encyclopedia of machine learning*. Springer.

LeCun, Y., Bottou, L., Orr, G. B., & Müller, K.-R. (1998). Efficient BackProp. *Neural Networks: Tricks of the Trade*, 9–48.

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. *Proceedings of the 30th International Conference on Machine Learning*, *28*.

Mira, J., & Sandoval, F. (Eds.). (1995). *From natural to artificial neural computation* (Vol. 930). Springer. Retrieved on January 1, 2024, from https://archive.org/details/fromnaturaltoart1995inte

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning*, 807–814.

Ng, A. Y. (2004). Feature selection, L1 vs. L2 regularization, and rotational invariance. *Proceedings of the Twenty-First International Conference on Machine Learning*, 78.

Philipp, G., Song, D., & Carbonell, J. G. (2018). The exploding gradient problem demystified - definition, prevalence, impact, origin, tradeoffs, and solutions. https://arxiv.org/abs/1712.05577

Powers, D. M. W. (2011). Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, *2*(1), 37–63.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

Tieleman, T., & Hinton, G. (2012). Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Wang, S.-C. (2003). Artificial neural network. In *Interdisciplinary computing in java programming* (pp. 81–100). Springer US. https://doi.org/10.1007/978-1-4615-0377-4_5

Waterworth, G., & Lees, M. (2000). Artificial Neural Networks in the Modelling and Control of Non-Linear Systems. *IFAC Workshop on Programmable Devices and Systems (PDS 2000), Ostrava, Czech Republic, 8-9 February 2000*, *33*(1), 95–97. https://doi.org/10.1016/S1474-6670(17)35594-5

Wooditch, A., Johnson, N. J., Solymosi, R., Medina Ariza, J., & Langton, S. (2021). Ordinary least squares regression. In *A beginner's guide to statistics for criminology and criminal justice using R* (pp. 245–268). Springer International Publishing. https://doi.org/10.1007/978-3-030-50625-4_15