



UNIVERSITETET I OSLO

Project 2: Building a Neural Network code Applied Data Analysis and Machine Learning UiO (FYS-STK4155)

Pietro PERRONE

pietrope@uio.no

November 10, 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/h2ifare1m1c31ud8vf>

Abstract

AA
AA
AA

Contents

1	Introduction	2
2	Theory and methods	2
2.1	The universal approximation theorem	2
2.2	Structure of a Neural Network	3
2.2.1	The Feed-forward algorithm	3
2.2.2	The Backpropagation algorithm	3
2.2.3	Training the Neural Network	5
2.3	Activation functions	5
2.3.1	Activation functions and numerical instabilities	5
2.3.2	Employed activation functions	6
2.4	Cost functions	7
2.5	Classification	8
2.6	Set up of the code	8
2.6.1	Implementation	8
2.6.2	Testing	8
3	Results	9
3.1	Comparison with linear OLS regression	9
3.2	Testing of different Stochastic Gradient Descent solvers	9
3.3	Custom NN and external libraries	11
3.4	Adding L1 and L2 regularizations	11
3.5	Classification of MNIST data	11
4	Discussion	11
5	Conclusion	13
	References	13

1 Introduction

An Artificial Neural Network is a computational model that emulates the functioning of human brain, in the way it can process several informations 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 node. Nodes can be regrouped in 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 *activation function*, which can be specified for each layer. Formally, a 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 (Goodfellow et al., 2016). The weights are the parameters used in a 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 these 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, particularly, we focus on so called *multilayer perceptrons* (MLPs). An other 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 correspond to a *fully connected* FFNN (Haykin, 1994). The operation by which the weights and biases are optimized is called *back-propagation*, that consists of calculating the gradients of the cost function with respect to weights and biases starting from the last layer; a training can then be performed in order to minimize this 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 a NN code in Python, that could perform both forward feeding and backpropagation. We will test this model initially on a simple 1-dimensional dataset, given by a cluster of samples computed with the Runge function for an input array $\mathbf{x} \in [-1, 1]$. Several types of activation functions and

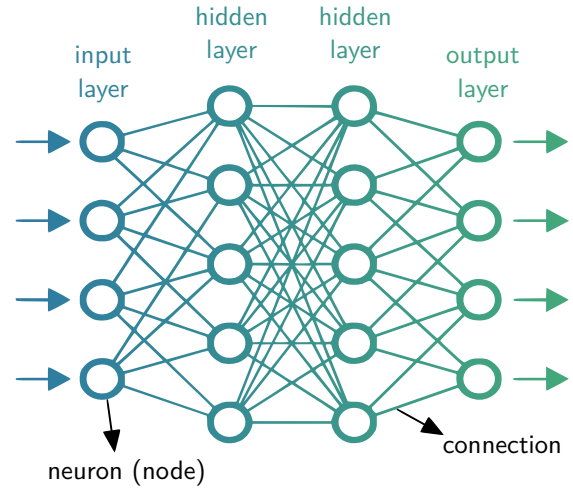


Figure 1: Neural Network | A typical structure.

layers depths will be tested.

A more complicated fit will be then performed on a dataset of handwritten digits, the *MNIST-784*. This consists of 70 000 digits samples, each of size 28×28 pixels. The objective will be to train the coded NN to make it recognize and classify correctly these digits. 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 the Neural Networks 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:

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

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

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

In other words, any continuous function $y = F(\mathbf{x})$ 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 $\mathbb{E}[|F(\mathbf{x})|^2]$:

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

it was concluded that

$$\begin{aligned} \mathbb{E}[|F(\mathbf{x}) - f(\mathbf{x}; \Theta)|^2] \\ = \int_{\mathbf{x} \in D} |F(\mathbf{x}) - f(\mathbf{x}; \Theta)|^2 p(\mathbf{x}) d\mathbf{x} < \epsilon. \end{aligned} \quad (2)$$

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:

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

$$\Theta = (\mathbf{W}, \mathbf{b}) \quad (3)$$

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

$$\mathbf{z} = \sum_{j=1}^m \sum_{i=1}^n w_{ij} x_i \quad (4)$$

$$\tilde{\mathbf{y}} = f(\mathbf{z}) \quad (5)$$

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 $f(\mathbf{z})$ directly corresponds to the output of the NN, which will be passed to the chosen cost function $\mathcal{C}(\Theta)$. This simple architecture is shown in Figure 2 (Goodfellow et al., 2016).

Generalization to a multi-layer network with multiple inputs

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

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

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

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

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

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

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

Here, we have

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

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

Implementation

A possible pseudocode for such algorithm could be:

Algorithm 1 Feedforward

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

2.2.b The Backpropagation algorithm

A consistent result from the NN calculations can only be achieved with properly trained parameters \mathbf{W} and \mathbf{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 same. *Backpropagation* is the operation of calculating these gradients. For the layer l :

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

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

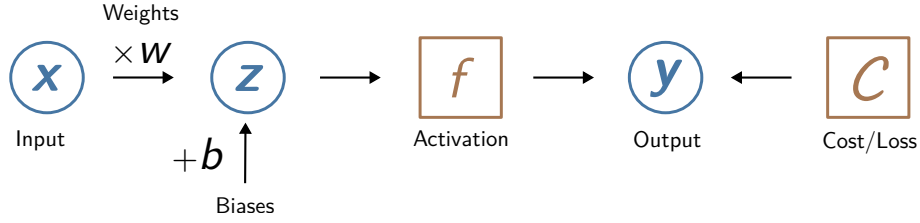


Figure 2: Simple perceptron model | In blue the different layers, in orange the functions.

the NN:

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

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. $\mathbf{A}^{(l-1)}$. In turn, $\mathbf{A}^{(l-1)}$ is given by the activation of $\mathbf{Z}^{(l-1)}$, which is a function of $\mathbf{A}^{(l-2)}$, and so on. The dependence of each layer from 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 broadcasting to all the hidden layers in reversed order (*backpropagation*, namely).

Formalism of backpropagation

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

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

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

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

The two derivatives respectively correspond to:

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

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

The overall derivative, then, reads:

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

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

node:

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

and, using vector notation again, we have:

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

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:

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

For the biases, the expression can be similarly derived:

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

since the only difference from the derivative with respect to the weights is the term $\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}}$, 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:

$$\begin{aligned} \boldsymbol{\delta}^{(l)} &= \left(f^{(l)} \right)' \odot \left(\mathbf{W}^{(l+1)} \right)^T \cdot \left(f^{(l+1)} \right)' \odot \dots \\ &\dots \odot \left(\mathbf{W}^{(L-1)} \right)^T \cdot \left(f^{(L-1)} \right)' \odot \left(\mathbf{W}^{(L)} \right) \cdot \left(f^{(L)} \right)' \odot \nabla \mathcal{C}(\mathbf{A}^{(L)}) \end{aligned}$$

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 layers $(l+1)$. Recalling equation (6), we can finally derive a general expression for $\delta^{(l)}$:

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

and in vector notation:

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

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

$$\nabla \mathcal{C}(\Theta^{(l)}) = \begin{bmatrix} \frac{\partial \mathcal{C}}{\partial \mathbf{W}^{(l)}} \\ \frac{\partial \mathcal{C}}{\partial \mathbf{b}^{(l)}} \end{bmatrix} = \begin{bmatrix} \delta^{(l)} \mathbf{A}^{(l-1)} \\ \delta^{(l)} \end{bmatrix}. \quad (18)$$

Implementation

Note that, in order to run backpropagation, the feed-forward algorithm A possible pseudocode for the backpropagation algorithm could be:

Algorithm 2 Backpropagation

Require: Feedforward activations $\{\mathbf{A}^{(l)}\}_{l=0}^L$, pre-activations $\{\mathbf{Z}^{(l)}\}_{l=1}^L$, weights $\{\mathbf{W}^{(l)}\}_{l=1}^L$, cost \mathcal{C}

Ensure: Gradients $\nabla \mathcal{C}(\Theta^{(l)})$

```

1: Initialize  $\delta^{(l)} = 0$  for all  $l$ 
2:  $L \leftarrow$  number of layers
3:  $\triangleright$  Compute output layer error
4:  $\delta^{(L)} \leftarrow \frac{\partial \mathcal{C}}{\partial \mathbf{A}^{(L)}} \odot f^{(L)'}(\mathbf{Z}^{(L)})$ 
5: for  $l = L - 1$  down to 1 do
6:    $\delta^{(l)} \leftarrow (\delta^{(l+1)} \mathbf{W}^{(l+1)})^\top \odot f^{(l)'}(\mathbf{Z}^{(l)})$ 
7: end for
8:  $\triangleright$  Compute gradients for weights and biases
9: for  $l = 1$  to  $L$  do
10:   $\frac{\partial \mathcal{C}}{\partial \mathbf{W}^{(l)}} \leftarrow (\mathbf{A}^{(l-1)})^\top \delta^{(l)}$ 
11:   $\frac{\partial \mathcal{C}}{\partial \mathbf{b}^{(l)}} \leftarrow \sum_i \delta_i^{(l)}$ 
12: end for
13: return  $\left\{ \frac{\partial \mathcal{C}}{\partial \mathbf{W}^{(l)}}, \frac{\partial \mathcal{C}}{\partial \mathbf{b}^{(l)}} \right\}_{l=1}^L$ 

```

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 \mathbf{X}, \mathbf{y} , cost \mathcal{C} , learning rate η

Ensure: optimized parameters

```

1: for each epoch do
2:   shuffle indices for batches
3:   for each batch  $(\mathbf{x}_b, \mathbf{y}_b)$  do
4:     backpropagate  $\mathbf{g} \leftarrow \nabla \mathcal{C}(\Theta)$ 
5:     update parameters  $\Theta \leftarrow \Theta - \eta \mathbf{g}$ 
6:   end for
7: end for
8: return  $\Theta$ 

```

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 function can actually be employed. We will present five of these, that will be used in our code. In Figure 3 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 the 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

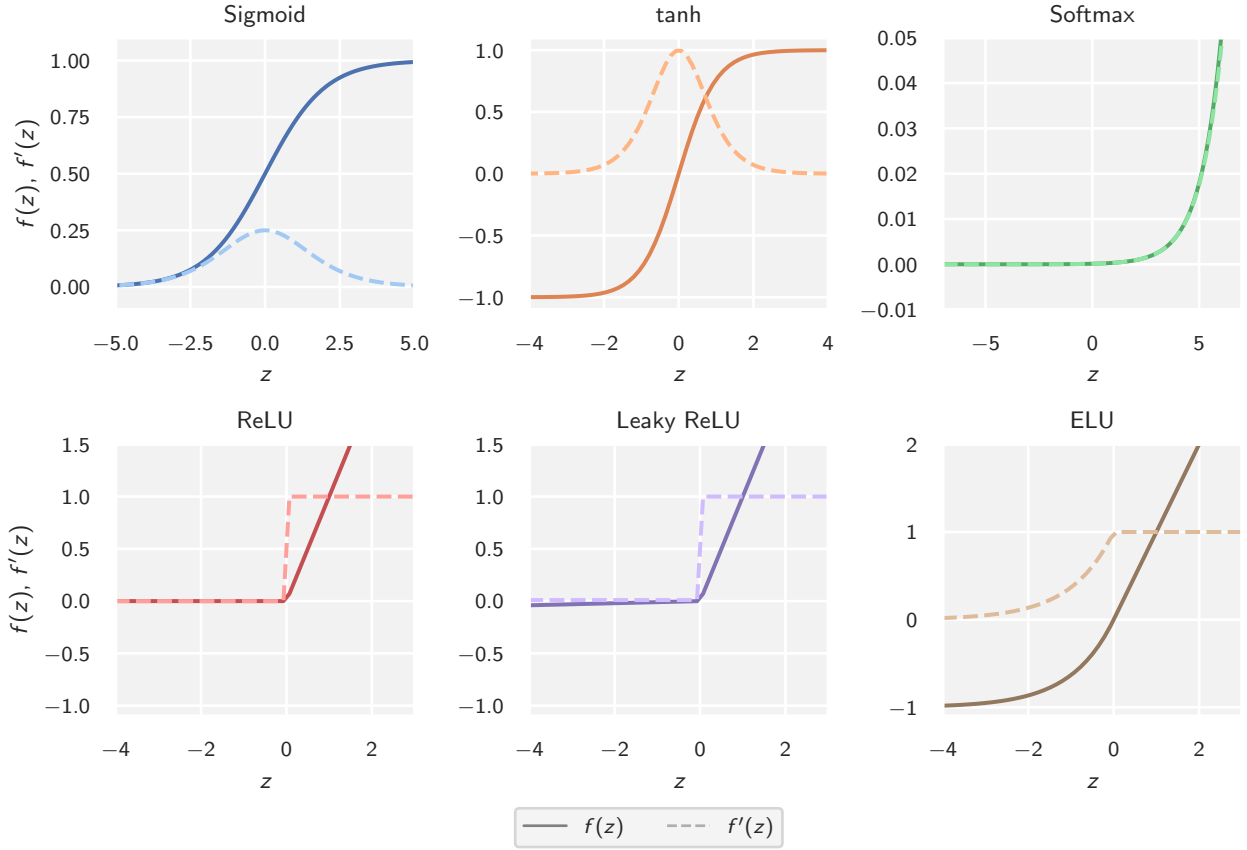


Figure 3: Activation functions | A representation of each function

and Bengio, 2010 stated that most of this numerical instabilities were caused by a improper random initialization of the weights at the beginning of backpropagation.

Leaky ReLU

The Leaky ReLU is defined as:

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

and its derivative reads

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

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).

2.3.b Employed activation functions

Rectified Linear Unit (ReLU)

The ReLU activation is defined as:

$$\text{ReLU}(z) = \max(0, z) \quad (19)$$

and its derivative reads

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

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).

Exponential Linear Unit (ELU)

The ELU activation is defined as:

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

and its derivative reads

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

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:

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

and its derivative reads

$$\sigma'(z) = \sigma(z) (1 - \sigma(z)) \quad (26)$$

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:

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

and its derivative reads

$$\tanh'(z) = 1 - \tanh^2(z) \quad (28)$$

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 $\mathbf{z} = (z_1, \dots, z_K)$ is defined as:

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

and its derivative for the Jacobian matrix elements is:

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

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).

Mean Squared Error

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

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

with derivative:

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

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

$$\text{BCE}(\mathbf{y}, \tilde{\mathbf{y}}, \Theta) =$$

$$= -\frac{1}{N} \sum_{i=1}^N [y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)]$$

with derivative:

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

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

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

with gradient:

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

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 label 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 applying 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

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

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 dataset and regression/classification frames.

Since we deal with learning machinery, Python libraries such as `scikit-learn` and `pytorch` have also been employed, mainly for assessing the relative behavior of the own coded NN with respect to external modules.

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 ?? As said previously, it supports all the differ-

Algorithm 4 Neural Network Class Structure

```

1: class NeuralNetwork
2:
3:   procedure __init__
4:   end procedure
5:   procedure _create_layers
6:   end procedure
7:   procedure cost
8:   end procedure
9:   procedure cost_der
10:  end procedure
11:  procedure _feedforward
12:  end procedure
13:  procedure _backpropagate
14:  end procedure
15:  procedure _define_scheduler
16:  end procedure
17:  procedure _update_weights
18:  end procedure
19:  procedure _train
20:  end procedure
21:  procedure _predict
22:  end procedure
23:  procedure _accuracy
24:  end procedure
25:  procedure _reset_weights
26:  end procedure
27:

```

ent 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.

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 pixel to the interval [0, 1], dividing by 255.

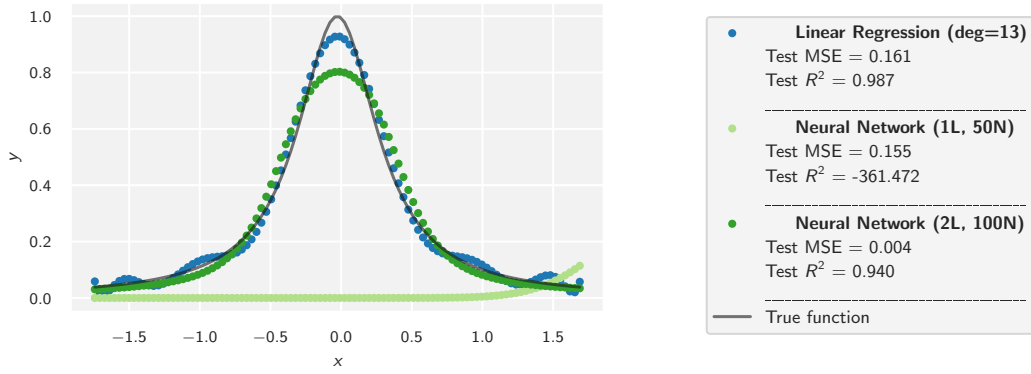


Figure 4: Linear OLS and Neural Networks | Three different models have been tested for a simple regression. A linear ordinary least squares model with degree 13 showed good test MSE and R^2 . The NN with two hidden layers of 100 nodes performed even better with an MSE 50 times smaller than the previous 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 .

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 and 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

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.

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: Different execution time for each SGD algorithm | ADAM and RMSProp are slower than the plain SGD. A standard deviation of the mean is also shown.

3.1 Comparison with linear OLS regression

In Figure 4, the result of a regression performed with both the `NeuralNetwork` and the `LinearRegression_own` classes. The second one has been set with a polynomial degree of 13; this choice was made in accord with the Bias-Variance tradeoff results from the analyses carried out in Project 1.

In general, we observe that the best performance is yielded by the NN with two hidden layers, that shows the smaller deviation from the targets. Despite this, its R^2 is generally fairly lower than the linear OLS one.

3.2 Testing of different Stochastic Gradient Descent solvers

First of all, the speed of each algorithm has been tested, by repeating for 50 iterations the same training with a NN of 2 hidden layers with 100 nodes each, all activated by a sigmoid. For all the algorithms, an initial learning rate of 5×10^{-3} and 500 epochs have been set. For each iteration, the execution time was registered. An average elapsed time over the 50 iterations was finally calculated for each scheduler, and the result can be seen in Table 1. RMSProp resulted in average 1.4 times slower than the plain SGD, and ADAM 1.7 times slower.

3 Results

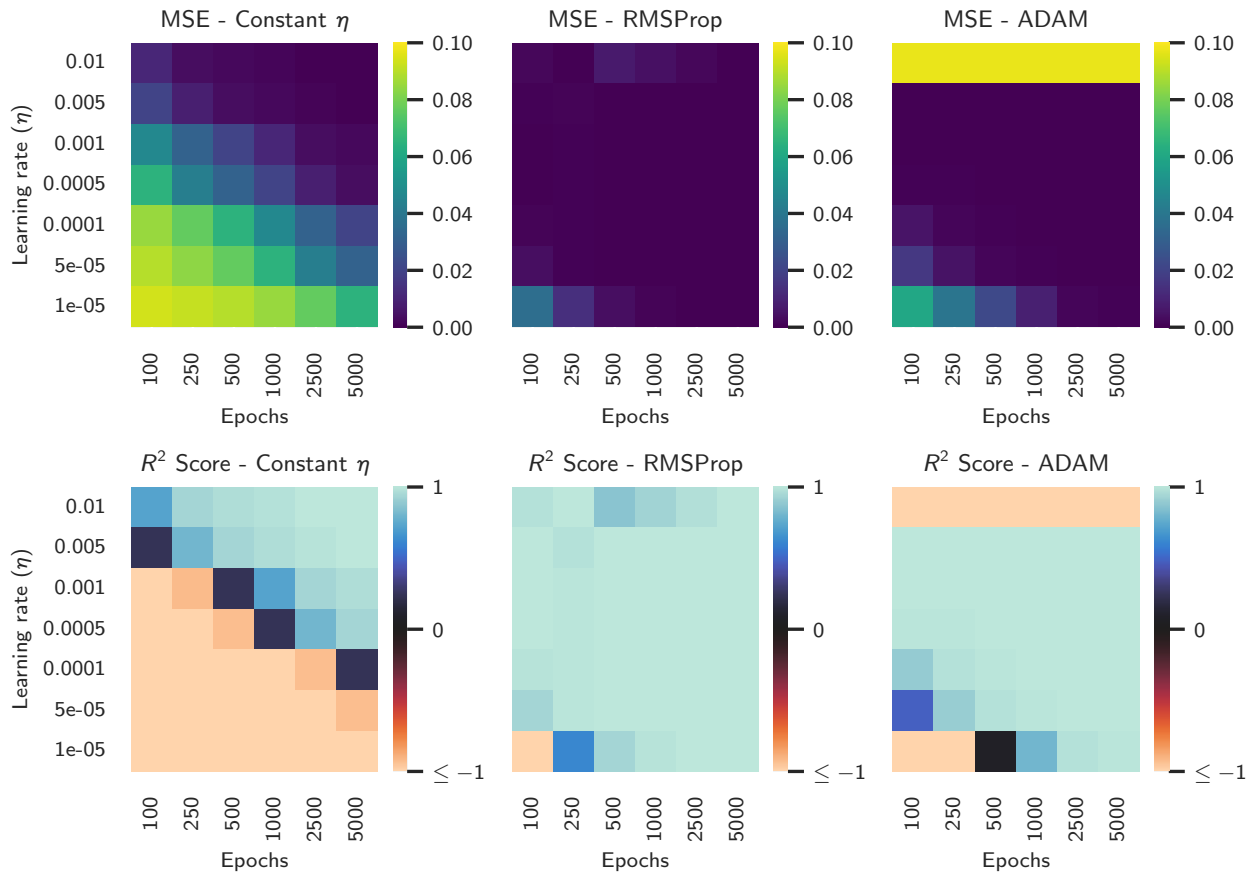


Figure 5: Performance of different SGD solvers | RMSProp and ADAM converge to acceptable results within a smaller amount of epochs, proving a fairly consistent behavior to any learning rate and number of iterations. On the opposite, plain SGD with constant learning rate seems performing better with lower rates and a higher number of epochs.

Secondly, the overall effectiveness of the three schedulers has been investigated as function of learning rate and number of epochs. The result of the analysis can be observed in Figure 5. Despite taking a longer time to be executed, RMSProp and ADAM show more than acceptable MSE and R^2 even with a relatively low number of epochs, *i.e.* they converge faster to an optimal value for the parameters.

3.3 Custom NN and external libraries

We compared the speed and effectiveness of our NN with the benchmark `MPLregressor` provided by `scikit-learn`. With the same characteristics, *i.e.* two hidden layers activated by a tanh, initial learning rate of 1×10^{-3} and small L2 regularizations, the difference in the execution time is huge. For both, ADAM has been used as optimizer. Our NN completed the task in ~ 11.5 s, whereas `scikit-learn` performed the same task in ~ 0.4 s, *i.e.* it is 29 times faster. The resulting fits are similar, even though our NN resulted in a slightly better approximation on the test set ($R_{\text{own}}^2 = 0.9995$, $R_{\text{sk-learn}}^2 = 0.9932$). The two fits are shown in Figure 6.

Moreover, we made sure to obtain the same gradients with `autograd`, setting the `autodiff` parameter to True, and with the manually computed ones. Two NNs with the same set up but with different differentiation methods yielded exactly the same fits.

3.4 Adding L1 and L2 regularizations

The results of the two regularized fits are shown in Figure 7. We can clearly see that acceptable results are obtained, in both cases, for $\lambda \leq 10^{-3}$ and for $\eta \leq 5 \times 10^{-5}$. We can observe that there is no gradual decay in the quality of MSE and R^2 : the transitions are very sharp at these two critical thresholds.

3.5 Classification of MNIST data

MNIST-784 have been correctly loaded, train-test split and scaled. Four samples from this collection can be observed in Figure ???. The classification of these data has been performed with NN of two hidden layers of size 128 and 64 nodes respectively. The Leaky ReLU has been chosen as activation function of these layers, whereas for the output a Softmax has been employed. This set up is known to yield the best results when it comes to multi-class classifications. ADAM has been set as optimizer, with 1000 epochs and learning rate of 10^{-3} , since this is one of the best combination according to previous results (Section 3.2).

The classification was successful: the result is shown in the confusion matrix of Figure 9. We can see that a high percentage of samples have been correctly classified, as

they fall on the diagonal of the matrix. The accuracy of the classification is of 0.92, which is more than acceptable.

4 Discussion

The custom NN performed overall as predicted. It has been able to correctly approximate simple functions and classify hand-written digits.

When it comes to simple regressions, the choice of a linear regressor such as the Ordinary Least Squares (OLS) is still a solid choice, for 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 the independent variable. For functions such as the Runge one, this can represent a problem, since a polynomial approximation will not be able to properly catch the non-linearities of the relationship. NNs, thanks to the non-linear activation functions that can include, are potentially a better choice for this type of systems (de Ridder et al., 1999; Waterworth & Lees, 2000). This can directly be observed in Figure 4: even a complex polynomial OLS model cannot perform well as a relatively simple, two-hidden layers NN, that exactly identifies non-linearities 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 a 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 will take a relevant amount of time to be executed. In our case, the OLS regression fitted the Runge dataset in 0.0011s, whereas the two-hidden layer NN with 100 nodes and 1000 epochs required 4.58s, that is around 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 raises one of the main criticalities of our custom code: despite the overall consistent results, the code is still far from being considered fine-tuned.

In the because it 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). Logistic regression and other probabilistic models are preferred alternatives, as they explicitly model the conditional class probabilities through non-linear link functions.

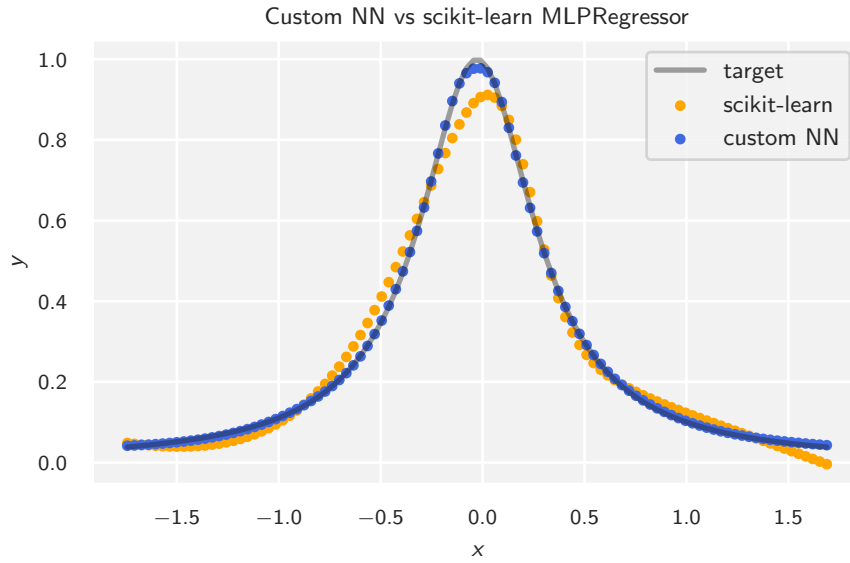


Figure 6: Custom NN vs `scikit-learn` | The two algorithms yield a similar result, even though our NN is 29 times slower.

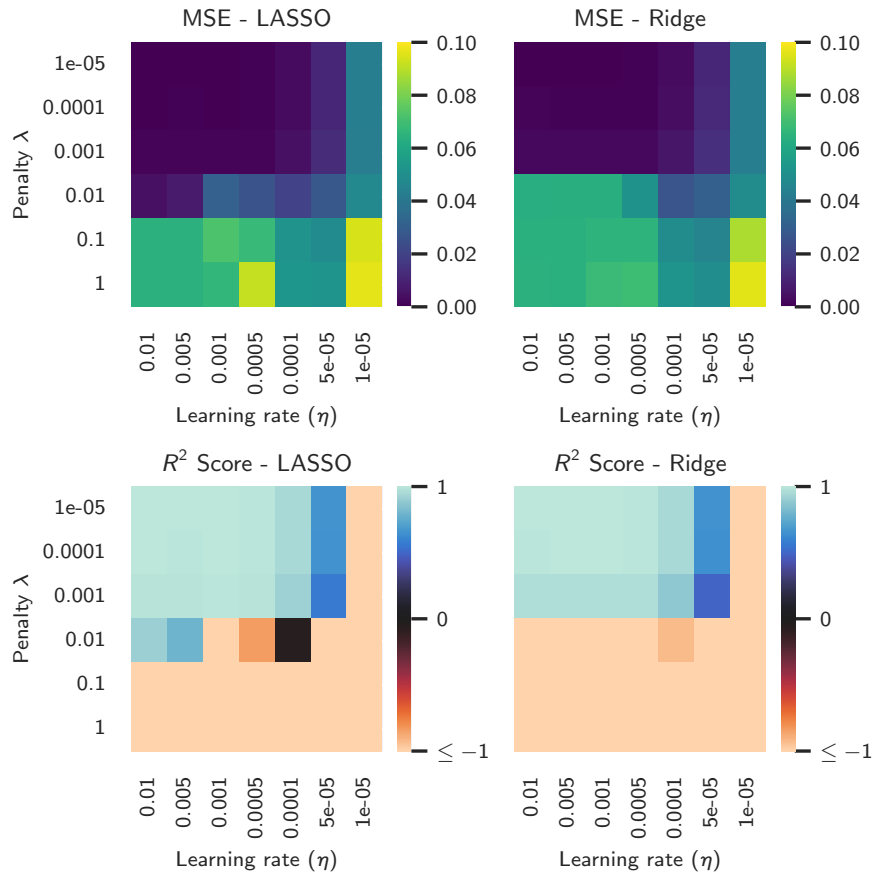


Figure 7: Custom NN vs `scikit-learn` | The two algorithms yield a similar result, even though our NN is 29 times slower.



Figure 8: Samples from MNIST-784 collection | The handwritten digits are 28×28 pixel sized and each pixel has gray scale value ranging from 0 to 255.

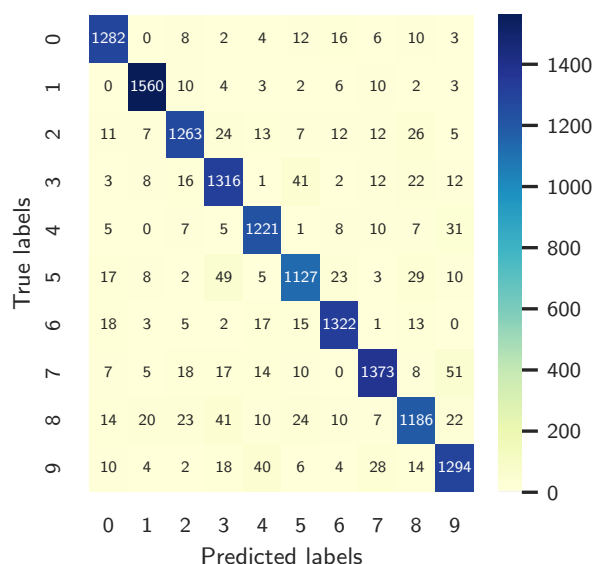


Figure 9: Confusion matrix of MNIST data classification | Good result, with overall accuracy of 0.92.

5 Conclusion

References

- Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Springer-Verlag.
- Bridle, J. S. (1990). Probabilistic interpretation of feed-forward 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. Titterton (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.
- 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](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