Students: Lorenzo Bianchi and Iacopo Bicchierini
Supervisor: Marco Cococcioni

July 7, 2022

GitHub: https://github.com/lorebianchi98/AutomaticDifferentation.

Contents

1 Introduction **2**

2 Dual Numbers **2**
2.1 Basic Operations 3
2.2 Differentiation 3
2.3 Absolute value 4

3 Dual2 and DualArray **5**
3.1 Dual2 5
3.2 DualArray 6

4 Benchmark in R^2 **6**

5 MLP with Dual Numbers **7**
5.1 MLP structure 8
5.1.1 MLP structure with 2 hidden layers 9
5.1.2 Generalization of the MLP network structure 10
5.2 Modification to use Dual Numbers 10

6 Conclusions **14**
6.1 Performance 14

7 Future Works **17**

1 Introduction

Differentiation is the process of finding the derivative of a function,this process is crucial in several fields of study like physics, mechanics and in recent years, with the advent of *Artificial Intelligence*, It becomes more crucial than ever also in Computer Science. It gains centrality due to the fact that in AI a lot of tasks need to be optimized and a lot of optimization techniques use derivatives.

There are 3 main methods to calculate derivative:

1. **Numerical Differentiation**: relies on the definition of derivative: you put a very small h and evaluate the function in two places. This is the most basic formula and on practice people use other formulas which give smaller estimation error This way of calculating a derivative is suitable mostly if you do not know your function and can only sample it. Also it requires a lot of computation for a high-dim function.

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

2. **Symbolic Differentiation** manipulates mathematical expressions. Systems that exploit this type of differentiation have for every math expression, the correspondent derivative formula and use various rules (product rule, chain rule) to calculate the resulting derivatives. It is common that the expression of the derivative becomes non manageable because of complexity.

$$\frac{d}{dx}\left(\frac{sin(x)}{1+x^2})\right) = \frac{(1+x^2)cos(x) - 2xsin(x)}{(1+x^2)^2}$$

3. **Automatic Differentiation** is a set of techniques to evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos..). We will use the technique that exploits the Dual Numbers.

2 Dual Numbers

In linear algebra, Dual Numbers are an extension of the Real Numbers introduced in the 19th century. They are expressions of the form a + bε, where a and b are real numbers, and ε is a symbol taken to satisfy $\varepsilon^2 = 0$ (nilpotent property), in a similar way as complex numbers adjoin i to satisfy $i = \sqrt{-1}$.

2.1 Basic Operations

Given the following two generic dual numbers z_1 and z_1:

$$z_1 = a_1 + b_2\varepsilon \qquad\qquad z_2 = a_2 + b_2\varepsilon$$

the arithmetic binary operations on them are defined as:

$$z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)\varepsilon$$

$$z_1 - z_2 = (a_1 - a_2) + (b_1 - b_2)\varepsilon$$

$$z_1 z_2 = (a_1 a_2) + (a_1 b_2 + a_2 b_1)\varepsilon + (b_1 b_2)\varepsilon^2 = (a_1 a_2) + (a_1 b_2 + a_2 b_1)\varepsilon$$

$$z_1 / z_2 = a_1/a_2 + ((a_2 b_1 - a_2 b_2)/a_2^2)\varepsilon \quad \text{(only defined when } a_2 \neq 0\text{)}$$

$$|z_1| = |a_1| + sign(a_1)\varepsilon$$

From the division between two dual numbers we can see that all duals without a real part are not invertible.

2.2 Differentiation

One application of dual numbers is automatic differentiation. Consider the real dual numbers above. Given any real polynomial $P(x) = p_0 + p_1 x + p_2 x^2 + ... + p_n x^n$, it is straightforward to extend the domain of this polynomial from the reals to the dual numbers. Then we have this result:

$P(a + b\varepsilon) = p_0 + p_1(a + b\varepsilon) + ... + p_n(a + b\varepsilon)^n$
$= p_0 + p_1 a + p_2 a^2 + ... + p_n a^n + p_1 b\varepsilon + p_2 ab\varepsilon + ... + p_n a^{n-1} b\varepsilon$
$= P(a) + bP'(a)\varepsilon$

where P' is the derivative of P. More generally, we can extend any (analytic) real function to the dual numbers by looking at its Taylor series:

$$f(a + b\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a) b^n \varepsilon^n}{n!} = f(a) + bf'(a)\varepsilon$$

since all terms of involving ε^2 or greater are trivially 0 by the definition of ε.
By computing compositions of these functions over the dual numbers and examining the coefficient of ε in the result we find we have automatically computed the derivative of the composition.

2.3 Absolute value

In the Basic Operations chapter we did not give a definition of absolute value for a dual number. This was done because it had to be made explicit the automatic differentiation property of a dual number to really understand this definition. The absolute value of $z = a + b\varepsilon$ is defined as:

$$|z| = |a| + sign(a)\varepsilon$$

This is coherent with the property of automatic differentiation of a dual number. In fact for a real number x the function $f(x) = |x|$ has for derivative $f'(x) = sign(x)$ as we can see from the following graphs:

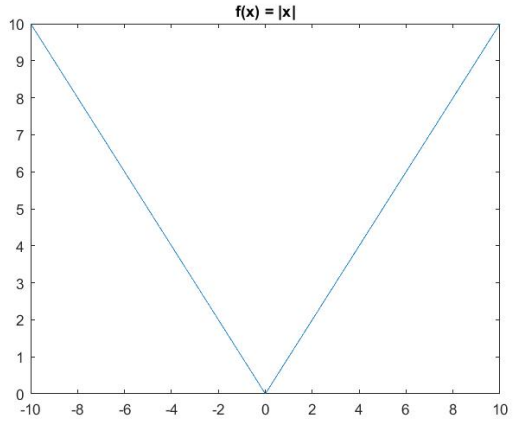


Figure 1: Absolute value of a real number **x**

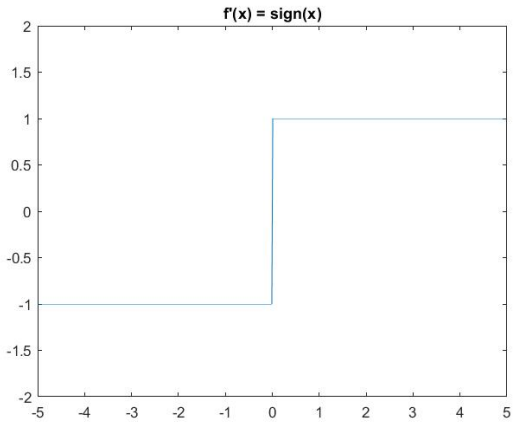


Figure 2: Derivative of the absolute value of x

This operation defined for a dual number will be crucial when we are gonna define a function to approximate the sigmoid in the following chapters.

3 Dual2 and DualArray

The idea was to create a new class that represents a single instance of Dual Number and then using it, create another class that represents an Array of Dual Numbers that implements all the operations useful to insert Dual Number inside the MLP implementation. This modular modeling is cleaner from the point of view of code and reasoning but will bring us problems from the point of view of performance as shown in the Performance paragraph.

3.1 Dual2

Firstly we developed the class that represents Dual Numbers in Matlab called Dual2, the class has real and dual values as properties. We defined the static method *setgetPrefs* which can set the modes in which to show the class Object, there are 3 modes related to the global view of the number:

- **ASCII**: (5.67, -12.34)
- **Intermediate**: 5.67 -12.34ϵ
- **Latex**: 5.67 -12.34\epsilon

and 3 modes of the format of the numerical parts:

- **'.4f'**:
- **'%+g'**:

- **'%+.4f'**:

The constructor is simple, it takes 2 values and set them into the real and dual parts of the object, if both or one lack the constructor put the correspondent value to 0. We implemented the binary arithmetic operations using the algebra shown before.

3.2 DualArray

The main constructor takes 2 matrices of the same size and put the values of the first matrix in the Real component of the dual element and the values of the second matrix in the Dual component. Even if we can construct a matrix of Dual Number, the operations work only with vectors, in the future works is suggested to extend the operations on the matrix ones.

All the arithmetic operations are element-wise, also the division and the multiplication. So in these operations, we simply perform the Dual2 operation for each correspondent component inside the values of the 2 vectors.

An important unary operation defined is the absolute value, since we use it inside the approximated Sigmoid. The idea is the make the absolute value of the real parts and take the sign of the real part as value of the dual component.

4 Benchmark in R^2

The first coding step is the developing of the Dual2 Matlab class. A Dual2 object contains a Dual Number and the redefinition of the 4 arithmetic operations. To test it we used a simple implementation of the Gradient Method, in which if the starting number is a Dual one, the method uses as derivative the Dual part of the Dual number, in the case of Real Number, it makes the derivative as usual.

Testing simple polynomial (since we redefined only the arithmetic operations) we observed as expected that the things go smoothly.

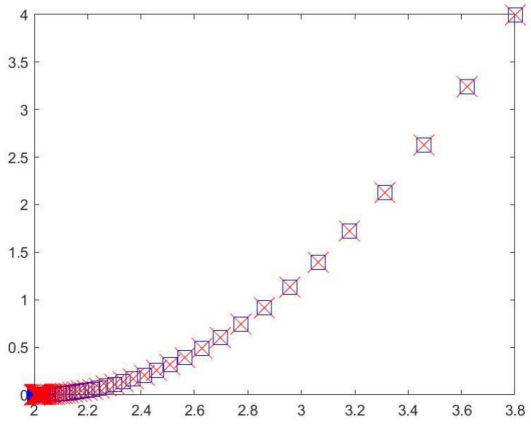


Figure 3: Comparison between the Dual and the Real Gradient Method

5 MLP with Dual Numbers

We decided to implement our version of an MLP with Dual Number starting from an existing implementation in MATLAB (see Appendix A). A MultiLayer Perceptron (MLP) is a fully connected class of feedforward artificial neural network.

5.1 MLP structure

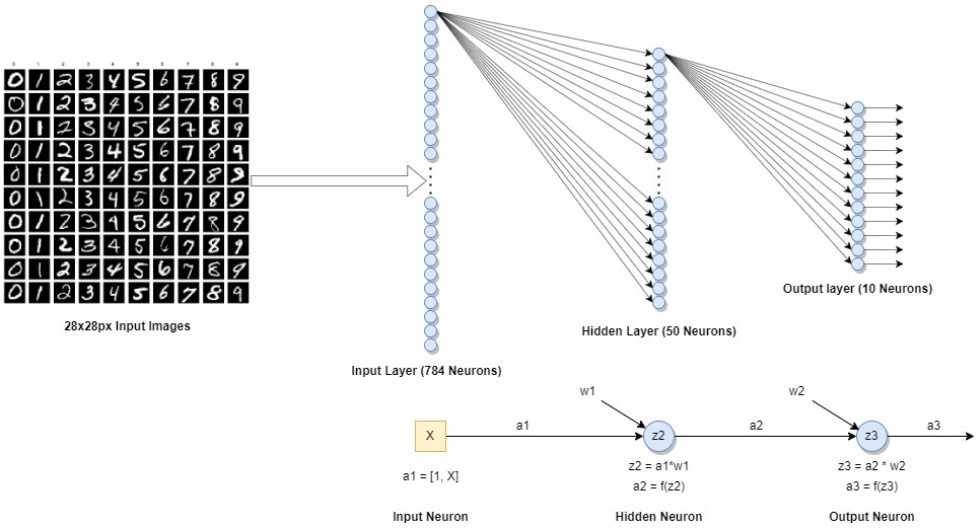


Figure 4: Notation and Structure of the MLP network

The network used for our task is a Multilayer Perceptron network with one hidden layer with 50 neurons.

We have **784 input neurons**, one for each pixel of the image(28x28). The value of an input neuron is the value in grayscale of a pixel of the image, and the vector **a1** (1x785) contains this values plus the bias. The Matrix **w1** (50x785) keeps the weights of the connections between input layer and hidden layer and **z2** = w1 * a1' (50x1) keeps the input of all the neurons in the hidden layer. Then the hidden layers neurons applies the approximated sigmoid function obtaining **a2** = approximated_sigmoid(z2) (50x1) and multiplying this vector for the weights **w2** (50x10) of the connections between hidden layer to output we obtain **z3** = w2 * a2 (10x1). The 10 output neurons apply the approximated sigmoid **a3** = approximated_sigmoid(z3) (10x1) the probability of the image to belong to each label.

The property of the Dual numbers will be useful during the backpropagation of the error in the hidden neurons: to update the weights w1 (connections between input layer and hidden layer) it is necessary the derivative of a2 = approximated_sigmoid(z2), and since when we apply a function to a dual number **x** we obtain also its derivative as **DualPart(x)**, we can store in a variable this value during the forward step and use it during the backpropagation avoiding an expensive calculation.

5.1.1 MLP structure with 2 hidden layers

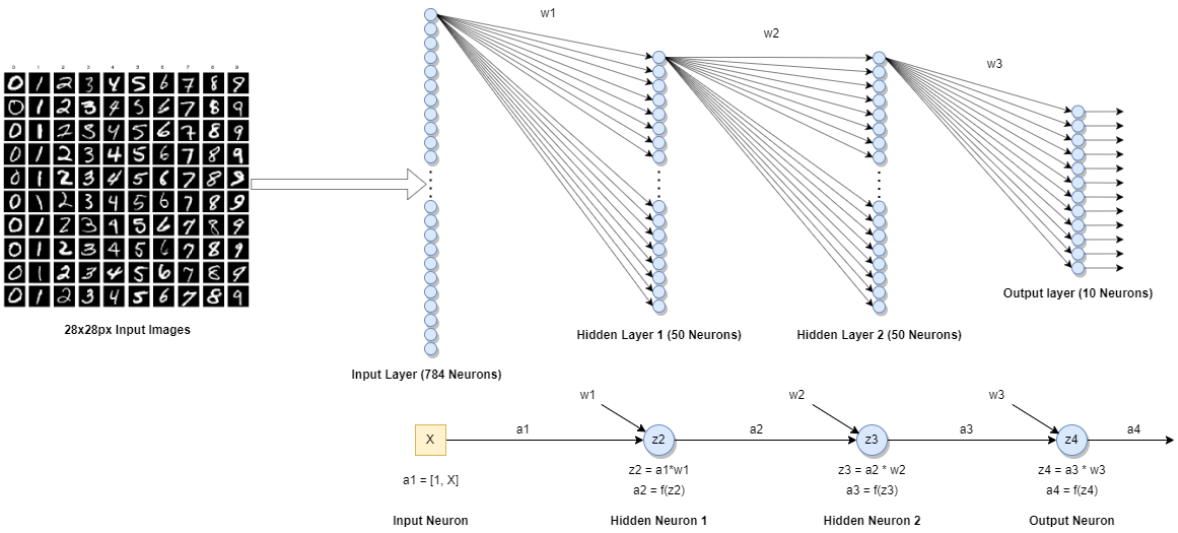


Figure 5: Notation and Structure of the MLP network with 2 hidden layers

We also implemented a MultylayerPerceptron Network with 2 hidden layers, both with 50 neurons.

This additional layer will lead to the following modification in the variables of the network: The input **a1** (1x785), the weights of the connections between input and hidden layer **w2** (50x785), the input of the hidden neurons **z2** = = w1 * a1' (50x1) and its activation **a2** = approximated_sigmoid(z2) (50x1) remain the same of the network with one hidden layer explained before. Then we have to compute the calculations for the second hidden layer: **z3** = w2 * a2 (50x1), where **w2** (50x51) is the matrix of the connections between the two hidden layers plus the bias, and **a3** = approximated_sigmoid(z3) is the application of the non linearity to the input of the hidden neurons. The calculation of the output neurons are the same: we obtain **z4** = w3 * a3 (10x1), where **w3** (50x10) are the weights of the connections between the second hidden layer and the output layer. The 10 output neurons apply the approximated sigmoid **a4** = approximated_sigmoid(z4) (10x1) the probability of the image to belong to each label.

The dual number property now will help us in avoiding the calculations in both the hidden layers.

5.1.2 Generalization of the MLP network structure

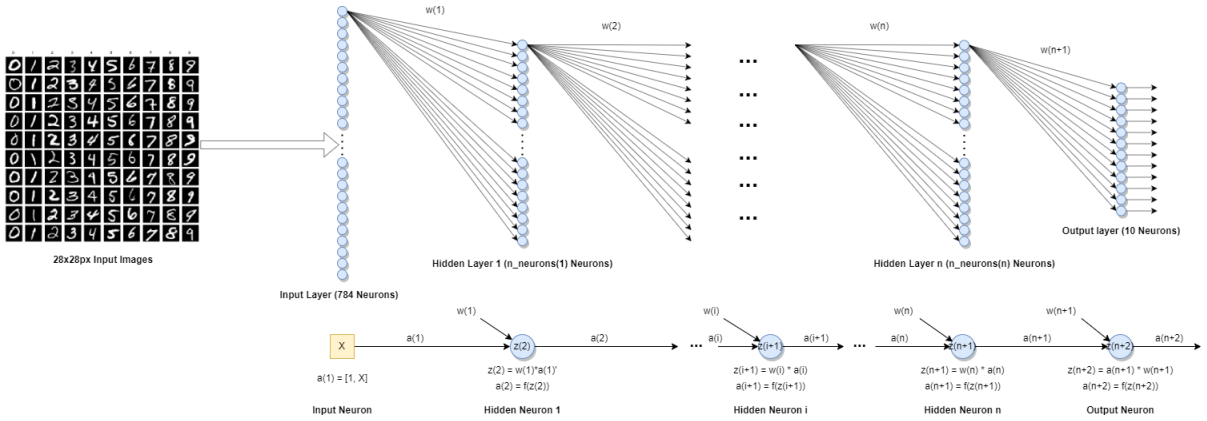


Figure 6: Notation and Structure of the MLP network with **n** hidden layers, with each layer **i** with **n_neurons(i)** neurons

From the two network analyzed we can retrieve a general form for a MLP network with **n** hidden layers and a number of neurons for the layer **i** stored in position i of a vector **n_neurons**.

As we can see in the figure we have that the variables of the activations **a**, of the weights **w** and of the input of neurons **z** becomes vector of matrix. In particular we have the following values for each variable:

a(1) = values of each pixel of the image + bias (1x785)
w(1) = weights of the connections input-hidden_1 layer(n_neurons(1)x785)

w(i+1) = weights of the connections hidden_i-hidden_1+1 layer(n_neurons(i+1) x n_neurons(i))
z(i+1) = w(i) * a(i) (n_neurons(i) x 1)
a(i+1) = f(z(i+1)) (n_neurons(i) x 1)

w(n+1) = weights of the connections hidden_n-output layer(10 x n_neurons(n))
z(n+2) = w(n+1) * a(n+1) * (10x1)
a(n+2) = f(z(n+2)) (10x1)

5.2 Modification to use Dual Numbers

The first modification is the usage of an approximation of the Sigmoid function, this is necessary in order to avoid to compute the exponential of a Dual Number that is not obvious to implement. Instead using this approximation we can do all the operations exploiting the arithmetic operations (+,-,*,/).

$$S(x) = \frac{1}{1 + e^{-x}}$$

$$S_\varepsilon(x) = \frac{1}{2}\left[\frac{x}{1 + |x|} + 1\right]$$

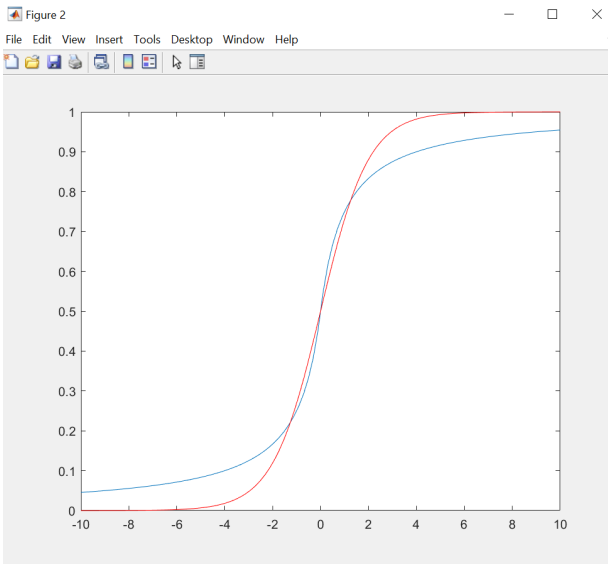


Figure 7: Comparison of sigmoids

The testing of this version of the Sigmoid with the MLP Network goes well, with results in term of Accuracy comparable with the original one.

It can be interesting to compare also the derivatives of these 2 sigmoids.

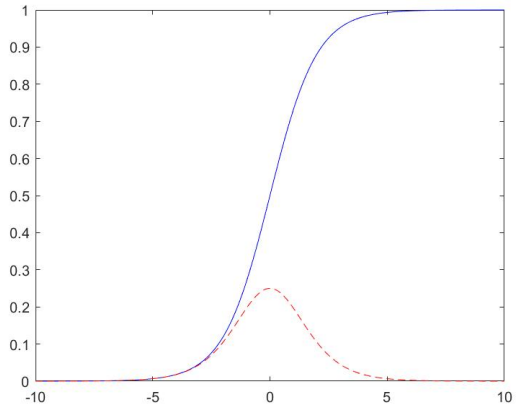


Figure 8: Original Sigmoid and its Derivative

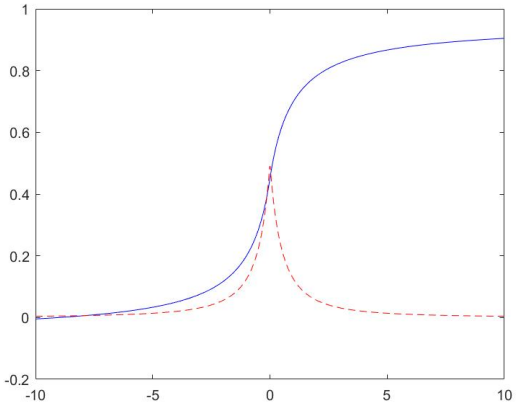


Figure 9: Approximated Sigmoid and its Derivative

The derivative of the approximated Sigmoid is very sharp in 0 due to the presence of the absolute value and the variance is smaller.

Furthermore we compared also the derivative of the approximated Sigmoid computed using the formula that we found making simple computations and the same value obtained evaluating the Sigmoid with the Dual Numbers, this was done to make sure that everything was fine.

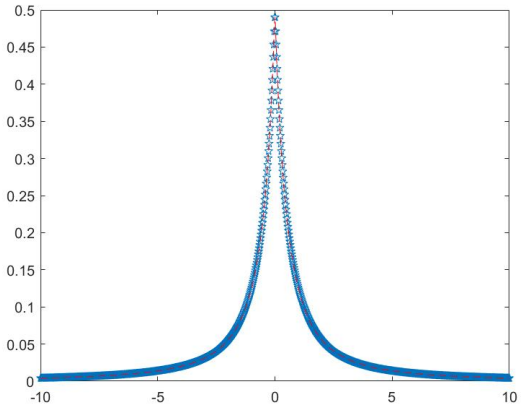


Figure 10: Comparison of derivatives

As we can see the derivative of the approximated Sigmoid goes to zero slightly slower than the original function. This property make this function moderately most robust to the Vanishing Gradient Problem, since when we need to backpropagate the error on an activation with input value far from zero, we have a derivative higher with the approximated function and the error is propagated more consistently. This property can explains the motivation behind the fact that on real numbers, the approximated Sigmoid function gave us better results than the original function.

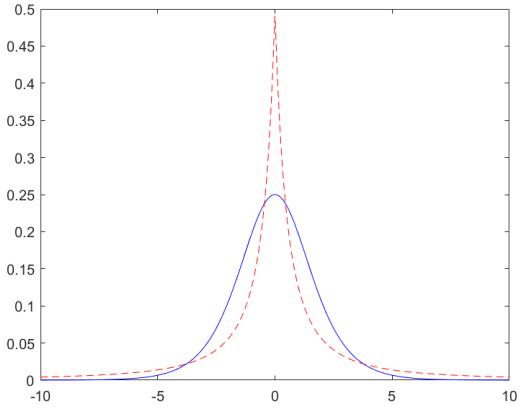


Figure 11: Comparison of derivatives of approximated Gradient and Gradient

Another change is the setting of the minibatch parameter to 60000, this value tells to the fit the number of minibatch to use during the learning phase, setting the parameter equals to the number of training samples we do online learning, updating the weights after every sample. This modification was needed because the operations in the original version was made in batch, so the computations were done packing samples in matrices. This would cause to us problems since we could do computations with Dual Number using DualArray not matrices (this aspect will be addressed in the future works paragraph).

The last major change in the original code was related to the combination of the *feedforward* and *getgradient* functions. Thanks to the use of Dual Numbers, the derivative is calculated directly when evaluating the function and therefore there is no need to separate the functions. The new function is called *feedforwardandgetdelta*.

6 Conclusions

The aim of our work is considered achieved since we demonstrate that we can avoid to compute the derivative using the Dual Numbers. We also understand better how the MLP is actually implemented, having changed some parts of its operation both to make dual numbers work and to add new layers. An important aspect to be considered is the Performance.

6.1 Performance

The performance in terms of Accuracy of the MLP using Dual Numbers and not using them is comparable. So the other metric to be considered is the time to execute the training. Even if the Dual Number implementation avoid to compute explicitly the derivative, this version is actually much slower than the one without dual number. Let's see the reason why this slowness exploiting a wonderful tool made available by Matlab: *profile viewer*.

Here we can see where most of the time is spent during the execution of the main method of the program, the *feedforwardandgetdelta*.

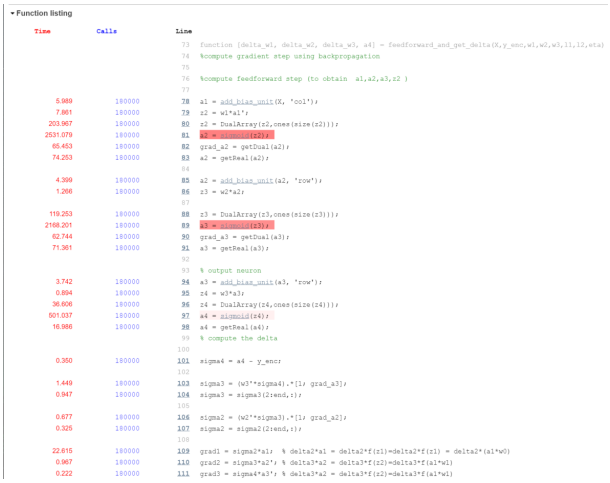


Figure 12: Profile Viewer of feedforwardandgetdelta

The Sigmoid computations take the majority of the execution time, this is something expected since now the computations of the derivative is inserted into the evaluation of the Sigmoid. Like mentioned in the 2.2 Paragraph, using Dual Numbers we can evaluate the derivative of a function in a point during its evaluation, looking at the Dual part of the Dual Number. It is worth mentioning also that the instantiation of the DualArray takes a not negligible time, one motivation behind this can be that this is a class defined by us and not a predefined class of Matlab and so the operations are not optimized.

Let's see now the Sigmoid profile.

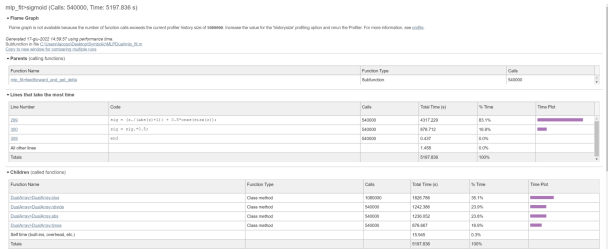


Figure 13: Profile Viewer of sigmoid

Here obviously the most heavy part is the one related to the computations of the approximated Sigmoid, more precisely as we can see from the children section, the 3 main operations that affect the performance are DualArray Plus, DualArray Divide and DualArray Absolute. Let's see them.

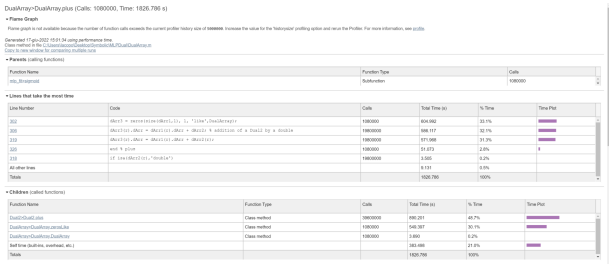


Figure 14: Profile Viewer of DualArray Plus

Here it is interesting to observe that the line that affect mostly the performance is the one related to the instantiation of the Dual Array and not the actual computation of the plus operation. The motivation is discussed before.

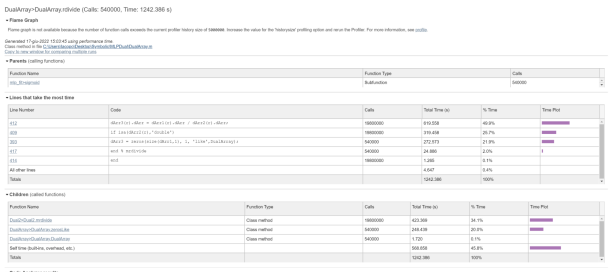


Figure 15: Profile Viewer of DualArray Divide

An aspect that emerges here is that the "isa" operation that control if the object is a double or a Dual2 is unexpectedly onerous. This can be avoided in future implementation, furthermore the branching of the code using "if" is not recommended since the compiler in this way is not facilitated in the optimization of the code and its execution.

The last meaningful operation is the Absolute one, which profile is presented below.

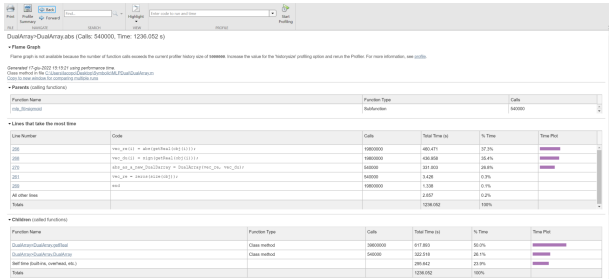


Figure 16: Profile Viewer of DualArray Absolute Value

7 Future Works

During the development of this project interesting ideas came out that can be developed in future works, even starting from our work, the main ones are:

- **Merging Dual2 with DualArray**: the problems related to the performance are mainly due to the usage of a non native class of Matlab. We decided to create 2 different classes Dual2 and DualArray to better model the idea of Dual Numbers separating the operations with the idea of having a vector. But since the use of non-native nested classes slows down the code a lot, you can think of making a single class with the intent of speeding up execution.
- **Implementing DualMatrix**: another problem related to performance is due to the use of online learning instead of mini batches, by implementing the DualMatrix class, you will be able to use minibatches and speed up learning.
- **CNN with Dual Numbers**: this work is focused on the use of Dual Numbers in an MLP, it would also be interesting to implement dual numbers related to CNN.

Appendix A - MLP implementation in Matlab, by Vadim Smolyakov:

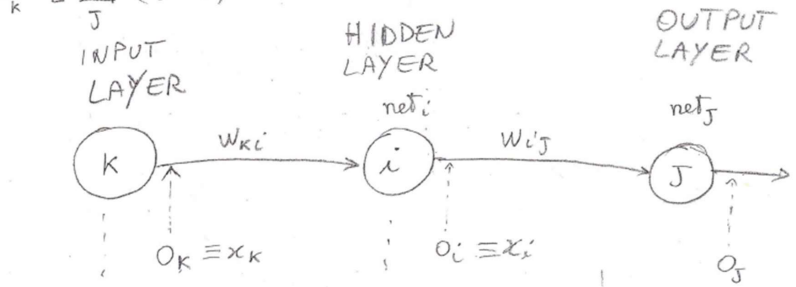
`https://it.mathworks.com/matlabcentral/fileexchange/62365-deep-neural-network`

This code is taken from Kevin Murphy's ML book: `https://probml.github.io/pml-book/book0.html`

The following link, instead, explains why the gradient on the output layer is not needed, in three very important cases at least: `https://www.ics.uci.edu/~pjsadows/notes.pdf`

Appendix B - Notes on back-propagation

$$E_k = \frac{1}{2} \sum_J (t_J - o_J)^2$$



$$\frac{\partial E}{\partial w_{iJ}} = \frac{\partial E}{\partial o_J} \frac{\partial o_J}{\partial net_J} \frac{\partial net_J}{\partial w_{iJ}} =$$

$$= -\underbrace{(t_J - o_J) f'(net_J)}_{\delta_J} x_i = -\delta_J x_i \qquad (o_i = x_i)$$

$$net_J = \sum_{i=0}^{m} x_i w_{iJ} = \sum_{i=0}^{n+1} o_i w_{iJ}$$

$$o_J = f(net_J)$$

$$\frac{\partial E}{\partial w_{Ki}} = \frac{\partial E}{\partial o_J} \frac{\partial o_J}{\partial net_J} \frac{\partial net_J}{\partial o_i} \frac{\partial o_i}{\partial net_i} \frac{\partial net_i}{\partial w_{Ki}} =$$

$$= -\underbrace{\delta_J w_{iJ} f'(net_i)}_{\delta_i} x_K = -\delta_i x_K \qquad (o_K = x_K)$$

If, e.g., neuron i is connected to more output neurons, we have:

$$\delta_i = f'(net_i) \sum_J \delta_J w_{iJ}$$

Appendix C - Notes on Hyper Dual Numbers

Hyper Dual Numbers are an extension of Dual Numbers. We have:

$$x = a + b\varepsilon_1 + c\varepsilon_2 + d\varepsilon_1\varepsilon_2$$

$$\varepsilon_1^2 = \varepsilon_2^2 = (\varepsilon_1\varepsilon_2)^2$$

$$\varepsilon_1 \neq \varepsilon_2 \neq \varepsilon_1\varepsilon_2 \neq 0$$

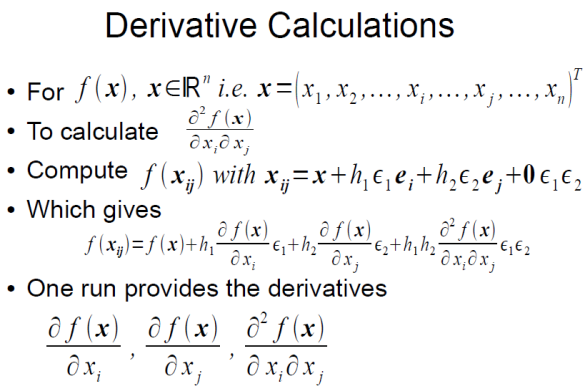


Figure 17: Slide taken from "The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations"

As shown in the slide, using this numbers we can have the exact second derivative calculations directly evaluating the function in that point. This method, as mentioned in this paper from the AEROSPACE DESIGN LAB of Stanford, is more precise with respect to the other methods, even if it is mainly used in aeronautics and mechanics, there is space to enlarge its usage to optimization method that exploits second order derivative.

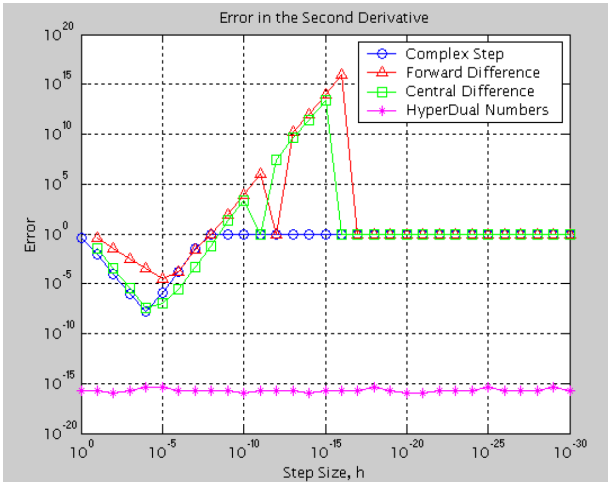


Figure 18: Error Second Derivative Comparison