Deep Learning with S-Shaped Rectified Linear Activation Units

Luca Tomei & Andrea Lombardo

February 2021

Abstract

The task of the project is to create S-shaped Rectified Linear Units (SReLU) from scratch and make a comparison with others activation functions, analyzing performance and behavior of different convolutional neural network. The comparison is done between different non saturated activation functions like ReLU and others Leaky ReLU,PReLU and SReLU, in addition to this also exponential activation functions have been tested.

Contents

1 Introduction

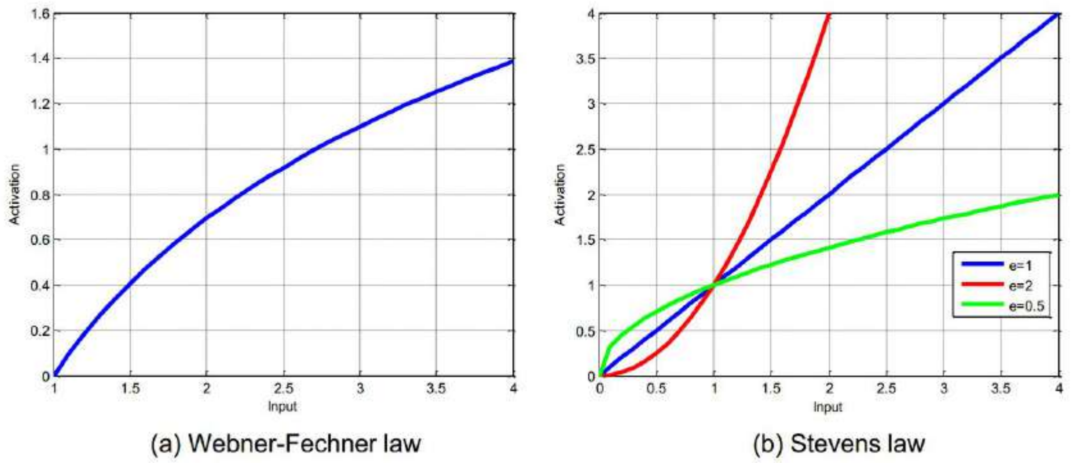
Technological development has involved numerous changes, one of these concerns the classification of images and objects. The classification and numbering of objects is an hard and annoying job for humans in fact nowadays they are aided by different artificial models trained on a machine, namely a computer. Depending on the model used the machine can reach different values of accuracy on the classification task but in general we can say that machines are able to recognize an element and classify it with a little value of miss-classification in comparison with humans. This task helps people saving a lot of time spent in counting and classifying objects without considering that a human is subjected to a miss-classification also for the constant and monotonous job and physical conditions it is subject like illness or more common tiredness. Indeed humans spent a lot of time than a machine for the job of classifying objects due to the fact everyday the continuous evolution and creation of new objects has made the action of classifying objects more important and difficult for us. A branch of Machine Learning that is responsible for the development of algorithm that solve this kind of problem takes the name of Deep Learning. More precisely, in that field we can talk about Deep Convolutional Neural Networks (DCNNs) where deep refers to the fact that the neural network has several layer and these networks have shown a great success in image classification and in this project we want to analyze the performance in terms of accuracy of different activation function, focusing on the non-saturated ones which allow to solve the problem of exploding/vanishing gradient and let the network to converge faster than the saturated ones. The examined paper [1] investigates the usage of a new activation function called S-shaped Rectified Linear Unit (SReLU) which is able to learn both convex and non-convex functions. The SReLU design is inspired by two biophysics and neural sciences laws, namely Webner-Fechner and Stevens, which describe the relationship between magnitude stimulus and its perception.

Webner-Fechner law is represented by the following logarithmic function :

$$s = k \log p$$

where s is the perceived magnitude computed applying the logarithmic function of the stimulus intensity p multiplied by a specific constant k. On the other hand, Stevens law focuses on the relationship of the same terms mentioned in the previous law but also include a new parameter e which depend of the kind of stimulus analyzed. In the below pictures we can see the two mentions laws taken from [1].

$$s = kp^e$$



SReLU is the activation function which tries to imitate the relationship between laws, in other words, how a perceived object is subject to react to a stimulus.

2 Activation Functions

In this section we will describe in a more detailed way the activation functions, linear and exponential ones, used in the experiments. For each of them it will be described the mathematical definition, their behaviour and their behaviour depending on the input they get.

2.1 ReLU

The Rectified Linear Unit activation function known as ReLU is a non saturated linear function that will directly return the input itself if it is positive, otherwise it will return zero. It is the default activation function for many types of neural networks because a model that uses it is easier to train, it doesn't need additional parameters and often it achieves better performance. Below there is the mathematical definition of the ReLU and its graphical representation.

$$h(x_i) = max(0, x_i)$$

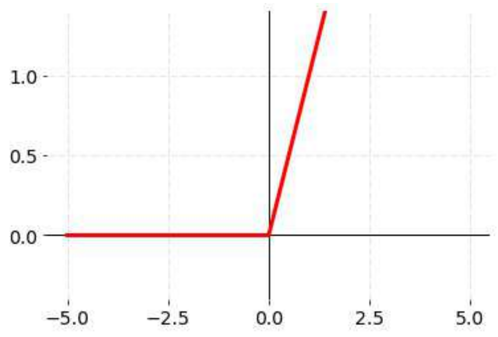


Figure 1: Rectified Linear Unit activation function

2.2 Parametric ReLU and Leaky ReLU

PReLU is the parametric form of the ReLU whereas the LeakyReLU is a particular form of the PReLu in facts, as it can be seen from the equations (2) and (3), all of them share the same behaviour for positive numbers while for the negative ones the LeakyReLU has a line with a positive slope fixed at 0.01 while the PReLU considers that slope as a parameter ($a_i \in (0, 1)$) of the activation function and because of that it has to be computed during the training time. In the following there is the equation that describe the Parametric ReLU activation function:

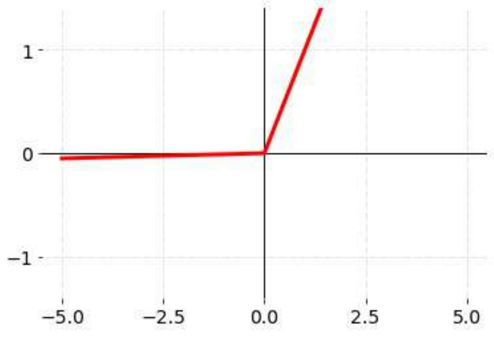
$$h(x_i) = min(0, a_i x_i) + max(0, x_i)$$

The PReLU and Leaky ReLU activation functions can be summarized as the below linear functions:

$$f(x) = \begin{cases} a_i x & \forall\ x \leq 0, \quad a_i \in (0, 1) \\ x & \forall x \geq 0 \end{cases} \tag{1}$$

Fixing the a_i parameter to the value 0.01 we get the Leaky ReLU, defined in the following way:

$$f(x) = \begin{cases} 0.01x & \forall\ x \leq 0 \\ x & \forall x \geq 0 \end{cases} \tag{2}$$


(a) Leaky ReLU

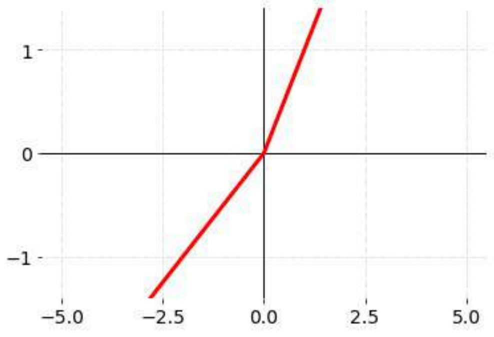

(b) PReLU

Figure 2

2.3 S-shaped ReLU

SReLU performs a mapping $\mathbb{R} \rightarrow \mathbb{R}$ and could be seen as the combination of three different piece-wise linear functions, which depend by four trainable parameters $\{a_i^r,\ t_i^r,\ a_i^l,\ t_i^l\}$ that let to reach a flexibility never obtained until now with other activation functions because it's able to learn convex and non convex function.

$$f(x_i) = \begin{cases} t_i^l + a_i^l(x_i - t_i^l) & \forall\ x_i \leq t_i^l \\ x_i & \forall\ t_i^l < x_i < t_i^r \\ t_i^r + a_i^r(x_i - t_i^r) & \forall\ x_i \geq t_i^r \end{cases} \tag{3}$$

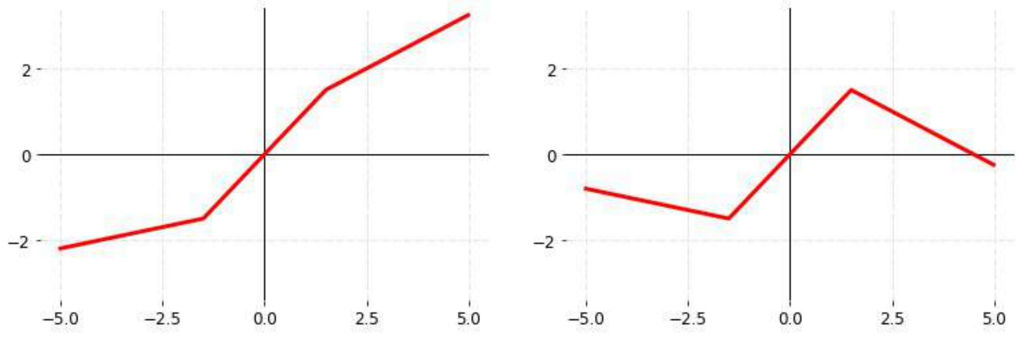
subterm i refers to the fact that the SReLU's values change in different channels,

term a used when the value exceed the threshold, so it refers to slope of the function and if it has the upper-term **r** refers to the line for the right part, on the other hand with the upper-term **l** it refers to the line for left part

term t refers to the intercept or threshold.

It can be noted that using different numbers for the parameter of the SReLU it assumes different shapes, in particular in the following it will be shown how the SReLU activation function can imitate the Stevens law and the Webner-Fechner law:

- when $t_i^r > 1$ and $a_i^r > 0$ it imitates the power function with the exponent $e > 1$ (Stevens Law);
- when $t_i^r = 1$ and $a_i^r > 0$ it imitates the power function with the exponent $e = 1$ (Stevens Law);
- when $1 > t_i^r > 0$ and $a_i^r > 0$ it imitates the logarithm function (Webner-Fechner law);



(a) $a_l = 0.199, t_l = -1.5, a_r = 0.5, t_r = 1.5$ (b) $a_l = -0.199, t_l = -1.5, a_r = -0.5, t_r = 1.5$

Figure 3: S-shaped ReLU

How is it possible to see from the above pictures and equations, we can split SReLU function into three different linear functions:

- $(t_i^l,\, t_i^r)$ $\rightarrow$the SReLU is represented by a linear function(x_i) with unit slope 1 and bias 0
- $x < t_i^l$ $\rightarrow$ the SReLU is represented by $t_i^l + a_i^l(x_i - t_i^l)$ where t_i^l is the intercept of the equation instead a_i^l is the slop of the equation.
- $x > t_i^r$ $\rightarrow$ the SReLU is represented by $t_i^r + a_i^r(x_i - t_i^r)$ where t_i^r is the intercept of the equation instead a_i^r is the slop of the equation.

2.4 ELU and Prametric ELU

Until now we have seen functions composed by polynomial functions of the first degree and the output of those functions will not be constrained in any range but they have the problem to not be differentiable in their entire domain in fact all the linear activation function mentioned before are not differentiable in the origin of the axis. This problem can lead to computation errors into the gradient-based optimization algorithm used to train the neural network. Now we focus on Exponential Activation functions which are able to solve the problems of the others described before due to the fact they are continuously differentiable so they don't have singularities in the domain. Into this class we focused on Exponential Linear Unit activation function and its parametric form, namely the Parametric ELU (PELU). The ELU activation function becomes smooth slowly until its output is equal to $-\alpha$ and for this reasons it has an exponential behaviour for negative numbers while for the positive ones it has a classical linear function with slope equal to 1. Exponential Linear Unit (ELU) is a function that tend to converge to zero faster compared to other ones to and produce more accurate results. Different to other activation functions, ELU has a extra α constant which should be a positive number. The parametric form of the ELU insert two positive parameters a, b both higher than 0.

$$f(x) = \begin{cases} \alpha(e^{\frac{x}{b}} - 1) & \forall\ x \leq 0 \\ \frac{a}{b}x & \forall x \geq 0 \end{cases} \tag{4}$$

Modifying the a and b value as e we obtain the ELU from PRELU:

$$f(x) = \begin{cases} \alpha(e^{x} - 1) & \forall\ x \leq 0 \\ x & \forall x \geq 0 \end{cases} \tag{5}$$

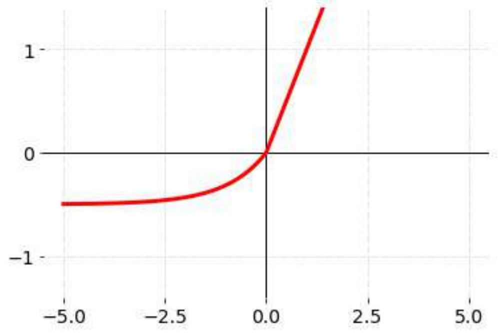


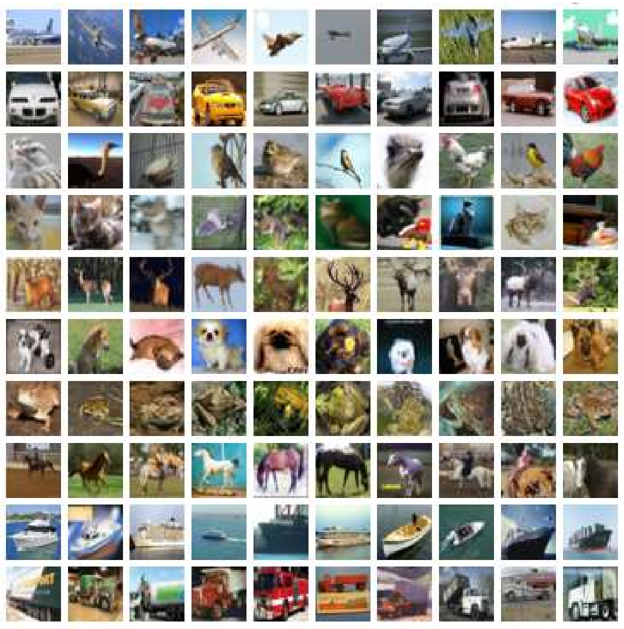
Figure 4: Exponential Linear Unit (ELU) activation function

3 Dataset

In the original paper and in the project we developed we have used two of the most commonly used datasets, namely MNIST and CIFAR-10 which are described more in detail in the following subsections.

3.1 CIFAR-10

The CIFAR-10 dataset is composed by 60,000 of 32x32 colour images from 10 classes, such as airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. We have chosen to have 50,000 images for the training dataset and 10,000 for the test set. In the following we can see some images taken from the dataset.



3.2 MINST

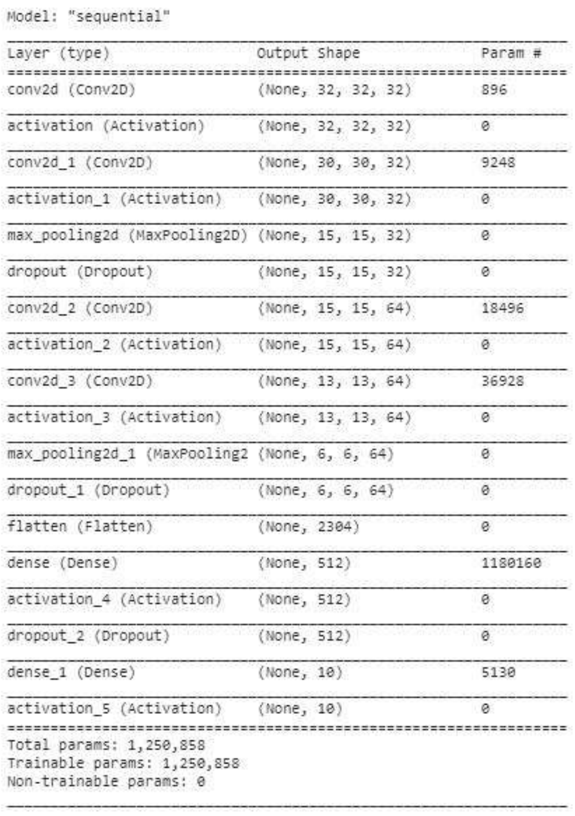
The MINST dataset is composed by 70,000 small square 28x28 gray scale images that contains single handwritten integer number between 0 to 9, divided as 60,000 images for training and 10,000 images for testing. In the following we can see some images taken from the dataset.

4 Design

After having discussed about different activation functions focusing on their mathematical definition and their behaviour, now we need a vehicle to compare them and in particular we compared 4 different architecture, 2 for the CIFAR-10 dataset and 2 for the MNIST one, in order to choose the best one and to use it as a benchmark for the comparison of the activation function. This preliminary comparison between the 4 architecture has been carried out using the most commonly used activation function. For these architectures we relied on the Keras library, which is based on the TensorFlow one, to build the Convolutional Neural Network (CNN) we used.

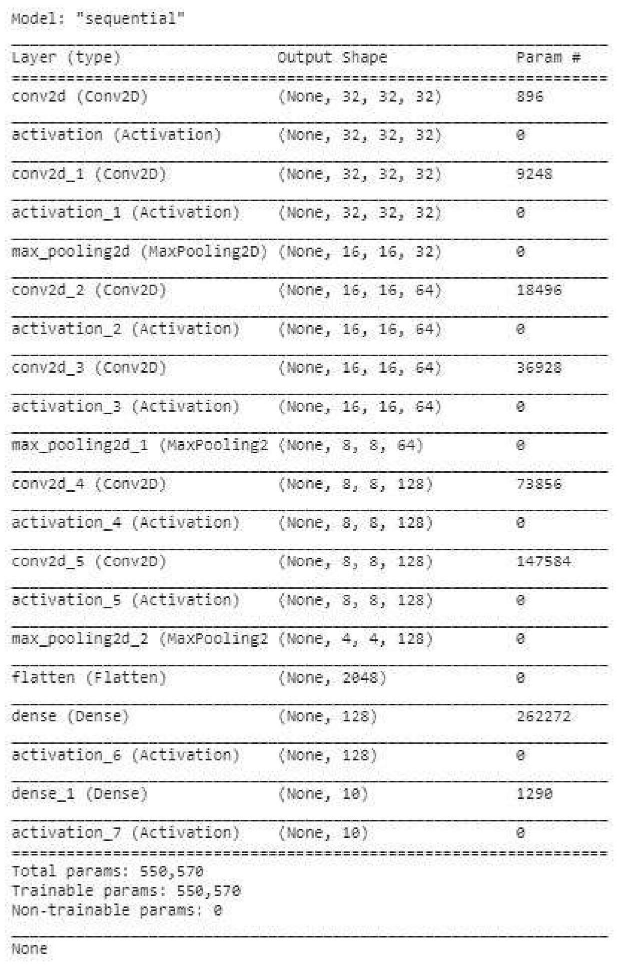
4.1 Convolutional Neural Networks for CIFAR-10

The first CNN architecture for the CIFAR-10 dataset is composed by 4 convolutional layers, 2 max pooling layers, 3 dense layers as it can be seen from the following resume of this CNN obtained with **model.summary()** TensorFlow call. It has to be noted that in the experiments we refer to this CNN as CNN1_CIFAR10.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896
_________________________________________________________________
activation (Activation)      (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_1 (Activation)    (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 15, 15, 64)        18496
_________________________________________________________________
activation_2 (Activation)    (None, 15, 15, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 13, 64)        36928
_________________________________________________________________
activation_3 (Activation)    (None, 13, 13, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1180160
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_5 (Activation)    (None, 10)                0
=================================================================
Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0
_________________________________________________________________
```

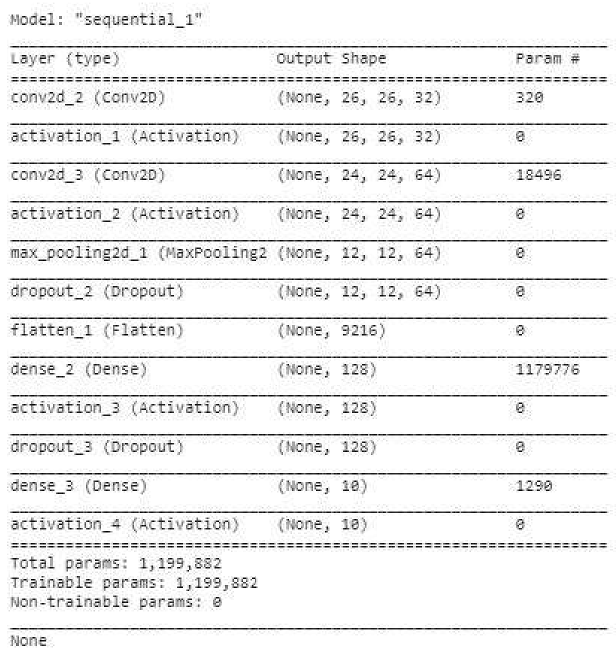
The second CNN architecture for the CIFAR-10 dataset is composed by 6 convolutional layers, 3 max pooling layers, 2 dense layers. In the following we can see a resume of this CNN obtained with **model.summary()** TensorFlow call. It has to be noted that in the experiments we refer to this CNN as CNN2_CIFAR10.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 32, 32, 32)        896       
_________________________________________________________________
activation (Activation)      (None, 32, 32, 32)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248      
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0         
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496     
_________________________________________________________________
activation_2 (Activation)    (None, 16, 16, 64)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        36928     
_________________________________________________________________
activation_3 (Activation)    (None, 16, 16, 64)        0         
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 8, 8, 128)         73856     
_________________________________________________________________
activation_4 (Activation)    (None, 8, 8, 128)         0         
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 128)         147584    
_________________________________________________________________
activation_5 (Activation)    (None, 8, 8, 128)         0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 128)         0         
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0         
_________________________________________________________________
dense (Dense)                (None, 128)               262272    
_________________________________________________________________
activation_6 (Activation)    (None, 128)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290      
_________________________________________________________________
activation_7 (Activation)    (None, 10)                0         
=================================================================
Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0
_________________________________________________________________
None
```

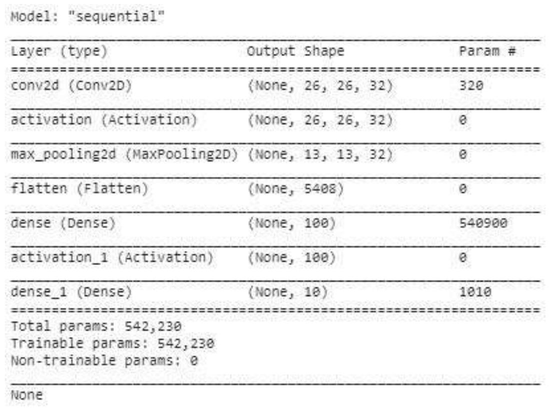
4.2 Convolutional Neural Networks for MNIST

The first CNN architecture for the MNIST dataset is composed by 2 convolutional layers, 1 max pooling layer, 3 dense layers. It has to be noted that in the experiments we refer to this CNN as CNN1_MNIST. Here we can see a schema of this CNN obtained with **model.summary()** TensorFlow call:



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_2 (Conv2D)            (None, 26, 26, 32)        320       
_________________________________________________________________
activation_1 (Activation)    (None, 26, 26, 32)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 64)        18496     
_________________________________________________________________
activation_2 (Activation)    (None, 24, 24, 64)        0         
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 64)        0         
_________________________________________________________________
dropout_2 (Dropout)          (None, 12, 12, 64)        0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 9216)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 128)               1179776   
_________________________________________________________________
activation_3 (Activation)    (None, 128)               0         
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290      
_________________________________________________________________
activation_4 (Activation)    (None, 10)                0         
=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
None
```

The second CNN architecture for the MNIST dataset is made up by 1 convolutional layers, 1 max pooling layer, 2 dense layers. It has to be noted that in the experiments we refer to this CNN as CNN2_MNIST. Here we can see a resume of this CNN obtained with **model.summary()** TensorFlow call:



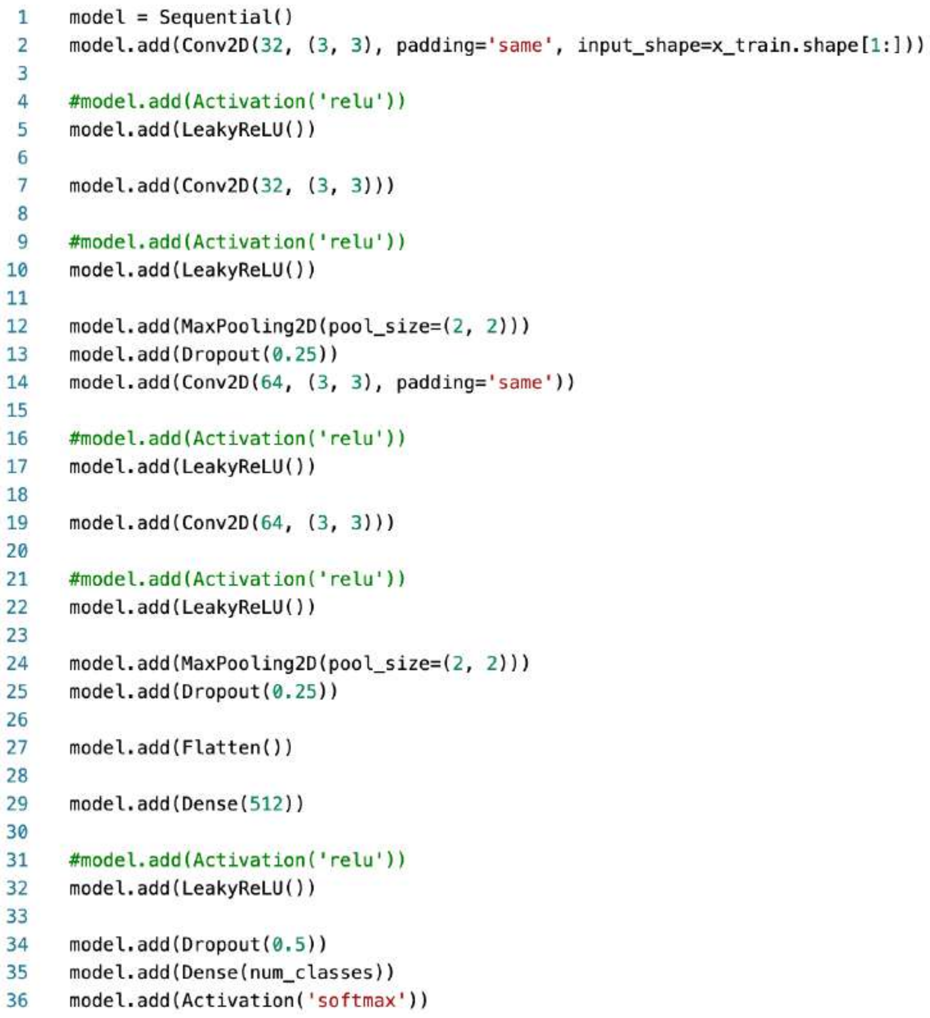
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320       
_________________________________________________________________
activation (Activation)      (None, 26, 26, 32)        0         
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0         
_________________________________________________________________
flatten (Flatten)            (None, 5408)              0         
_________________________________________________________________
dense (Dense)                (None, 100)               540900    
_________________________________________________________________
activation_1 (Activation)    (None, 100)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1010      
=================================================================
Total params: 542,230
Trainable params: 542,230
Non-trainable params: 0
_________________________________________________________________
None
```

5 Implementations

In this section we will be showing the architectures that backed the experiments in particular for the CNNs and the custom SReLU.

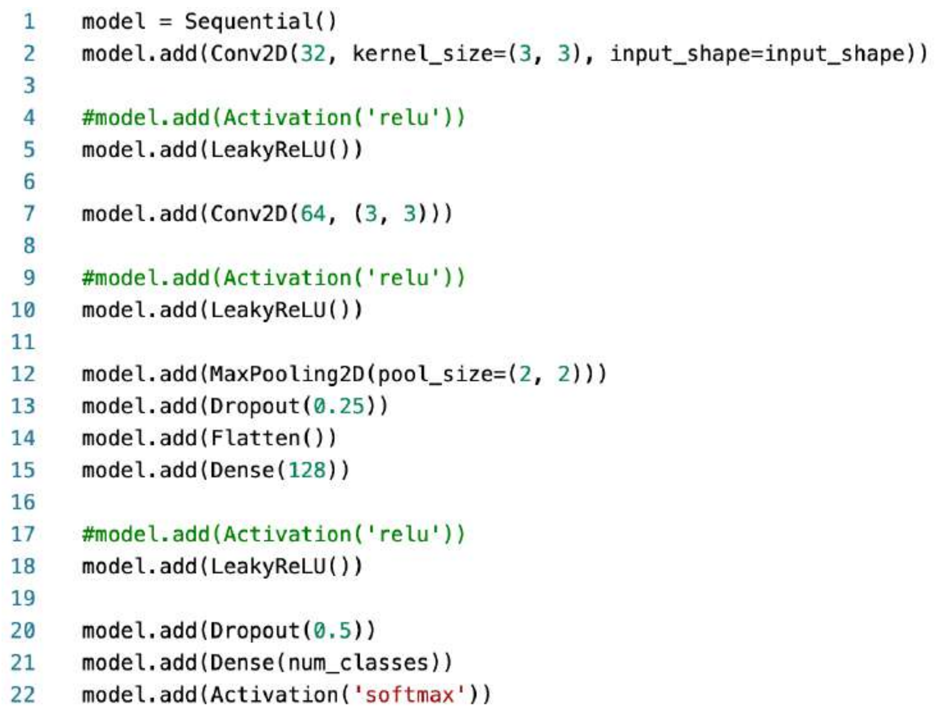
5.1 Convolutional Neural Networks implementation

As mentioned before, we did some preliminary experiments using two different convolutional neural networks in order to choose which one performs better in terms of accuracy and loss and then use it as a standard upon which test the activation funcitons. The first one couple of CNNs present an architecture inspired by the one present in the Keras website, for the CIFAR-10 dataset [10] and for the MNIST dataset [11]. On the other hand, the second one couple of CNNs is taken from an online machine learning blog named machinelearningmastery.com, in particular the CNN for the CIFAR10 dataset [13] and for the MNIST dataset [14]. All these implementation are showed in the following images.



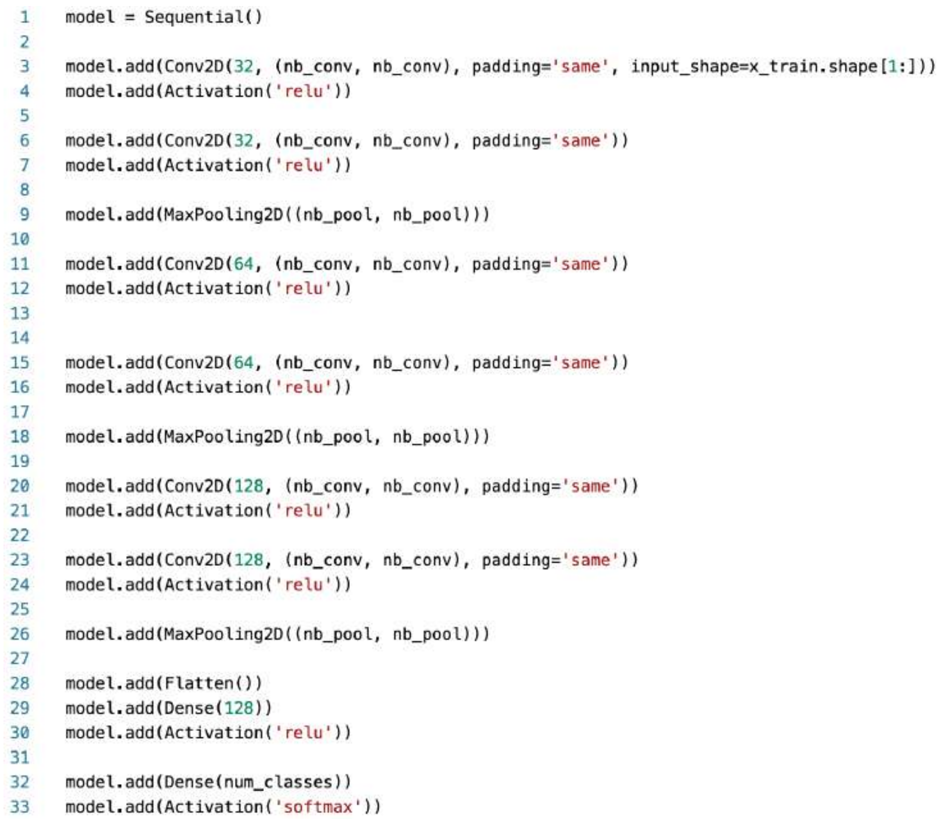
```python
model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(Conv2D(32, (3, 3)))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(Conv2D(64, (3, 3)))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

Figure 5: CNN1 for CIAFR10



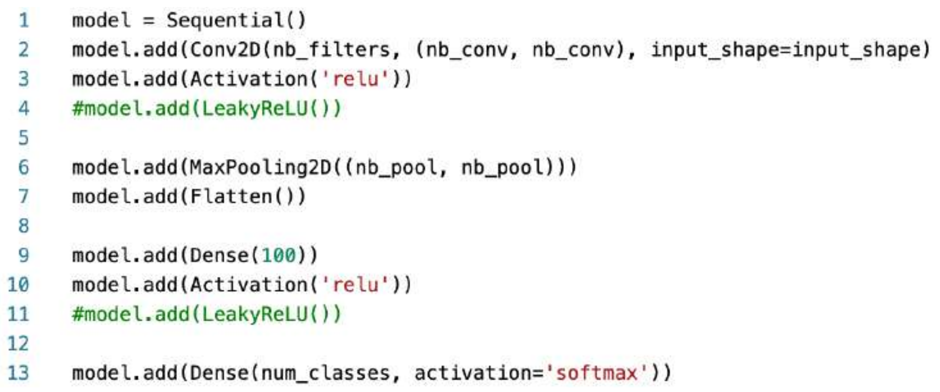
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=input_shape))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(Conv2D(64, (3, 3)))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128))

#model.add(Activation('relu'))
model.add(LeakyReLU())

model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

Figure 6: CNN1 for MNIST



```
model = Sequential()

model.add(Conv2D(32, (nb_conv, nb_conv), padding='same', input_shape=x_train.shape[1:]))
model.add(Activation('relu'))

model.add(Conv2D(32, (nb_conv, nb_conv), padding='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D((nb_pool, nb_pool)))

model.add(Conv2D(64, (nb_conv, nb_conv), padding='same'))
model.add(Activation('relu'))


model.add(Conv2D(64, (nb_conv, nb_conv), padding='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D((nb_pool, nb_pool)))

model.add(Conv2D(128, (nb_conv, nb_conv), padding='same'))
model.add(Activation('relu'))

model.add(Conv2D(128, (nb_conv, nb_conv), padding='same'))
model.add(Activation('relu'))

model.add(MaxPooling2D((nb_pool, nb_pool)))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))

model.add(Dense(num_classes))
model.add(Activation('softmax'))
```

Figure 7: CNN2 for CIAFR10



```
model = Sequential()
model.add(Conv2D(nb_filters, (nb_conv, nb_conv), input_shape=input_shape)
model.add(Activation('relu'))
#model.add(LeakyReLU())

model.add(MaxPooling2D((nb_pool, nb_pool)))
model.add(Flatten())

model.add(Dense(100))
model.add(Activation('relu'))
#model.add(LeakyReLU())

model.add(Dense(num_classes, activation='softmax'))
```

Figure 8: CNN2 for MNIST

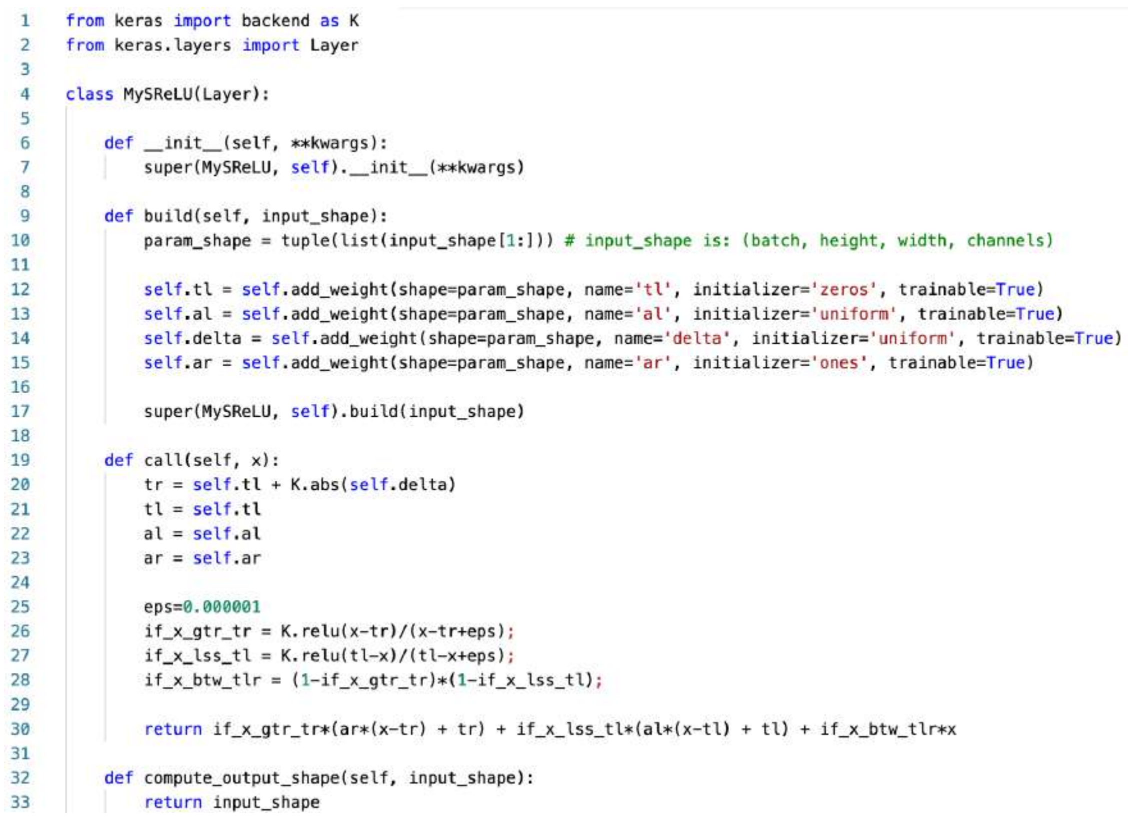
5.2 MySReLU implementation

The implementation by scratch of this activation function is the main goal of this project. Therefore we 've compared our implementation of SReLU with that developed by the Keras Library so we called our version as "MySReLU" in order to distinguish the two implementations. Following the keras documentation [12] we implement our version of the S-shaped Rectified Linear Unit, in order to see how to develop a custom activation function.

In details we implement this using only three methods :

- `build(input_shape)`: where you will define custom weights. This method must set `self.built = True` at the end, which can be done by calling `super([Layer], self).build()`;
- `call(x)`: where the layer's logic lives. Unless you want your layer to support masking, you only have to care about the first argument passed: the input tensor;
- `compute_output_shape(input_shape)`: in case your layer modifies the shape of its input, you should specify here the shape transformation logic. This allows Keras to do automatic shape inference.

In particular following the order of the list of methods, we introduced the four parameters we need to specify: t_l, a_l, *delta* and a_r described above in 2.3 Section. *delta* is used for obtain t_r depending by t_l. It's necessary in a way to avoid that t_r is at left of t_l. Into the function call we've created three Boolean variables (`if_x_gtr_tr, if_x_lss_tl, if_x_btw_tlr`) that allow us to verify where the input x is for detecting the range and so the behaviour of the right activation function that we need to show. In order to complete this task we use Keras backend (`K.relu()`): in particular we made a step function that returns 1 if the input is in that range (e.g. $x \leq t_r$), 0 otherwise. In case we had used a pure version of the step function, we would have had computational problems due to gradient-based optimization when passing through a danger point ($x = t_r$ or $x = t_l$) where the function is not differentiable. To solve this problem, we added epsilon, an arbitrarily small quantity, in order to avoid a division of type 0/0. As a result, we constructed a smoothed version of the step function. The last argument of the call returns the right chunk of the SReLU, based on the value returned by the entire step function. In the following image there is shown the code of the implementation of the SReLU.



```python
from keras import backend as K
from keras.layers import Layer

class MySReLU(Layer):

    def __init__(self, **kwargs):
        super(MySReLU, self).__init__(**kwargs)

    def build(self, input_shape):
        param_shape = tuple(list(input_shape[1:])) # input_shape is: (batch, height, width, channels)

        self.tl = self.add_weight(shape=param_shape, name='tl', initializer='zeros', trainable=True)
        self.al = self.add_weight(shape=param_shape, name='al', initializer='uniform', trainable=True)
        self.delta = self.add_weight(shape=param_shape, name='delta', initializer='uniform', trainable=True)
        self.ar = self.add_weight(shape=param_shape, name='ar', initializer='ones', trainable=True)

        super(MySReLU, self).build(input_shape)

    def call(self, x):
        tr = self.tl + K.abs(self.delta)
        tl = self.tl
        al = self.al
        ar = self.ar

        eps=0.000001
        if_x_gtr_tr = K.relu(x-tr)/(x-tr+eps);
        if_x_lss_tl = K.relu(tl-x)/(tl-x+eps);
        if_x_btw_tlr = (1-if_x_gtr_tr)*(1-if_x_lss_tl);

        return if_x_gtr_tr*(ar*(x-tr) + tr) + if_x_lss_tl*(al*(x-tl) + tl) + if_x_btw_tlr*x

    def compute_output_shape(self, input_shape):
        return input_shape
```

Figure 9: MySReLU implementation

6 Experiments

In this section we will show the accuracy and loss values obtained with the various CNN architectures using a different activation function. in particular, we made a first test using all the implemented CNN and using the classic activation functions, that is ReLU, Leaky ReLU, PReLU and ELU, to see which CNN behaved better on the respective datasets. Once it was established which CNN had the best performance, these were used as standard to carry out further and more in-depth experiments by verifying their behavior with the activation functions MySReLU and SReLU in addition to those previously mentioned. From the experiments it emerged that CNN1 was the one that behaved better on the CIFAR10 dataset while on the MNIST dataset CNN2 was the one that had the best performances and in both cases there was a significant gap in terms of accuracy and loss between the best and the worst CNN. Specifically CNN1 obtained on the CIFAR10 dataset accuracy values between 0.77 and 0.78 while CNN2 on the same dataset obtained values between 0.69 and 0.76 obtaining therefore a greater volatility in the performances. Since the experiments with the CNN1 on the CIFAR10 showed a low variability in the accuracy values it could be inferred that the use of a different activation function is not such a determining factor in obtaining high accuracy values as the architecture of the CNN itself. Nevertheless the MySReLU and ELU activation functions showed the highest accuracy values with the latter also obtaining a faster convergence to the final accuracy value. In the case of the MNIST dataset, CNN1 obtained accuracy values between 0.88 and 0.89 while CNN2 obtained values between 0.98 and 0.99 and also in this case the considerations previously made remained valid, in fact MySReLU and ELU obtained the highest accuracy values. The results of these experiments will be presented in tabular form below.

Table 1: CNN1 on CIFAR10

Activation function	Accuracy	Loss
ReLU	0.7808	0.6608
Leaky ReLU	0.7826	0.6302
PReLU	0.7760	0.6663
SReLU	0.7779	0.6577
MySReLU	0.7873	0.8037
ELU	0.7817	0.6565
PELU	0.7797	0.6757

Table 2: CNN2 on CIFAR10

Activation function	Accuracy	Loss
ReLU	0.7204	1.6560
Leaky ReLU	0.7167	1.6220
PReLU	0.6938	1.9421
ELU	0.7603	1.7570

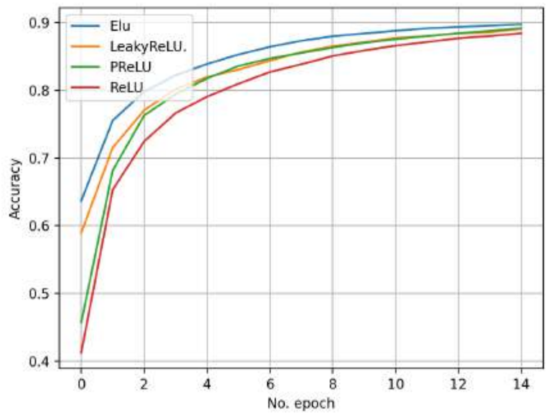
Table 3: CNN1 on MNIST

Activation function	Accuracy	Loss
ReLU	0.8841	0.4305
Leaky ReLU	0.8905	0.4013
PReLU	0.8918	0.4194
ELU	0.8974	0.3692

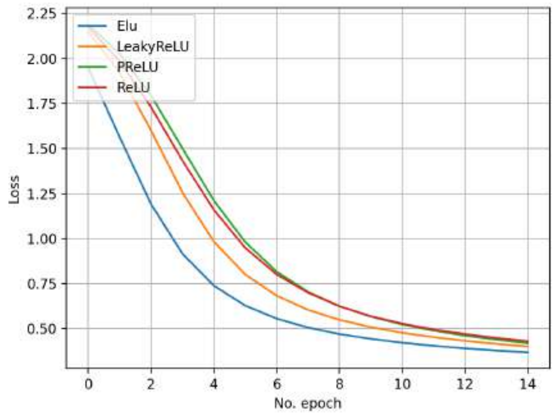
Table 4: CNN2 on MNIST

Activation function	Accuracy	Loss
ReLU	0.9874	0.0437
Leaky ReLU	0.9861	0.0450
PReLU	0.9859	0.0552
SReLU	0.9863	0.0468
MySReLU	0.9926	0.0260
ELU	0.9875	0.0456
PELU	0.9860	0.0540

In the following we will show graphs of the accuracy and loss values obtained by CNNs at each epoch on the CIFAR10 and MNIST datasets.

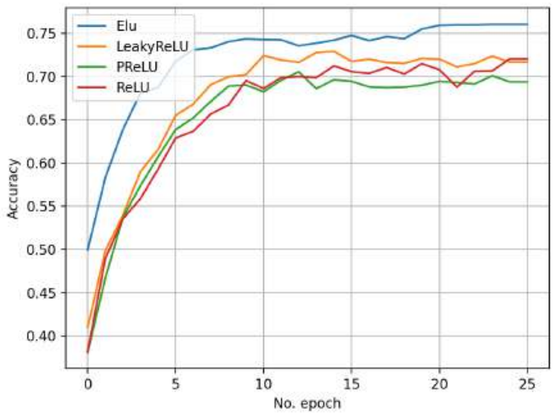


(a) *CNN1 on MNIST accuracy*

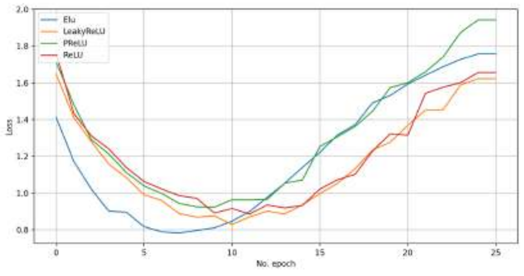


(b) *CNN1 on MNIST loss*

Figure 10: CNN1 on MNIST (the worst CNN)

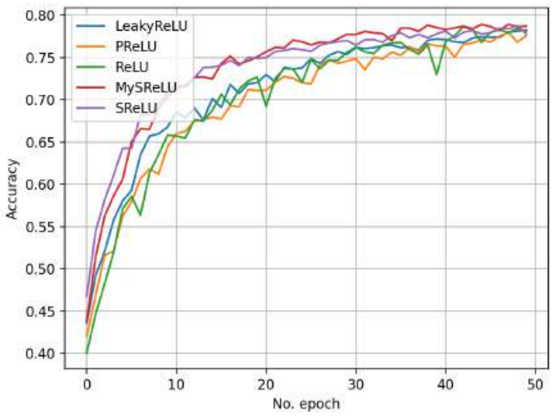


(a) *CNN2 on CIFAR10 accuracy*

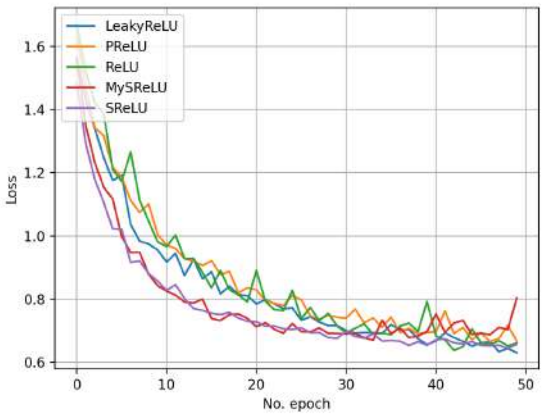


(b) *CNN2 on CIFAR10 loss*

Figure 11: CNN2 on CIFAR10 (the worst CNN)

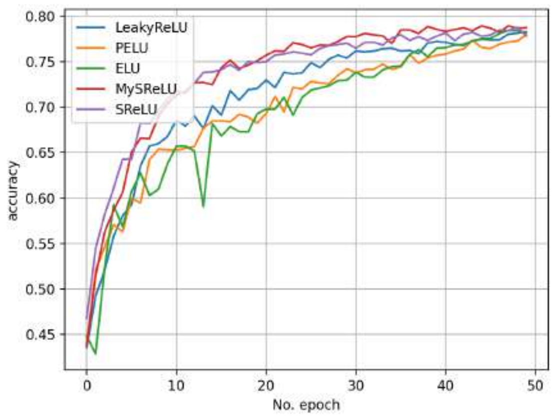


(a) *CNN1 on CIFAR10 accuracy*

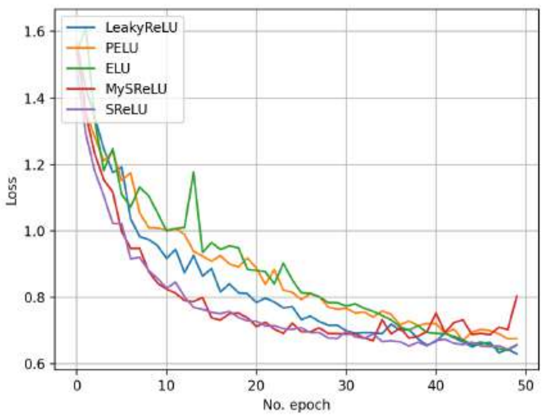


(b) *CNN1 on CIFAR10 loss*

Figure 12: CNN1 on CIFAR10 (linear activation function)

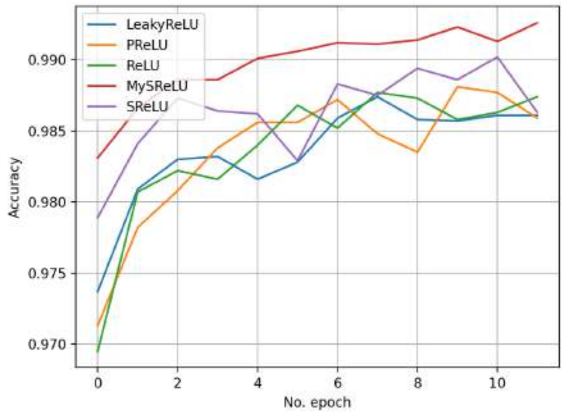


(a) *CNN1 on CIFAR10 accuracy*

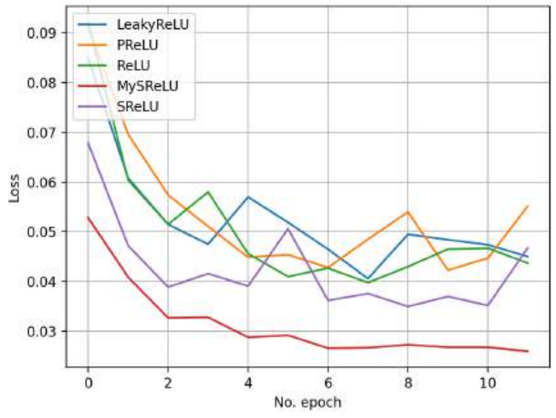


(b) *CNN1 on CIFAR10 loss*

Figure 13: CNN1 on CIFAR10 (exponential activation function)

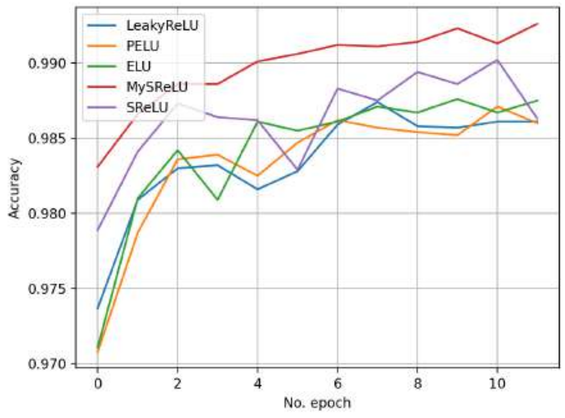


(a) *CNN2 on MNIST accuracy*

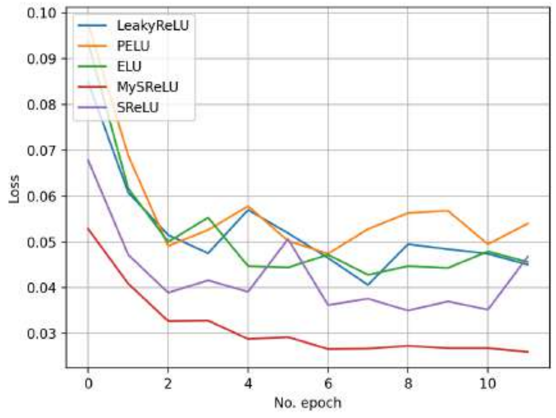


(b) *CNN2 on MNIST loss*

Figure 14: CNN2 on MNIST (linear activation function)



(a) *CNN2 on MNIST accuracy*



(b) *CNN2 on MNIST loss*

Figure 15: CNN2 on MNIST (exponential activation function)

7 Conclusion and further development

In conclusion, to test the usefulness of having an S-shaped rectified activation function, we could extract, after training the neural network with such an activation function, the values obtained for a_l, a_r, t_l, and t_r and plot them to see if this function is S-shaped or collapsed into a more common activation function such as ReLU and PReLU and whether or not the neural network rejects such complexity. Unfortunately, we were unable to capture these values and thus could not perform such an examination due to the fact that Keras/TensoFlow hide this kind of detail of the implementation. For future developments we could think of testing this kind of activation function with different datasets, different architectures or neural networks for tasks different from those studied in the paper, for example not computer vision tasks but natural language processing and prediction of time series.

References

[1] Original Paper - *Deep Learning with S-shaped Rectified Linear Activation Units*, Xiaojie Jin, Chunyan Xu, Jiashi Feng, Yunchao Wei, Junjun Xiong and Shuicheng Yan, Cornell University - `https://arxiv.org/abs/1512.07030`

[2] Leaky ReLU - *Rectifier Nonlinearities Improve Neural Network Acoustic Models*, Andrew L. Maas, Awni Y. Hannun and Andrew Y. Ng, Stanford University - `https://pdfs.semanticscholar.org/367f/2c63a6f6a10b3b64b8729d601e69337ee3cc.pdf`

[3] PReLU - *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, Cornell University - `https://arxiv.org/abs/1502.01852`

[4] ELU - *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*, Djork-Arné Clevert, Thomas Unterthiner and Sepp Hochreiter, Cornell University - `https://arxiv.org/abs/1511.07289`

[5] PELU - *Parametric Exponential Linear Unit for Deep Convolutional Neural Networks*, Ludovic Trottier, Philippe Giguère and Brahim Chaib-draa, Cornell University - `https://arxiv.org/abs/1605.09332`

[6] Keras Python Library - `https://keras.io/`

[7] MNIST Database - `http://yann.lecun.com/exdb/mnist/`

[8] CIFAR-10 Dataset - `https://www.cs.toronto.edu/~kriz/cifar.html`

[9] Activation Function - `https://en.wikipedia.org/wiki/Activation_function`

[10] CNN1_CIFAR10 (Keras) - `https://keras.io/examples/cifar10_cnn/`

[11] CNN1_MINIST (Keras) - `https://keras.io/examples/mnist_cnn/`

[12] Writing your own Keras layers - `https://keras.io/layers/writing-your-own-keras-layers/`

[13] CNN2_CIFAR10 (machinelearningmastery) - `https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/`

[14] CNN2_MINIST (machinelearningmastery) - `https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classificat`

[15] Stevens Law - *On the psychophysical law. Psychological review 64(3):153*, Stevens, S. S. 1957

[16] Webner-Fechner Law - *Annotationes anatomicae et physiolog- icae [anatomical and physiological annotations]. Leipzig: CF Koehler*, Webner, E. 1851