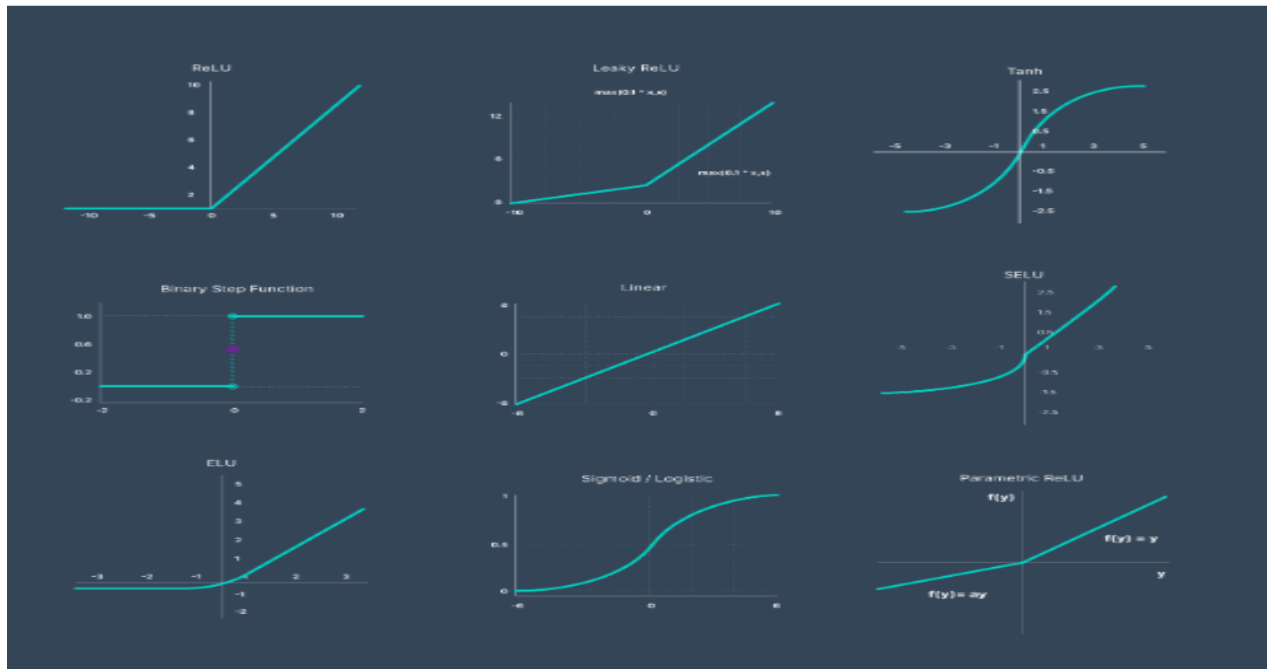


ACTIVATION FUNCTIONS



ACTIVATION FUNCTIONS:

1. SIGMOID ACTIVATION FUNCTION- **Multilabel classification, Binary Classification**
2. TANH ACTIVATION FUNCTION
3. SWISH ACTIVATION FUNCTION
4. SOFTMAX ACTIVATION FUNCTION-**Multiclass Classification**
5. RELU
6. LEAKYRELU
7. EXPONENTIAL LEAKY RELU(ELU)
8. PARAMETRIC LEAKY RELU(PReLU)
9. MAXOUT
10. SOFTPLUS
11. STEP FUNCTION-**BINARY CLASSIFIER**
12. LINEAR ACTIVATION FUNCTION

ACTIVATION FUCTIONS THAT ARE BEST TO BE USED FOR IMAGE PROCESSING USING NEURAL NETWORK

1. SIGMOID FUNCTION

The sigmoid function is a special form of the logistic function and is usually denoted by $\sigma(x)$ or $\text{sig}(x)$. It is given by:

$$\sigma(x) = 1/(1+\exp(-x))$$

PROPERTIES OF SIGMOID FUNCTION

A few other properties include:

1. Domain: $(-\infty, +\infty)$
2. Range: $(0, +1)$
3. $\sigma(0) = 0.5$
4. The function is monotonically increasing.
5. The function is continuous everywhere.
6. The function is differentiable everywhere in its domain.
7. Numerically, it is enough to compute this function's value over a small range of numbers, e.g., $[-10, +10]$. For values less than -10, the function's value is almost zero. For values greater than 10, the function's values are almost one. For value 0, it's 0.5. It implies that for values greater than 10 or less than -10, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the *Vanishing gradient* problem.

PROS

- It is nonlinear in nature.
- It will give an analog activation unlike step function.
- It has a smooth gradient too.
- It's good for a classifier.

- The output of the activation function is always going to be in range (0,1) compared to $(-\infty, \infty)$ of linear function. So we have our activations bound in a range.

CONS

- Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.
- It gives rise to a problem of “vanishing gradients”.
- Its output isn’t zero centered. It makes the gradient updates go too far in different directions. $0 < \text{output} < 1$, and it makes optimization harder.
- Sigmoids saturate and kill gradients.
- The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits).

Further reading

- [Yes You Should Understand Backprop](#), Karpathy (2016)

CODE FOR SIGMOID ACTIVATION FUNCTION:

- **VERILOG CODE:**

```
module sigmoid(
input signed [31:0] x,
output reg signed [31:0] y
);
//SIGMOID FUNCTION
function real sigmoid func;
input real x;
begin
sigmoid func = 1.0/ (1.0+ $exp(-x));
end
endfunction
```

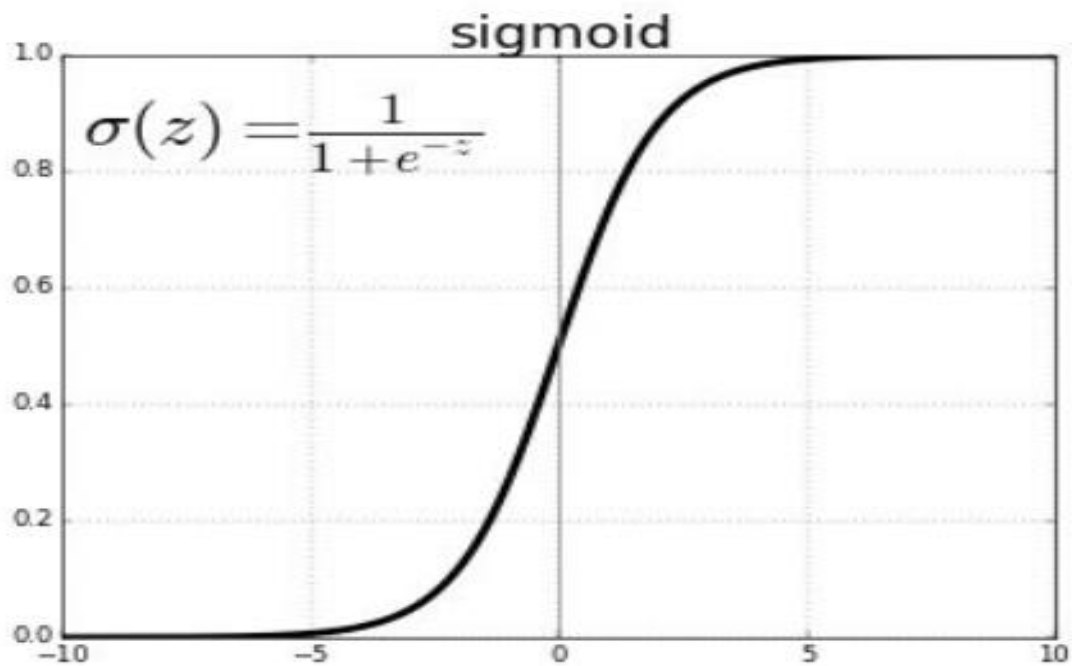
```
always @ (*) begin
```

```
y = $signed(sigmoid_func(x)) ;  
end
```

- **PYTHON CODE:**

```
# Testing Out Our Sigmoid Function  
import numpy as np  
def sigmoid(x):  
    return 1.0 / (1.0 + np.exp(-x))  
print(sigmoid(0.5))
```

CURVE OUTPUT:



2.TANH ACTIVATION FUNCTION:

TanH compress a real-valued number to the range [-1, 1]. It's non-linear, But it's different from Sigmoid, and its output is zero-centered. The main advantage of this is that the negative inputs will be mapped strongly to the negative and zero inputs will be mapped to almost zero in the graph of TanH. In tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

MATHEMATICAL FORM:

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

TanH Activation Function—Equation

Pros and Cons:

- TanH also has the vanishing gradient problem, but the gradient is stronger for TanH than sigmoid (derivatives are steeper).
- TanH is zero-centered, and gradients do not have to move in a specific direction.

CODE FOR TANH ACTIVATION FUNCTION:

• VERILOG CODE:

```
module tanh (
    input clk,
    input reset,
    input reg signed [31:0] x,
    input reg signed [31:0] y
);

//INTERNAL VARIABLES
reg signed [31:0] z;
real exp_val, exp_neg_val;

//TANH FUNCTION
function real tanh func;
    input real x;
    begin
        tanh func = ( exp(x) - exp(-x)) / (exp(x) + exp(-x));
    end
endfunction

always @ (posedge clk or posedge reset ) begin
    if (reset) begin
        y <= 0;

    end else begin
        z <= x;
        exp_val = $exp(z);
        exp_neg_val = $exp(-z);
        y <= $signed (tanh func (exp_val,exp_neg_val));
    end
end
endmodule
```

PYTHON CODE:

```
import numpy as np
import matplotlib.pyplot as plt

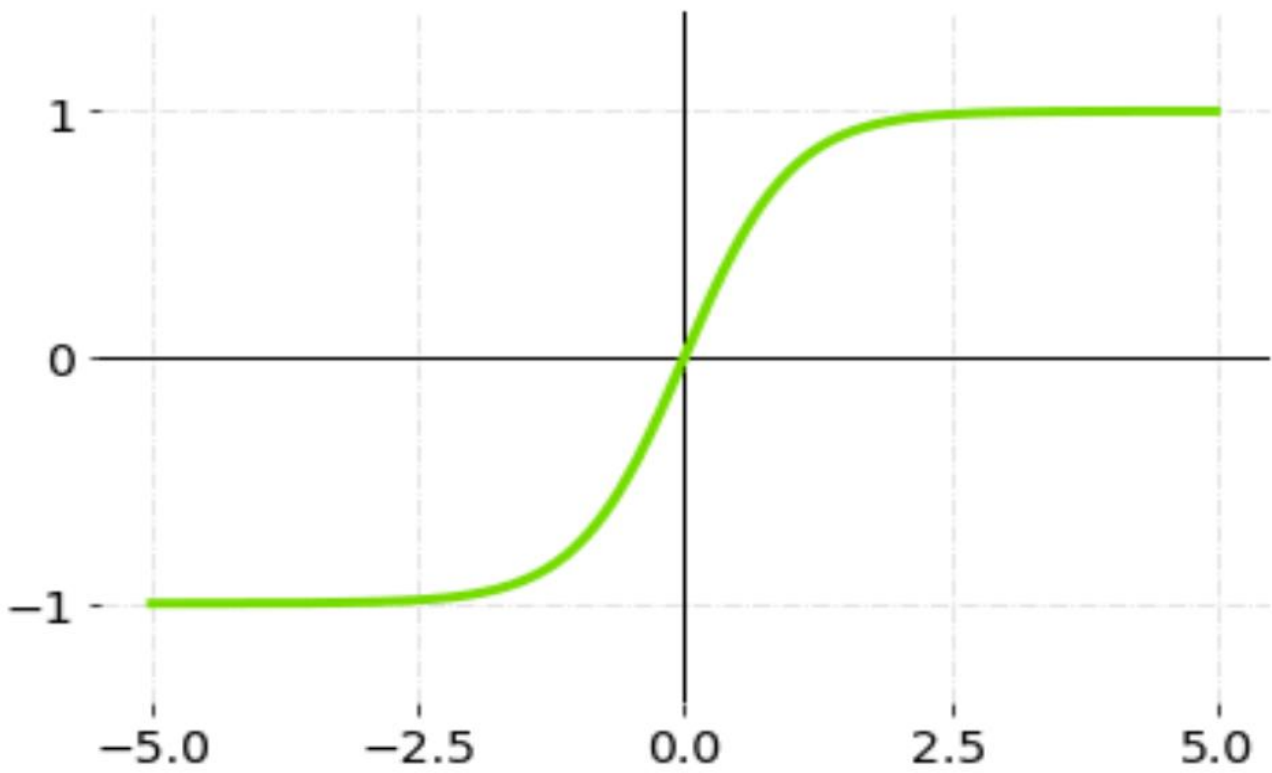
# Hyperbolic Tangent (htan) Activation Function
def htan(x):
    return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))

# htan derivative
def der_htan(x):
    return 1 - htan(x) * htan(x)

# Generating data for Graph
x_data = np.linspace(-6,6,100)
y_data = htan(x_data)
dy_data = der_htan(x_data)

# Graph
plt.plot(x_data, y_data, x_data, dy_data)
plt.title('htan Activation Function & Derivative')
plt.legend(['htan','der_htan'])
plt.grid()
plt.show()
```


OUTPUT CURVE:



TanH Activation Function—Graph

3.STEP FUNCTION:

Step Function is used as a “BINARY CLASSIFIER”.

MATHEMATICAL FORM:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

If the value of Y is above a certain value, declare it activated. If it's less than the threshold, then say it's not.

PROS:

- Great for binary classification.

CONS:

- The gradient of the step function is zero. This makes the step function not so useful since during back-propagation when the gradients of the activation functions are sent for error calculations to improve and optimize the results.
- It cannot be used for multi-class classification.

- **CODE FOR STEP ACTIVATION FUNCTION:**

- **VERILOG CODE:**

```
module step(
    input clk,
    input reset,
    input signed [31:0] x,
    output reg y
);

    //INTERNAL VARIABLES
    reg signed [31:0] z;

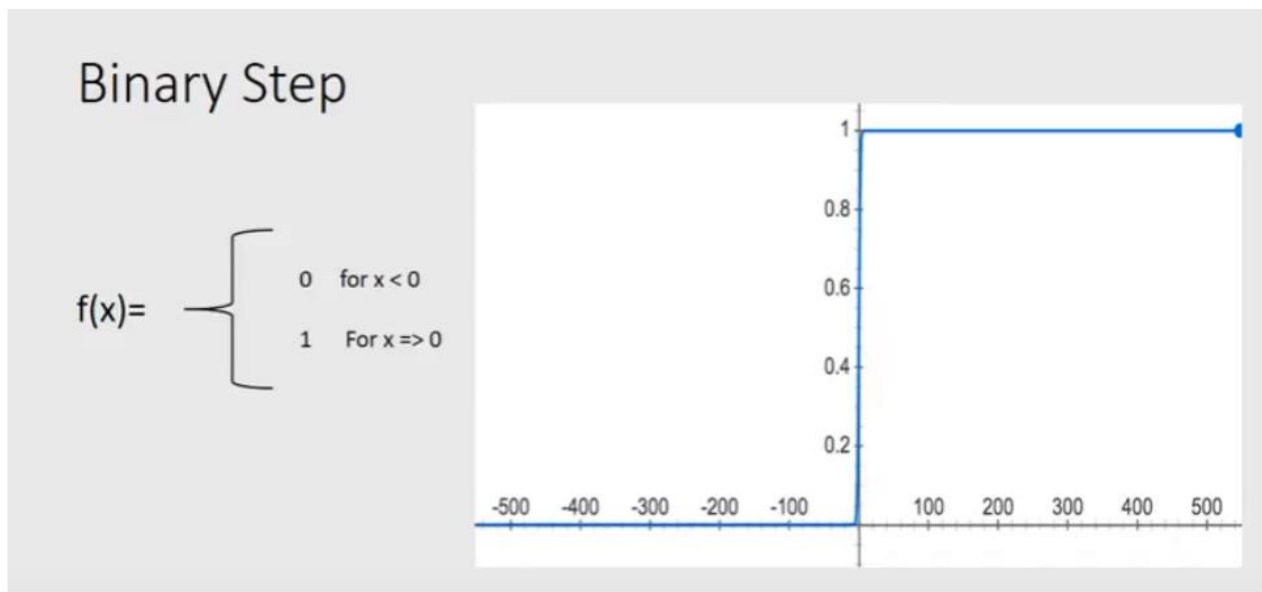
    //STEP FUNCTION
    function integer step func;
        input integer x;
        begin
            if ( x >= 0 ) begin
                step_func = 1;
            end else begin
                step_func = 0;
            end
        end
    endfunction

    always @ (posedge clk or posedge reset) begin
        if (reset) begin
            y <= 0;
        end else begin
            z <= x;
            y <= step_func(z);
        end
    end
endmodule
```

• PYTHON CODE:

```
def binary_step(x):  
    if x<0:  
        return 0  
    else:  
        return 1
```

OUTPUT CURVE



4. LINEAR ACTIVATION FUNCTION:

- The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input. Linear function has the equation similar to as of a straight line i.e. $y = x$
- No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.

Mathematically it can be represented as:

$$f(x) = x$$

PROS :

- It Is Used At Only One Place I.E. Output Layer. Takes The Inputs, Multiplies It By The Weights, And Creates An Output Signal Proportional To The Input.
- It Is Better Than A Step Function Because It Allows Multiple Outputs, Not Just Yes And No.

DISADVANTAGES :

- If We Try To Differentiate Linear Function To Bring Nonlinearity Then Our Result Will No More Depend On Input "X" And Function Will Become Constant And In Turn, Will Not Introduce Any Groundbreaking Behavior To Our Algorithm. So It Is Not Possible Use Backpropagation To Go Back And Understand Which Weights In The Input Neurons Can Provide A Better Prediction.
- It's not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x.
- All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

CODE FOR LINEAR ACTIVATION FUNCTION:

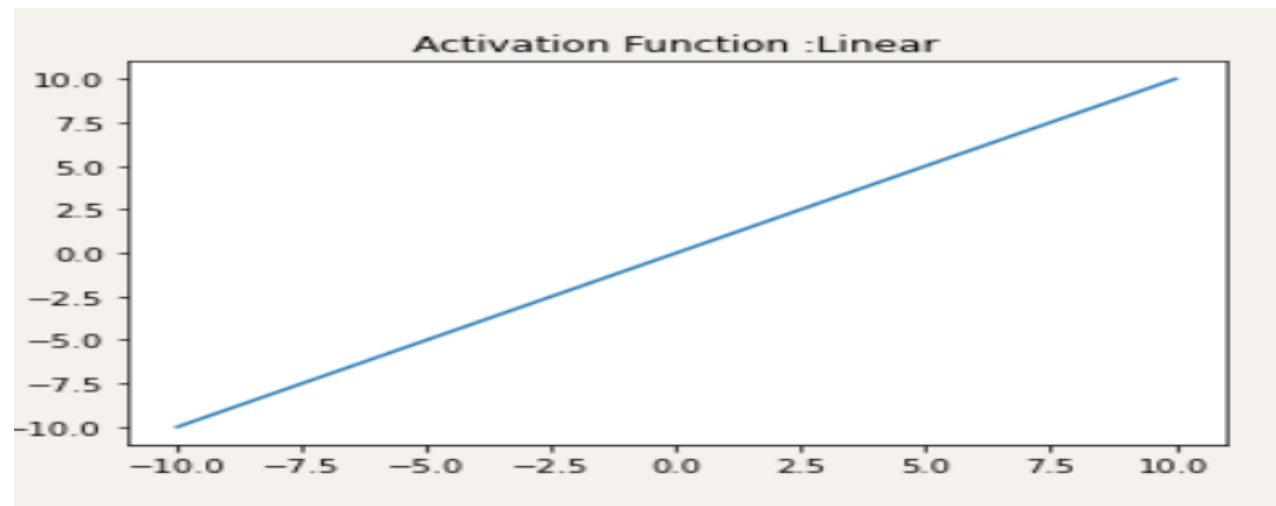
• VERILOG CODE:

```
module linear_activation(  
    input signed [31:0] x,  
    output reg signed [31:0] y  
);  
    always @ (posedge clk) begin  
        y <= x;  
    end  
endmodule
```

• PYTHON CODE:

```
def linear(x):  
    """ y = f(x) It returns the input as it is"""  
    return x  
x = np.linspace(-10, 10)  
plt.plot(x, linear(x))  
plt.title('Activation Function :Linear')  
plt.show()
```

OUTPUT CURVE:



5. RELU:

ReLU (Rectified Linear Unit) is a popular activation function that has been shown to be effective in deep learning applications. It is simple, computationally efficient, and has been shown to work well for image processing tasks. ReLU can be effective in identifying and highlighting important features of an image, which can be useful for evaluating eye diseases. ReLU activation functions are faster than sigmoid and tanh because it has no exponential function, that's why time consumption is less as compared to sigmoid and tanh.

WORKING OF RELU:

- If negative input is given, output will be 0.(For instance, if we give input -7,output will be 0).
- If positive input is given, output will be same value . For instance, if we give input 7,output will be 7).

MATHEMATICAL FORM:

$$\sigma(x) = \begin{cases} \max(0, x) & , x \geq 0 \\ 0 & , x < 0 \end{cases}$$

PROS:

- It avoids and rectifies vanishing gradient problem.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

CONS:

- One of its limitations is that it should only be used within hidden layers of a neural network model.

- Some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. In other words, ReLu can result in dead neurons.
- In another words, For activations in the region ($x < 0$) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called the dying ReLu problem.
- The range of ReLu is $[0, \infty)[0, \infty)$. This means it can blow up the activation.

Further reading

- [Deep Sparse Rectifier Neural Networks](#) Glorot et al., (2011)
- [Yes You Should Understand Backprop](#), Karpathy (2016)

CODE FOR RELU ACTIVATION FUNCTION:

• **VERILOG CODE:**

```

module relu_activation(
    input signed [31:0] x,
    output reg signed [31:0] y
);

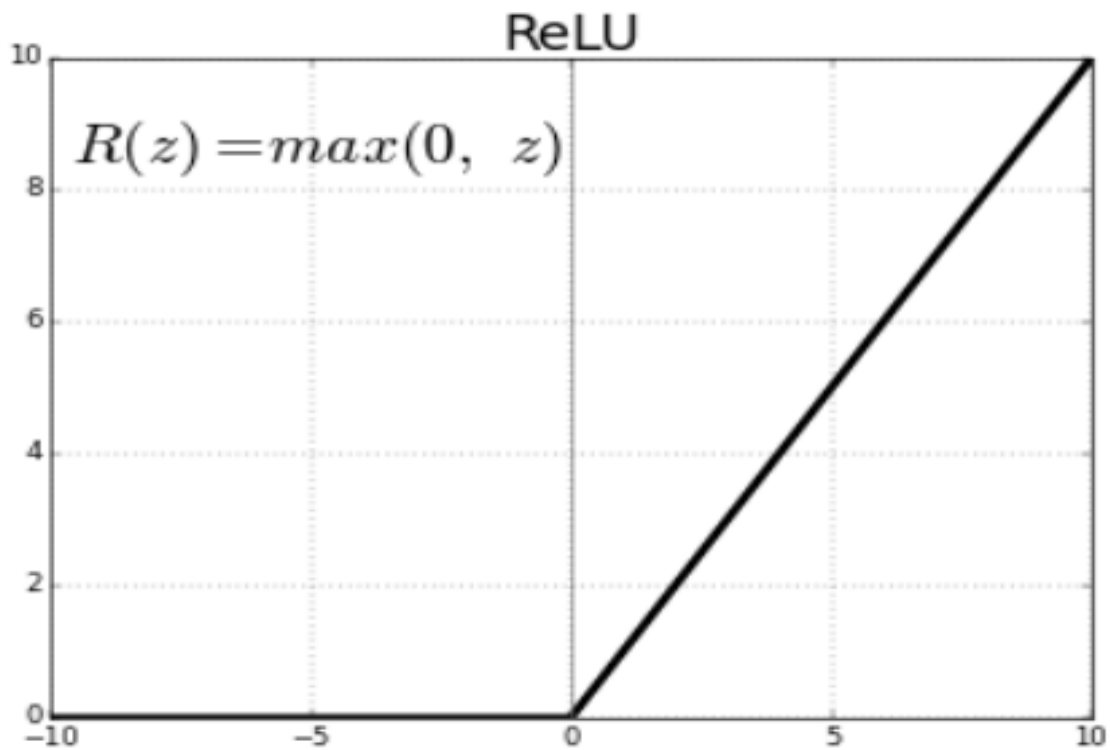
always @( posedge clk) begin
    if (x >= 0) begin
        y <= x;
    end
    else begin
        y <= 0;
    end
end
endmodule

```

PYTHON CODE:

```
def relu(x):  
  
    return(np.maximum(0, x))
```

OUTPUT GRAPH:



For fixing dead neurons problems(as for dead neurons it'll not be activated) , we can use LeakyReLU , PRELU, EXPONENTIAL RELU. When input is negative, output is always "0" which means it is not activated at that time.so this is called "DEAD NEURON." To solve this problem, we use "[Leaky ReLU](#)".

6. LeakyReLU:

Maas et al. [24] proposed LReLU. This function is similar compared to ReLU, except for allowing small, non-negative and constant gradient of input in other to reduce the dying neuron input problem. In other to solve the dying problem, LReLU introduced an alpha (α) parameter which is the leak, so that gradients will be small but not zero. This approach reduces the sparsity but tends to make the gradient more robust for optimization and on the other hand, there is an adjustable weight for the nodes that were not active with ReLU.

Leaky ReLU is a variation of ReLU that is designed to address some of the limitations of the standard ReLU function, which can cause neurons to "die" (i.e., stop learning). Leaky ReLU introduces a small slope for negative inputs, which can help to prevent neurons from dying and improve the performance of the neural network.

MATHEMATICAL FORM:

$$f(x, \alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

where $\alpha = 0.01$. It'll fire when x is positive and it'll also fire, when x is less than or equals to 0, then y is $0.0001(\text{input}(x))$.

PROS

- It fixes the “dying ReLU” problem, as it doesn’t have zero-slope parts.
- It speeds up training. There is evidence that having the “mean activation” be close to 0 makes training faster. (It helps keep off-diagonal entries of the Fisher information matrix small, but you can safely ignore this.) Unlike ReLU, leaky ReLU is more “balanced,” and may therefore learn faster.

CONS

- Unlike PReLU, the coefficient of x is predefined and it has alpha parameter to solve the problem of “Dying Neuron”.

CODE FOR LEAKYRELU ACTIVATION FUNCTION:

- **VERILOG CODE:**

```
module leaky_relu_activation(  
    input signed [31:0] x,  
    output reg signed [31:0] y  
);  
  
    parameter real alpha = 0.01; // leaky coefficient  
  
    always @(posedge clock) begin  
        if (x >= 0)  
            begin  
                y <= x;  
            end  
        else begin  
            y <= alpha * x;  
        end  
    end  
  
endmodule
```

PYTHON CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Leaky Rectified Linear Unit (leaky ReLU) Activation Function
def leaky_ReLU(x):
    data = [max(0.05*value,value) for value in x]
    return np.array(data, dtype=float)

# Derivative for leaky ReLU
def der_leaky_ReLU(x):
    data = [1 if value>0 else 0.05 for value in x]
    return np.array(data, dtype=float)

# Generating data For Graph
x_data = np.linspace(-10,10,100)
y_data = leaky_ReLU(x_data)
dy_data = der_leaky_ReLU(x_data)

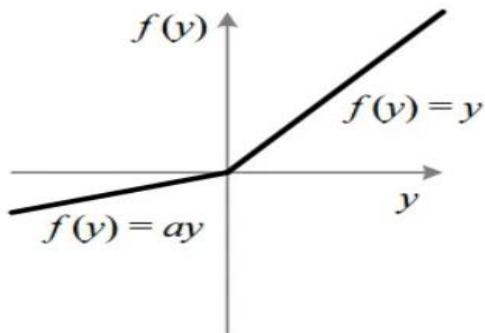
# Graph
plt.plot(x_data, y_data, x_data, dy_data)
plt.title('leaky ReLU Activation Function & Derivative')
plt.legend(['leaky_ReLU','der_leaky_ReLU'])
```

```
plt.grid()
plt.show()
```

FUNCTION OF LEAKY RELU:

```
def leaky_relu(x, alpha=0.01):
    """
    Leaky ReLU activation function.
    :param x: Input value.
    :param alpha: Slope of the negative part of the function (default 0.01).
    :return: Output value.
    """
    return max(alpha * x, x)
```

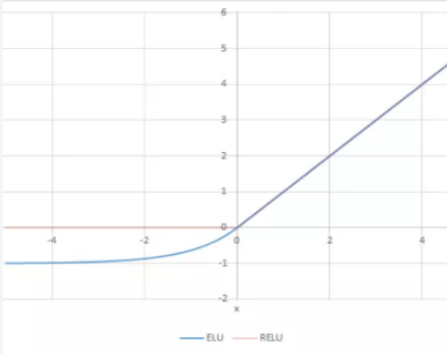
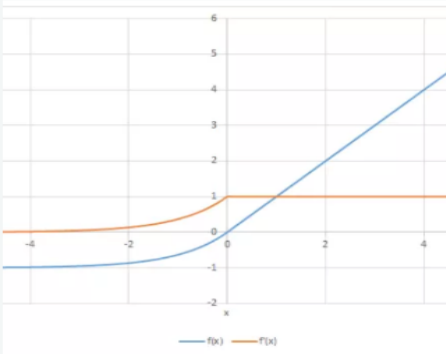
GRAPH OF LEAKYRELU:



's Leaky ReLU

7. EXPONENTIAL LINEAR UNIT (ELU):

Exponential Linear Unit or its widely known name ELU is a function that tends to converge cost to zero faster and produce more accurate results. Different to other activation functions, ELU has an extra alpha constant which should be a positive number. ELU is very similar to RELU except for negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equals $-\alpha$ whereas RELU sharply smooths.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z < 0 \end{cases}$
	
<pre>def elu(z,alpha): return z if z >= 0 else alpha*(e^z - 1)</pre>	<pre>def elu_prime(z,alpha): return 1 if z > 0 else alpha*np.exp(z)</pre>

PROS:

- ELU becomes smooth slowly until its output equals $-\alpha$ whereas RELU sharply smooths.
- ELU is a strong alternative to ReLU.
- Unlike to ReLU, ELU can produce negative outputs.

CONS:

- For $x > 0$, it can blow up the activation with the output range of $[0, \infty]$.

Unlike the leaky relu and parametric ReLU functions, instead of a straight line, ELU uses a log curve for defining the negative values.

CODE FOR EXPONENTIAL LINEAR UNIT ACTIVATION FUNCTION:

- **VERILOG CODE:**

```
module elu_activation(  
    input signed [31:0] x,  
    output reg signed [31:0] y  
);  
  
    parameter real alpha = 1.0; // scaling factor for negative inputs  
  
    real exp_x;  
  
    always @(x)  
    begin  
        if (x >= 0)  
            begin  
                y <= x;  
            end  
        else  
            begin  
                exp_x = exp(x);  
                y <= alpha * (exp_x - 1);  
            end  
    end  
end  
endmodule
```


PYTHON CODE:

```
import numpy as np
```

```
def elu(x, alpha=1.0):
```

```
    """
```

```
    Exponential Linear Unit (ELU) activation function.
```

```
    :param x: Input value.
```

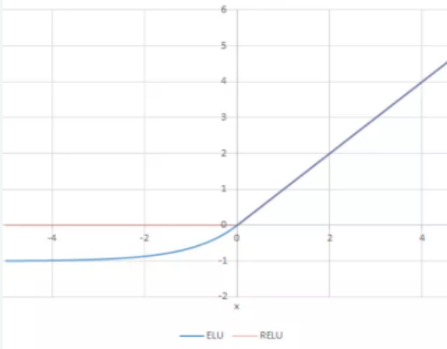
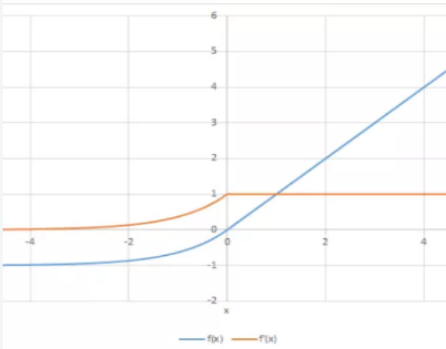
```
    :param alpha: Scaling factor for negative inputs (default 1.0).
```

```
    :return: Output value.
```

```
    """
```

```
    return np.where(x >= 0, x, alpha * (np.exp(x) - 1))
```

GRAPH:

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z < 0 \end{cases}$
	
<pre>def elu(z, alpha): return z if z >= 0 else alpha*(e^z - 1)</pre>	<pre>def elu_prime(z, alpha): return 1 if z > 0 else alpha*np.exp(z)</pre>

8.SWISH ACTIVATION FUNCTION:

Swish is a lesser known activation function which was discovered by researchers at Google. Swish is as computationally efficient as ReLU and shows better performance than ReLU on deeper models. The values for swish ranges from negative infinity to infinity. As you can see, the curve of the function is smooth and the function is differentiable at all points. This is helpful during the model optimization process and is considered to be one of the reasons that swish outperforms ReLU. A unique fact about this function is that swish function is not monotonic. This means that the value of the function may decrease even when the input values are increasing.

MATHEMATICAL FORM:

$$f(x) = x \cdot \sigma(x) \text{ -----(1)}$$

where $\sigma(x) = (1 + \exp(-x))^{-1}$ is the sigmoid function.

PROS

- Swish consistently outperforms or matches the ReLU function on a variety of deep models.
 - Unboundedness is desirable because it avoids saturation, where training is slow due to near-zero gradients.
 - Smoothness plays a beneficial role in optimization and generalization which swish function provides.
 - BEST WHEN THERE ARE MORE THAN 40 LAYERS.
 - USED FOR MULTICLASS CLASSIFICATION.
 - FOR LAYERS LESS THAN 40, IT WON'T WORK.
 - SOLVE DEAD NEURON PROBLEM
-

CONS

- The function is time-intensive to compute for deeper layers with large parameter dimensions.
- Lower sums of weighted inputs which would want to increase their output would not work because the function has a non-monotonic region for negative values closer to zero.

CODE FOR SWISH ACTIVATION FUNCTION:

- **VERILOG CODE:**

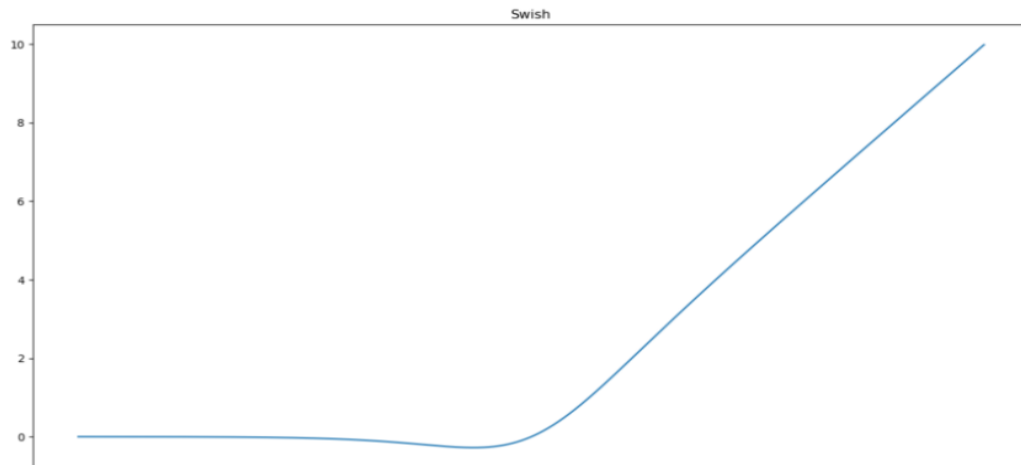
```
module swish(  
    input [N-1:0] x,  
    input [M-1:0] w,  
    output reg [M-1:0] y  
);  
  
    parameter alpha = 1.0; // Swish scaling factor  
  
    reg [N-1:0] x_scaled;  
  
    always @(*) begin  
        x_scaled = alpha * x;  
        y = x_scaled / (1 + $exp(-w * x_scaled));  
    end  
  
endmodule
```

where swish function takes “x” as an input vector of size N, weight matrix of size “MxN” and produces vector “y” of size “M”. Alpha specifies scaling factor for input values.

PYTHON CODE:

```
def swish_function(x):  
    return x/(1-np.exp(-x))  
swish_function(-67), swish_function(4)
```

GRAPH:



9.SOFTMAX ACTIVATION FUCTION:

Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

Building a network for a multiclass problem, the output layer would have as many neurons as the number of classes in the target. For instance if you have three classes, there would be three neurons in the output layer. Suppose you got the output from the neurons as [1.2 , 0.9 , 0.75].

Applying the softmax function over these values, you will get the following result – [0.42 , 0.31, 0.27]. These represent the probability for the data point belonging to each class. Note that the sum of all the values is 1.

PROS:

- Used for tasks such as multiclassification.
- Helps to ensure that the outputs sum to 1, which is useful for some applications where the output should represent a proportion or likelihood.

CONS:

- Softmax is sensitive to outliers and can produce large errors when the correct class is very different from the others.
- The softmax function is computationally expensive, especially when the number of output classes is large.
- Softmax tends to produce very small gradients when the output is far from the correct class, which can make training slow or difficult.

CODE FOR SOFTMAX ACTIVATION FUNCTION:

VERILOG CODE:

```

module softmax(
    input [N-1:0] x,
    output reg [N-1:0] y
);

reg [N-1:0] exp_x;
reg [N-1:0] sum_exp_x;

always @(*) begin
    // Compute the exponentiation of the input vector
    for (int i = 0; i < N; i++) begin
        exp_x[i] = $exp(x[i]);
    end

    // Compute the sum of the exponentiated values
    sum_exp_x = 0;
    for (int i = 0; i < N; i++) begin
        sum_exp_x += exp_x[i];
    end

    // Compute the softmax output
    for (int i = 0; i < N; i++) begin
        y[i] = exp_x[i] / sum_exp_x;
    end
end

endmodule

```

PYTHON CODE:

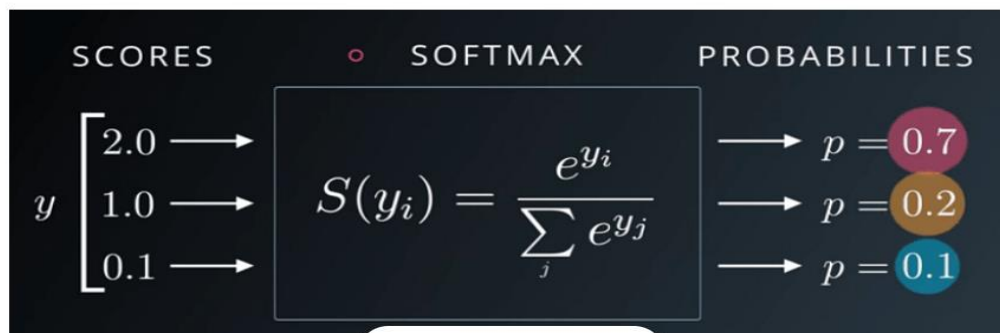
```

def softmax_function(x):
    z = np.exp(x)
    z_ = z/z.sum()
    return z_
softmax_function([0.8, 1.2, 3.1])

```

PICTORIAL REPRESENTATION OF SOFTMAX:

The softmax function would squeeze the outputs for each class between 0 and 1 and the sum of the outputs is always 1.



10.MAXOUT:

The Maxout activation is a generalization of the ReLU and the leaky ReLU functions. It is a piecewise linear function that returns the maximum of inputs, designed to be used in conjunction with the dropout regularization technique. Both ReLU and leaky ReLU are special cases of Maxout. The Maxout neuron, therefore, enjoys all the benefits of a ReLU unit and does not have any drawbacks like dying ReLU. However, it doubles the total number of parameters for each neuron, and hence, a higher total number of parameters need to be trained.

MATHEMATICAL FORM:

$$h(x) = \max (Z_1, Z_2, \dots, Z_n)$$

$$h(x) = \max (W_1 \cdot x + b_1, W_2 \cdot x + b_2, \dots, W_n \cdot x + b_n)$$

PROS

1. **Non-linearity:** Like other activation functions, Maxout introduces non-linearity into the network, which enables it to learn complex and non-linear relationships between the inputs and outputs.
2. **Sparsity:** Maxout can help induce sparsity in the network by only activating the output of the maximum element in each group of neurons. This can reduce the number of active neurons and the computational cost of the network.
3. **Flexibility:** Maxout is a very flexible activation function that can learn any convex piecewise linear function. It can be used with any number of linear pieces, which allows it to fit a wide variety of input-output mappings.

CONS

1. **Parameter Overhead:** Maxout requires more parameters than other activation functions, as it needs to learn the weights and biases of the linear pieces in addition to the weights of the connections between the layers. This can make the training process more difficult and computationally expensive.
2. **Sensitivity to Initialization:** Maxout is sensitive to weight initialization, and the optimal initialization can be difficult to find. If the weights are initialized poorly, the network can get stuck in a local minimum or take a long time to converge.
3. **Not Widely Used:** Despite its advantages, Maxout has not been widely adopted in practice, and ReLU and its variants remain the most commonly used activation functions. This is partly due to the complexity and computational cost of the function, as well as the difficulty of tuning the parameters.

CODE FOR MAXOUT ACTIVATION FUNCTION:

- **VERILOG CODE:**


```

module maxout(
  input [N-1:0] x,
  input [M-1:0] w,
  output reg [M-1:0] y
);

  parameter K = 2; // Number of linear pieces

  reg [K-1:0] z;

  always @(*) begin
    for (int i = 0; i < K; i = i + 1) begin
      z[i] = w[i*N +: N] * x;
    end
    y = $max(z);
  end

endmodule

```

where x takes an input vector of size 'N' a weight matrix 'w' of size 'MxN' and produces output vector 'y' of size 'M'.

PYTHON CODE:

```

import numpy as np

def maxout(x, w, k=2):
    z = np.dot(w, x)
    z = z.reshape((-1, k))
    return np.max(z, axis=1)

```

11.PARAMETRIC RELU:

A **Parametric Rectified Linear Unit**, or **PReLU**, is an activation function that generalizes the traditional rectified unit with a slope for negative values. The Parametric ReLU (PReLU) is an activation function commonly used in deep learning neural networks. It is a variant of ReLU activation function, which has become popular due to its simplicity and effectiveness in many types of neural networks.

MATHEMATICAL FORM:

$$f(y_i) = \begin{cases} y_i, & \text{if } y_i > 0 \\ a_i y_i, & \text{if } y_i \leq 0 \end{cases}$$

Mathematical definition of PReLU

Above, y_i is any input on the i th channel and a_i is the negative slope which is a learnable parameter.

- if $a_i=0$, f becomes ReLU
- if $a_i>0$, f becomes leaky ReLU
- if a_i is a learnable parameter, f becomes PReLU

PROS

1. **Non-zero gradient for negative input values:** Unlike the traditional ReLU function, which has zero gradient for negative input values, PReLU has a non-zero gradient for negative values. This can help address the issue of "dying ReLU" in which neurons with negative input values can become permanently inactive.
2. **Flexibility:** PReLU introduces a learnable parameter, which allows the neural network to adapt the activation function to the specific needs of the data. This can result in improved performance over the traditional ReLU function.
3. **Reduced overfitting:** The PReLU activation function has been shown to reduce overfitting in deep neural networks, particularly when the training data is limited. By introducing a learnable parameter, PReLU can help the neural network better generalize to new data.
4. **Better performance:** In some cases, PReLU has been shown to outperform traditional ReLU, particularly in networks with multiple layers.

CONS

1. Increased computational complexity: PReLU introduces a learnable parameter for each neuron in the network, which increases the number of parameters that need to be learned during training. This can lead to increased computational complexity and longer training times.
2. Increased risk of overfitting: While PReLU has been shown to reduce overfitting in some cases, the introduction of a learnable parameter for each neuron can also increase the risk of overfitting. This is especially true in networks with a large number of parameters.
3. Difficulty in interpreting the learned parameters: Unlike some other activation functions, such as the sigmoid or tanh functions, it can be difficult to interpret the learned parameters in PReLU. This can make it harder to gain insights into how the network is processing the data.
4. Sensitivity to initialization: The performance of PReLU can be sensitive to the initialization of the learnable parameters. If the parameters are not initialized properly, the network may fail to converge or exhibit poor performance.

CODE FOR PReLU ACTIVATION FUNCTION:

- **VERILOG CODE:**

```
module PReLU(  
    input wire clk,  
    input wire [DATA_WIDTH-1:0] x,  
    output reg [DATA_WIDTH-1:0] y,  
    input wire [DATA_WIDTH-1:0] alpha  
);  
  
    reg [DATA_WIDTH-1:0] pos_x; // positive input values  
    reg [DATA_WIDTH-1:0] neg_x; // negative input values  
  
    always @(posedge clk) begin  
        if (x >= 0)  
            begin
```

```

        pos_x <= x;
        neg_x <= 0;
    end
else
    begin
        pos_x <= 0;
        neg_x <= x;
    end
end
end

always @(posedge clk) begin
    if (x >= 0)
        begin
            y <= pos_x;
        end
    else
        begin
            y <= alpha * neg_x;
        end
    end
end
endmodule

```

PYTHON CODE:

```

import numpy as np

class PReLU:
    def __init__(self, alpha=0.01):
        self.alpha = alpha

    def forward(self, x):
        self.mask = x > 0
        out = np.copy(x)
        out[~self.mask] *= self.alpha
        return out

    def backward(self, dout):
        dx = np.copy(dout)

```

```
dx[~self.mask] *= self.alpha  
return dx
```

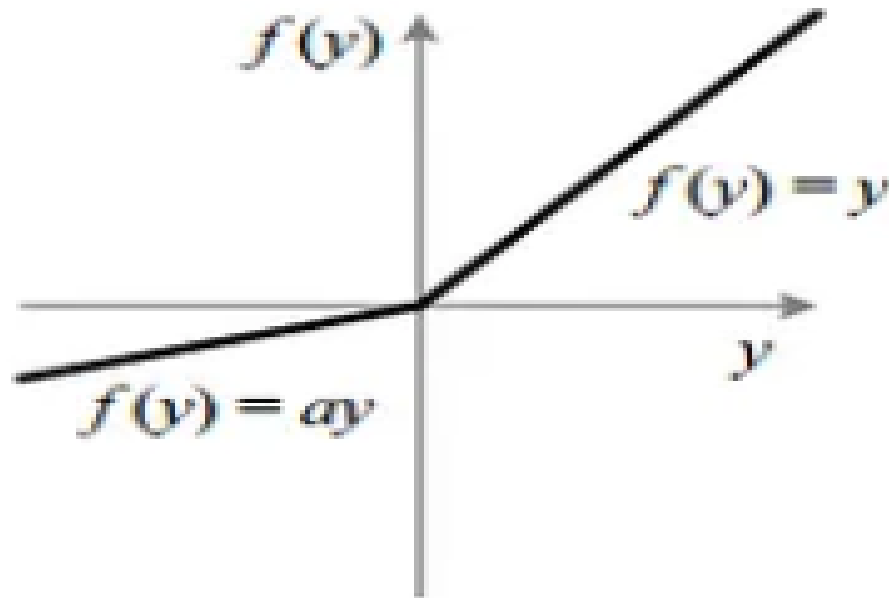
In this implementation, the PReLU function takes in an input array `x` and a learnable parameter `alpha`. The output is the PReLU activation function value.

The `forward` method computes the PReLU activation function value by applying the `alpha` parameter to the negative input values, and keeping the positive input values unchanged. The `mask` attribute is used to keep track of which values are positive and which are negative.

The `backward` method computes the gradient of the PReLU activation function. It multiplies the gradient `dout` by `alpha` for negative input values, and leaves the gradient unchanged for positive input values.

Note that this implementation assumes that `x` is a numpy array, and uses the `numpy.copy()` method to create a copy of the input array.

GRAPH:



12. SOFTPLUS:

Softplus function is similar to the ReLU activation function, but has a smoother gradient and output. It is often used in the output layer of neural networks for binary classification problems, where it can be used to model the probability of a positive class. The Softplus activation function is a smooth and continuously differentiable activation function that is often used in neural networks.

MATHEMATICAL FORM:

$$f(x) = \log_2(1 + e^{**x}) \text{-----}(2)$$

PROS:

- The significant role of SoftPlus compared with Sigmoid and ReLU to achieve fast convergence with fewer epochs during training.
- Non-zero gradient for all inputs: Unlike the ReLU activation function, the Softplus function has a non-zero gradient for all inputs, ensuring that neurons continue to learn and adapt even when the input is negative. This avoids the "dying ReLU" problem and makes the Softplus function more robust to noisy inputs.
- Smooth and continuous: The Softplus function is a smooth and continuous function, which means that it can be easily differentiated and optimized using gradient-based methods. This is important for training deep neural networks, where gradient descent is often used to optimize the weights of the network.
- Captures non-linearity: The Softplus function is a non-linear activation function that allows neural networks to capture non-linear relationships in the data. This is important for modeling complex patterns in the data, such as those found in natural language processing, image recognition, and speech recognition.
- Computationally efficient: The Softplus function is computationally efficient to evaluate, which makes it a good choice for large-scale neural network architectures that require fast training and inference.

CONS

1. Sensitive to initialization: The Softplus function can be sensitive to initialization, which can make it difficult to train neural networks that use this activation function. Improper initialization can result in neurons getting stuck in local minima or producing unstable gradients.
2. Saturation: The Softplus function can suffer from saturation when the input is large, which can lead to vanishing gradients and slow learning. This can be a problem in deep neural network architectures, where the gradient signal can become very weak after passing through multiple layers.
3. Output range: The Softplus function outputs values between 0 and infinity, which can be problematic for some types of neural networks, such as those used for classification tasks, where a probability distribution over classes is needed. In these cases, the output range can be normalized using additional layers or activation functions.
4. Non-monotonicity: The Softplus function is a non-monotonic function, meaning that it is not strictly increasing or decreasing over its entire domain. This can make it more difficult to reason about the behavior of neural networks that use this activation function.

CODE FOR SOFTPLUS ACTIVATION FUNCTION:

- **VERILOG CODE:**

```
module softplus (input [31:0] x, output reg [31:0] y);
```

```
    always @*
    begin
        if (x > 0)
            y = x + log(1 + exp(-x));
        else
            y = log(1 + exp(x));
    end
```

```
endmodule
```

- **PYTHON CODE:**

```
import numpy as np
```

```
def softplus(x):  
    return np.log(1 + np.exp(x))
```

where x is a Numpy array.

GRAPH:

