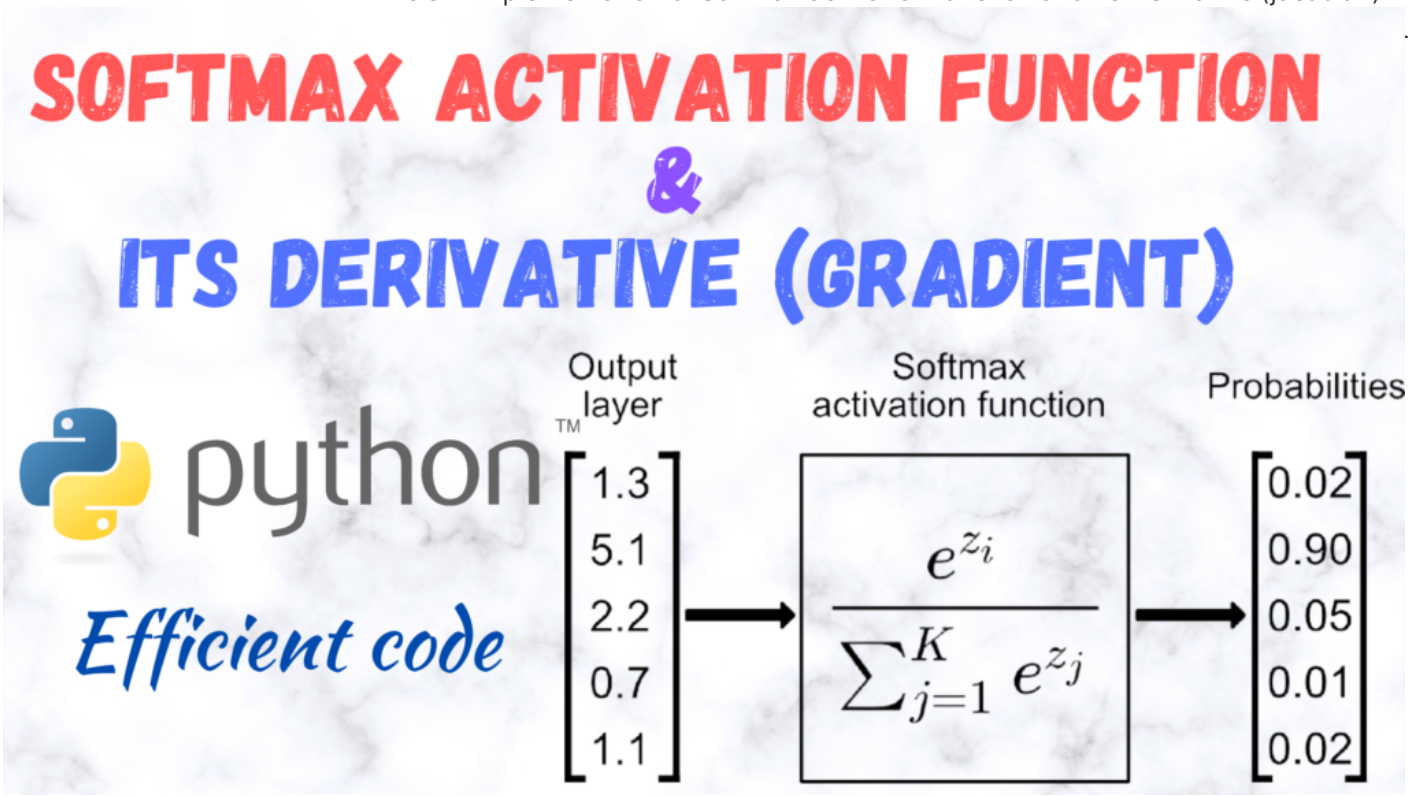
Efficient implementation of Softmax activation function and its Derivative (jacobian) in



The mathematical definition of the Softmax activation function is

$$S\left(z_i\right) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

with the derivative defined as

$$\frac{\partial S(z_i)}{\partial z_j} = \begin{cases} S\left(z_i\right) \times \left(1 - S\left(z_i\right)\right) & \text{if } i = j \\ -S\left(z_i\right) \times S\left(z_j\right) & \text{if } i \neq j \end{cases}$$

The Softmax function and its derivative for a batch of inputs (a 2D array with nRows=nSamples and nColumns=nNodes) can be implemented in the following manner:
Softmax simplest implementation

```
import numpy as np
def Softmax(x):
    '''
    Performs the softmax activation on a given set of inputs
    Input: x (N,k) ndarray (N: no. of samples, k: no. of nodes)
    Returns:
    Note: Works for 2D arrays only(rows for samples, columns for nodes/outputs)
    '''
    max_x = np.amax(x, 1).reshape(x.shape[0],1) # Get the row-wise maximum
    e_x = np.exp(x - max_x ) # For stability
    return e_x / e_x.sum(axis=1, keepdims=True)
```

Softmax gradient (technically jacobian) simplest implementation

```
import numpy as np
def Softmax_grad(x): # Best implementation (VERY FAST)
    '''Returns the jacobian of the Softmax function for the given set of inputs.
    Inputs:
    x: should be a 2d array where the rows correspond to the samples
        and the columns correspond to the nodes.
    Returns: jacobian
    '''
    s = Softmax(x)
    a = np.eye(s.shape[-1])
    temp1 = np.zeros((s.shape[0], s.shape[1], s.shape[1]),dtype=np.float32)
    temp2 = np.zeros((s.shape[0], s.shape[1], s.shape[1]),dtype=np.float32)
    temp1 = np.einsum('ij,jk->ijk',s,a)
    temp2 = np.einsum('ij,ik->ijk',s,s)
    return temp1-temp2
```

Please note, and I can't stress this enough, the above and the following implementations are only tested and fine-tuned for a batch of inputs, i.e., the expected input for the functions is a 2d array with rows representation different samples, and columns representing different nodes.

However, these implementations can be further accelerated (sped-up) by using Numba (https://numba.pydata.org/). Numba is a Just-in-time (JIT) compiler that

> translates a subset of Python and NumPy code into fast machine code.

To use numba, install it as:

```
pip install numba
```

Also, make sure that your numpy is compatible with Numba or not, although usually pip takes care of that. You can get the info here: https://pypi.org/project/numba/

Accelerating the above functions using Numba is quite simple. Just modify them in the following manner:

Softmax NUMBA implementation

```
from numba import njit
@njit(cache=True,fastmath=True)  # Best implementation (VERY FAST)
def Softmax(x):
    '''
    Performs the softmax activation on a given set of inputs
    Input: x (N,k) ndarray (N: no. of samples, k: no. of nodes)
    Returns:
    Note: Works for 2D arrays only(rows for samples, columns for nodes/outputs)
    '''
    max_x = np.zeros((x.shape[0],1),dtype=x.dtype)
    for i in range(x.shape[0]):
        max_x[i,0] = np.max(x[i,:])
    e_x = np.exp(x - max_x)
    return e_x / e_x.sum(axis=1).reshape((-1, 1)) # Alternative of keepdims=True for
Numba compatibility
```

Softmax derivative (jacobian) NUMBA implementation

```
from numba import njit
@njit(cache=True,fastmath=True)
def Softmax_grad(x): # Best implementation (VERY FAST)
    '''Returns the jacobian of the Softmax function for the given set of inputs.
    Inputs:
    x: should be a 2d array where the rows correspond to the samples
        and the columns correspond to the nodes.
    Returns: jacobian
    '''
    s = Softmax(x)
    a = np.eye(s.shape[-1])
    temp1 = np.zeros((s.shape[0], s.shape[1], s.shape[1]),dtype=np.float32)
    temp2 = np.zeros((s.shape[0], s.shape[1], s.shape[1]),dtype=np.float32)
    # Einsum is unsupported with Numba (nopython mode)
    # temp1 = np.einsum('ij,jk->ijk',s,a)
    # temp2 = np.einsum('ij,ik->ijk',s,s)
    for i in range(s.shape[0]):
        for j in range(s.shape[1]):
            for k in range(s.shape[1]):
                temp1[i,j,k] = s[i,j]*a[j,k]
                temp2[i,j,k] = s[i,j]*s[i,k]
    return temp1-temp2
```

This is quite fast and competitive with Tensorflow and PyTorch (https://github.com/manassharma07/crysx_nn/blob/main/benchmarks_tests/Performance_Activation_Functions_CPU.ipynb).

It is in fact also used in the CrysX-Neural Network library (crysx_nn)

Furthermore, the above implementations can be further accelerated using Cupy (CUDA), if using single precision (float32) is not a problem.

> CuPy is an open-source array library for GPU-accelerated computing with Python. CuPy utilizes CUDA Toolkit libraries to make full use of the GPU architecture.

The Cupy implementations look as follows:

```
import cupy as cp
def Softmax_cupy(x):
    '''
    Performs the softmax activation on a given set of inputs
    Input: x (N,k) ndarray (N: no. of samples, k: no. of nodes)
    Returns:
    Note: Works for 2D arrays only(rows for samples, columns for nodes/outputs)
    '''
    max_x = cp.amax(x, 1).reshape(x.shape[0],1)
    e_x = cp.exp(x - max_x) # For stability as it is prone to overflow and underflow
#     return e_x / e_x.sum(axis=1, keepdims=True) # Alternative 1
    return e_x / e_x.sum(axis=1).reshape((-1, 1)) # Alternative 2
```

```
def Softmax_grad_cupy(x): # Best implementation (VERY FAST)
    '''Returns the jacobian of the Softmax function for the given set of inputs.
    Inputs:
    x: should be a 2d array where the rows correspond to the samples
        and the columns correspond to the nodes.
    Returns: jacobian
    '''
    s = Softmax_cupy(x)
    a = cp.eye(s.shape[-1])
    temp1 = cp.zeros((s.shape[0], s.shape[1], s.shape[1]),dtype=cp.float32)
    temp2 = cp.zeros((s.shape[0], s.shape[1], s.shape[1]),dtype=cp.float32)
    temp1 = cp.einsum('ij,jk->ijk',s,a)
    temp2 = cp.einsum('ij,ik->ijk',s,s)
    return temp1-temp2
```

The above code is also used in the crysx_nn library.
To see how the crysx_nn implementations of Softmax compare with TensorFlow and PyTorch, click here.

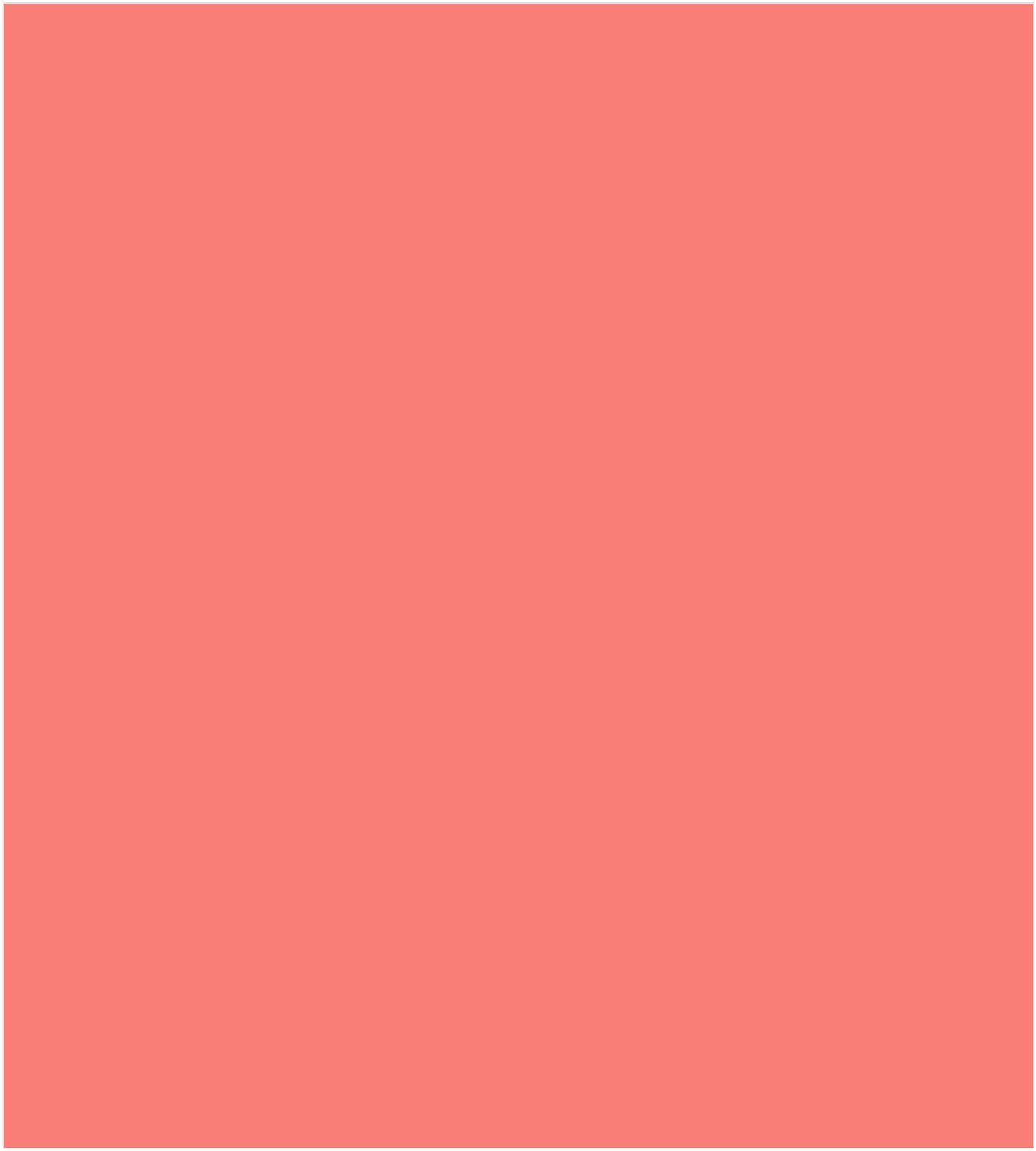
I hope you found this information useful.

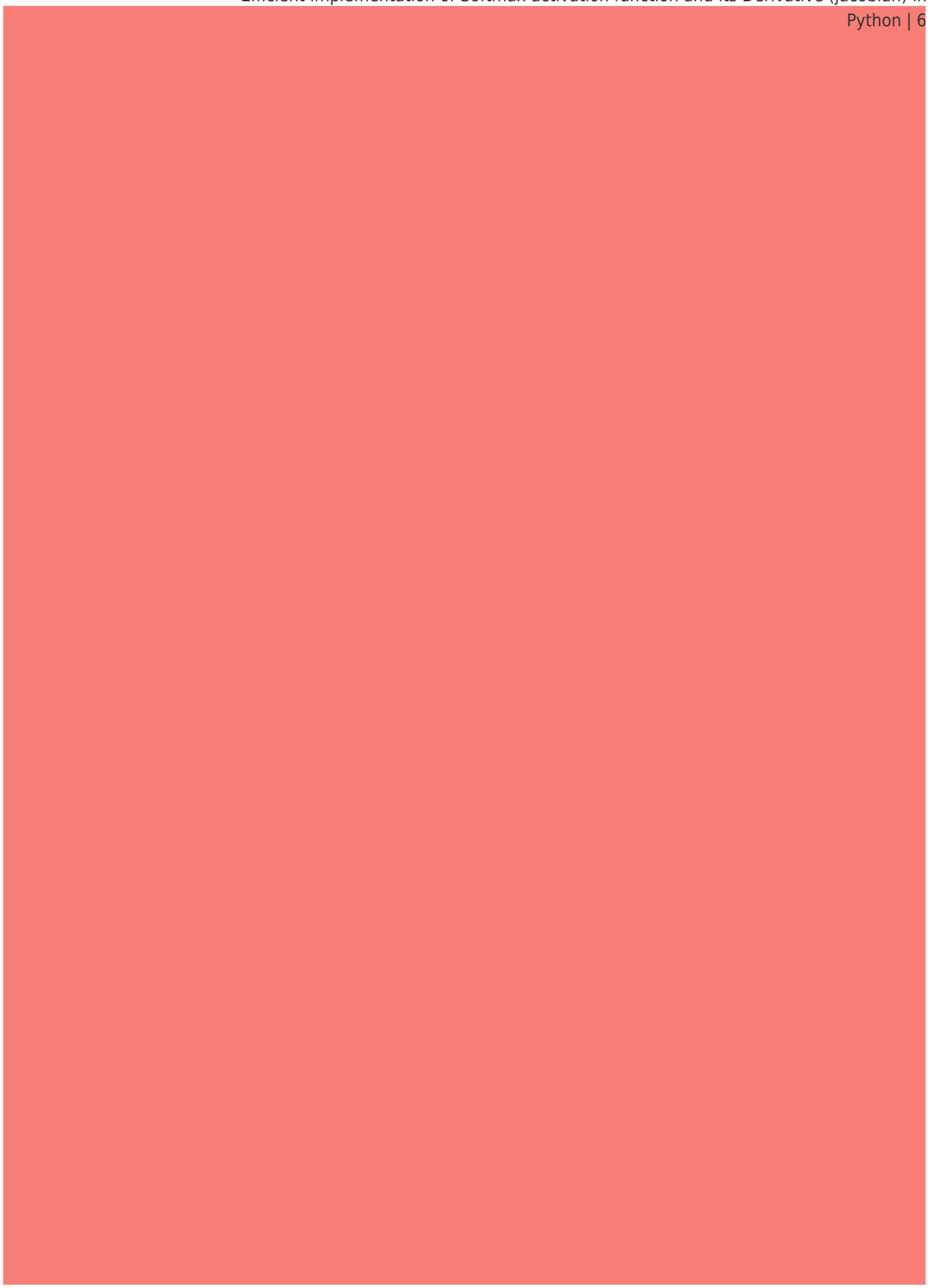
If you did, then don't forget to check out my other posts on Machine Learning and efficient implementations of activation/loss functions in Python.

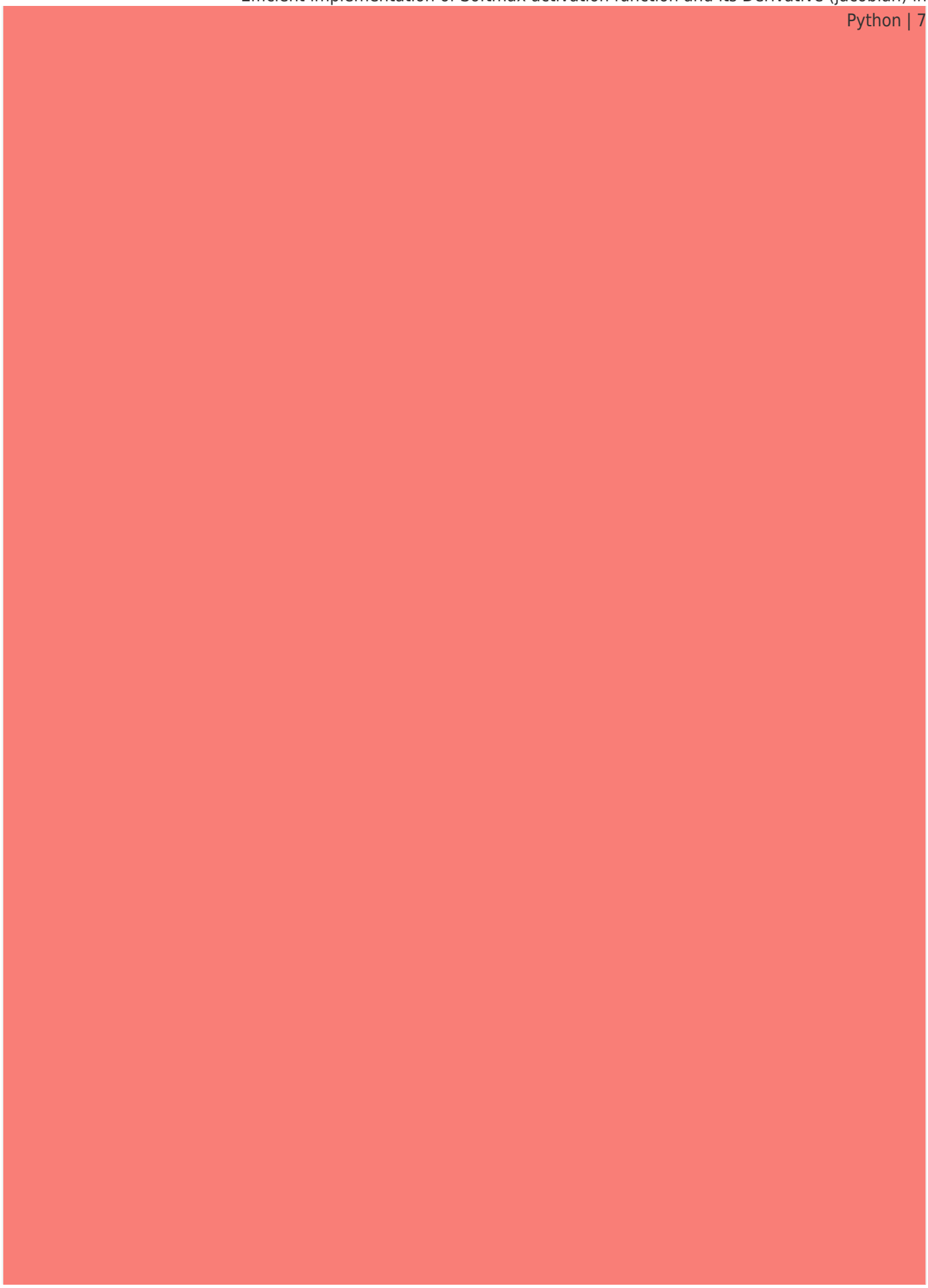
Manas Sharma

I’m a physicist specializing in computational material science with a PhD in Physics from Friedrich-Schiller University Jena, Germany. I write efficient codes for simulating light-matter interactions at atomic scales. I like to develop Physics, DFT, and Machine Learning related apps and software from time to time. Can code in most of the popular languages. I like to share my knowledge in Physics and applications using this Blog and a YouTube channel.

manas.bragitoff.com/







Share this:

Click to share on Facebook (Opens in new window)

Click to share on Twitter (Opens in new window)

Click to share on WhatsApp (Opens in new window)

Click to share on Pinterest (Opens in new window)

Click to share on Reddit (Opens in new window)

Click to share on LinkedIn (Opens in new window)

Click to email a link to a friend (Opens in new window)

Click to print (Opens in new window)

Click to share on Tumblr (Opens in new window)

Click to share on Pocket (Opens in new window)

Click to share on Telegram (Opens in new window)

[wpedon id="7041" align="center"]