1. **Write the Python code to implement a single neuron.**

**# Python program to implement a**

**# single neuron neural network**

**# import all necessery libraries**

**from numpy import exp, array, random, dot, tanh**

**# Class to create a neural**

**# network with single neuron**

**class NeuralNetwork():**

**def \_\_init\_\_(self):**

**# Using seed to make sure it'll**

**# generate same weights in every run**

**random.seed(1)**

**# 3x1 Weight matrix**

**self.weight\_matrix = 2 \* random.random((3, 1)) - 1**

**# tanh as activation function**

**def tanh(self, x):**

**return tanh(x)**

**# derivative of tanh function.**

**# Needed to calculate the gradients.**

**def tanh\_derivative(self, x):**

**return 1.0 - tanh(x) \*\* 2**

**# forward propagation**

**def forward\_propagation(self, inputs):**

**return self.tanh(dot(inputs, self.weight\_matrix))**

**# training the neural network.**

**def train(self, train\_inputs, train\_outputs,**

**num\_train\_iterations):**

**# Number of iterations we want to**

**# perform for this set of input.**

**for iteration in range(num\_train\_iterations):**

**output = self.forward\_propagation(train\_inputs)**

**# Calculate the error in the output.**

**error = train\_outputs - output**

**# multiply the error by input and then**

**# by gradient of tanh funtion to calculate**

**# the adjustment needs to be made in weights**

**adjustment = dot(train\_inputs.T, error \***

**self.tanh\_derivative(output))**

**# Adjust the weight matrix**

**self.weight\_matrix += adjustment**

**# Driver Code**

**if \_\_name\_\_ == "\_\_main\_\_":**

**neural\_network = NeuralNetwork()**

**print ('Random weights at the start of training')**

**print (neural\_network.weight\_matrix)**

**train\_inputs = array([[0, 0, 1], [1, 1, 1], [1, 0, 1], [0, 1, 1]])**

**train\_outputs = array([[0, 1, 1, 0]]).T**

**neural\_network.train(train\_inputs, train\_outputs, 10000)**

**print ('New weights after training')**

**print (neural\_network.weight\_matrix)**

**# Test the neural network with a new situation.**

**print ("Testing network on new examples ->")**

**print (neural\_network.forward\_propagation(array([1, 0, 0])))**

1. **Write the Python code to implement ReLU.**

**What is ReLU function in Python?**

**Image result for write a program in python of relu, with out of a graph.**

**Relu or Rectified Linear Activation Function is the most common choice of activation function in the world of deep learning. Relu provides state of the art results and is computationally very efficient at the same time.**

**The rectified linear activation function (RELU) is a piecewise linear function that, if the input is positive say x, the output will be x. otherwise, it outputs zero.**

**The mathematical representation of ReLU function is,**

**f(x)= max(0,x)**

**ReLU**

**The coding logic for the ReLU function is simple,**

**if input\_value > 0:  
return input\_value  
else:  
return 0**

**A simple python function to mimic a ReLU function is as follows,**

**def ReLU(x):  
data = [max(0,value) for value in x]  
return np.array(data, dtype=float)**

**The derivative of ReLU is,**

**derivative ReLU**

**A simple python function to mimic the derivative of ReLU function is as follows,**

**def der\_ReLU(x):  
data = [1 if value>0 else 0 for value in x]  
return np.array(data, dtype=float)**

**ReLU is used widely nowadays, but it has some problems. let's say if we have input less than 0, then it outputs zero, and the neural network can't continue the backpropagation algorithm. This problem is commonly known as Dying ReLU. To get rid of this problem we use an improvised version of ReLU, called Leaky ReLU.**

**Python Code**

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

**# Rectified Linear Unit (ReLU)  
def ReLU(x):  
data = [max(0,value) for value in x]  
return np.array(data, dtype=float)**

**# Derivative for ReLU  
def der\_ReLU(x):  
data = [1 if value>0 else 0 for value in x]  
return np.array(data, dtype=float)**

**# Generating data for Graph  
x\_data = np.linspace(-10,10,100)  
y\_data = ReLU(x\_data)  
dy\_data = der\_ReLU(x\_data)**

**# Graph  
plt.plot(x\_data, y\_data, x\_data, dy\_data)  
plt.title('ReLU Activation Function & Derivative')  
plt.legend(['ReLU','der\_ReLU'])  
plt.grid()  
plt.show()**

1. **Write the Python code for a dense layer in terms of matrix multiplication.**

**class MyLayer(layers.Layer):**

**def \_\_init\_\_(self, output\_dim, \*\*kwargs):**

**self.output\_dim = output\_dim**

**super(MyLayer, self).\_\_init\_\_(\*\*kwargs)**

**def build(self, input\_shape):**

**shape = tf.TensorShape((input\_shape[1], self.output\_dim))**

**# Create a trainable weight variable for this layer.**

**self.kernel = self.add\_weight(name='kernel',**

**shape=shape,**

**initializer='uniform',**

**trainable=True)**

**super(MyLayer, self).build(input\_shape)**

**def call(self, inputs):**

**y = tf.matmul(inputs,self.kernel)**

**return (y)**

The model is as follows:

**model = tf.keras.Sequential([**

**keras.layers.Flatten(input\_shape=(28, 28)),**

**MyLayer(20, input\_shape=(1, 784)),**

**#MyLayer(input\_shape=(10,)),**

**layers.Activation('relu'),**

**MyLayer(10,input\_shape=(1, 20)),**

**#MyLayer(input\_shape=(10,)),**

**layers.Activation('relu'),**

**keras.layers.Dense(10, input\_shape=(1, 10), activation='softmax')])**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**model.fit(data, labels, epochs=1, batch\_size=1,**

**validation\_data=(val\_data, val\_labels))**

**However, when I replace the tf.matmul () with my own custom python-based matrix multiplication algorithm, it gives the following errors.**

**My custom matrix multiplication algorithm is using three nested loops to compute the output. Can somebody please clarify why the custom matrix multiplication is has 'None' for the gradient. Or can somebody guide me what am I doing wrong here. Thanks**

1. **Write the Python code for a dense layer in plain Python (that is, with list comprehensions and functionality built into Python).**

**A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.**

**Example:**

|  |
| --- |
| **numbers = [12, 13, 14,]**  **doubled = [x \*2  for x in numbers]**  **print(doubled)** |

1. **What is the “hidden size” of a layer?**

**The "hidden size" of a layer refers to the number of neurons or units in a hidden layer of a neural network. It determines the capacity of the layer to learn and represent complex patterns in the data.**

**For example, in a fully connected neural network, if a hidden layer has 128 neurons, the hidden size of that layer is 128.**

**In the context of recurrent neural networks (RNNs) or long short-term memory networks (LSTMs), the hidden size also indicates the dimensionality of the hidden state vector.**

**If you are defining a hidden layer in Python using a library like PyTorch, it might look like this:**

**Python**

**import torch.nn as nn**

**hidden\_size = 128**

**layer = nn.Linear(input\_size, hidden\_size)**

**In summary, the hidden size is a crucial hyperparameter that affects the model's ability to learn from data.**

1. **What does the t method do in PyTorch?**

**The t method in PyTorch transposes a 2D tensor, swapping its rows and columns. This is equivalent to the mathematical operation of taking the transpose of a matrix.**

**For example, if you have a tensor A of shape (m, n), applying A.t() will result in a tensor of shape (n, m).**

**Here is a simple example in Python using PyTorch:**

**Python**

**import torch**

**# Create a 2D tensor**

**A = torch.tensor([[1, 2, 3], [4, 5, 6]])**

**# Transpose the tensor**

**A\_t = A.t()**

**print(A\_t)**

**This will output:**

**tensor([[1, 4],**

**[2, 5],**

**[3, 6]])**

**In this example, the original tensor A of shape (2, 3) is transposed to a tensor of shape (3, 2).**

1. **Why is matrix multiplication written in plain Python very slow?**

**I try to find an explanation why my matrix multiplication with Numba is much slower than using NumPy's dot function. Although I am using the most basic code for writing a matrix multiplication function with Numba, I don't think that the significantly slower performance is due to the algorithm. For simplicity, I consider two k x k square matrices, A and B. My code reads**

**@njit('f8[:,:](f8[:,:], f8[:,:])')**

**def numba\_dot(A, B):**

**k=A.shape[1]**

**C = np.zeros((k, k))**

**for i in range(k):**

**for j in range(k):**

**tmp = 0.**

**for l in range(k):**

**tmp += A[i, l] \* B[l, j]**

**C[i, j] = tmp**

**return C**

**Running this code repeatedly with two random matrices 1000 x 1000 Matrices, it typically takes at least about 1.5 seconds to finish. On the other hand, if I don't update the matrix C, i.e. if I drop line 14, or replace it for the sake of a test by for example the following line:**

**C[i, j] = i \* j**

**the code finishes in about 1-5 ms. Compared to that, NumPy's dot function requires for this matrix multiplication around 10 ms.**

**What is the reason behind the discrepancy of the running times between the above code for the matrix multiplication and this small variation? Is there a way to store the value of the variable tmp in C[i, j] without deteriorating the performance of the code so significantly?**

1. **In matmul, why is ac==br?**

**In matrix multiplication (matmul), the condition ac==br ensures that the number of columns in the first matrix matches the number of rows in the second matrix. This is necessary for the dot product of the rows of the first matrix and the columns of the second matrix to be defined.**

**If matrix A has dimensions a×b and matrix B has dimensions c×d, then for the product AB to be defined, b (the number of columns in A) must equal c (the number of rows in B).**

**a x b and c x d with b = c**

1. **In Jupyter Notebook, how do you measure the time taken for a single cell to execute?**

**Measure the code execution time of a cell**

**The other way to call timeit is by using the %%timeit magic command. Rather than calculating the time it takes for a single line of code to execute, %%timeit will calculate how long it takes to run all the code inside a Jupyter notebook cell. To use it, you simply place %%timeit at the top of a cell, and then run the cell.**

**%%timeit**

**import pandas as pd**

**df = pd.DataFrame({'a': range(100000)})**

**df['b'] = df['a'] \* 5**

1. **What is elementwise arithmetic?**

**here's the final answer to perform element-wise multiplication of two matrices A and B using a mechanical solution in Python:**

**def elementwise\_multiply(A, B):**

**# get shape of A and B**

**m, n = A.shape**

**# create an empty matrix C of shape (m x n)**

**C = [[0 for j in range(n)] for i in range(m)]**

**# loop over each element in A**

**for i in range(m):**

**for j in range(n):**

**# multiply the element (i,j) in A with the corresponding element (i,j) in B**

**C[i][j] = A[i][j] \* B[i][j]**

**# return the resulting matrix C**

**return C**

**You can call this function by passing two matrices A and B as arguments:**

**import numpy as np**

**# create two matrices A and B**

**A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])**

**B = np.array([[9, 8, 7], [6, 5, 4], [3, 2, 1]])**

**# perform element-wise multiplication of A and B**

**C = elementwise\_multiply(A, B)**

**# print the resulting matrix C**

**print(C)**

**This will output the following matrix:**

**[[ 9 16 21]**

**[24 25 24]**

**[21 16 9]]**

**which is the result of element-wise multiplication of matrices A and B.**

1. **Write the PyTorch code to test whether every element of a is greater than the corresponding element of b.**

**I have a collection of tensors of common shape (2,ncol). Example:**

**torch.tensor([[1, 2, 3, 7, 8], [3, 3, 1, 8, 7]], dtype=torch.long)**

**For each tensor, I want to determine if, for each column [[a], [b]], the reversed column [[b], [a]] is also in the tensor. For example, in this case, since ncol is odd, I can immediately say that this is not the case. But in this other example**

**torch.tensor([[1, 2, 3, 7, 8, 4], [3, 3, 1, 8, 7, 2]], dtype=torch.long)**

**I would actually have to perform the check. A naive solution would be**

**test = torch.tensor([[1, 2, 3, 7, 8, 4], [3, 3, 1, 8, 7, 2]], dtype=torch.long)**

**def are\_column\_paired(matrix: torch\_geometric.data.Data) -> bool:**

**ncol = matrix.shape[1]**

**if ncol % 2 != 0:**

**all\_paired = False**

**return all\_paired**

**column\_has\_match = torch.zeros(ncol, dtype=torch.bool)**

**for i in range(ncol):**

**if column\_has\_match[i]:**

**continue**

**column = matrix[:, i]**

**j = i + 1**

**while not (column\_has\_match[i]) and (j <= (ncol - 1)):**

**if column\_has\_match[j]:**

**j = j + 1**

**continue**

**current\_column = matrix[:, j]**

**current\_column = current\_column.flip(dims=[0])**

**if torch.equal(column, current\_column):**

**column\_has\_match[i], column\_has\_match[j] = True, True**

**j = j + 1**

**all\_paired = torch.all(column\_has\_match).item()**

**return all\_paired**

1. **What is a rank-0 tensor? How do you convert it to a plain Python data type?**

**Use [Tensor.tolist()](https://pytorch.org/docs/stable/tensors.html" \l "torch.Tensor.tolist) e.g:**

**>>> import torch**

**>>> a = torch.randn(2, 2)**

**>>> a.tolist()**

**[[0.012766935862600803, 0.5415473580360413],**

**[-0.08909505605697632, 0.7729271650314331]]**

**>>> a[0,0].tolist()**

**0.012766935862600803**

**To remove all dimensions of size 1, use a.squeeze().tolist().**

**Alternatively, if all but one dimension are of size 1 (or you wish to get a list of every element of the tensor) you may use [a.flatten().tolist()](https://pytorch.org/docs/stable/torch.html" \l "torch.flatten).**

1. **How does elementwise arithmetic help us speed up matmul?**

**Elementwise arithmetic helps speed up matrix multiplication (matmul) by allowing operations to be performed in parallel, reducing the overall computational complexity. In matrix multiplication, each element of the resulting matrix is computed as the dot product of corresponding row and column vectors from the input matrices. Elementwise operations can be parallelized, leveraging modern hardware capabilities like SIMD (Single Instruction, Multiple Data) and GPU acceleration.**

**Criteria for elementwise arithmetic:**

* **Parallelism: Operations can be performed simultaneously on multiple data points.**
* **Efficiency: Reduces the number of required operations by breaking down complex tasks into simpler, concurrent tasks.**
* **Hardware Utilization: Takes advantage of specialized hardware for faster computation.**

**Definitions:**

* **Elementwise Addition: Adding corresponding elements of two matrices.**
* **Elementwise Multiplication: Multiplying corresponding elements of two matrices.**

**Example of matrix multiplication: Given matrices A and B:**

**Ci,j=∑k=1n(Ai,k×Bk,j)**

**Python code snippet for elementwise multiplication:**

**Python**

**import numpy as np**

**A = np.array([[1, 2], [3, 4]])**

**B = np.array([[5, 6], [7, 8]])**

**# Elementwise multiplication**

**C = A \* B**

**print(C)**

**This approach ensures that each element of the resulting matrix is computed efficiently, leveraging parallel processing capabilities.**

1. **What are the broadcasting rules?**

## General Broadcasting Rules

**When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimension and works its way left. Two dimensions are compatible when**

1. **they are equal, or**
2. **one of them is 1.**

**If these conditions are not met,a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes.**

**Input arrays do not need to have the same number of dimensions. The resulting array will have the same number of dimensions as the input array with the greatest number of dimensions, where the size of each dimension is the largest size of the corresponding dimension among the input arrays. Note that missing dimensions are assumed to have size one.**

**For example, if you have a 256x256x3 array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:**

**Image (3d array): 256 x 256 x 3**

**Scale (1d array): 3**

**Result (3d array): 256 x 256 x 3**

**When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or “copied” to match the other.**

**In the following example, both the A and B arrays have axes with length one that are expanded to a larger size during the broadcast operation:**

**A (4d array): 8 x 1 x 6 x 1**

**B (3d array): 7 x 1 x 5**

**Result (4d array): 8 x 7 x 6 x 5**

1. **What is expand\_as? Show an example of how it can be used to match the results of broadcasting.**

**a = torch.rand(2, 3)**

**b = torch.rand(4, 3)**

**c = b.expand\_as(a)**

**print(c)**

**a = torch.rand(2, 3)**

**b = torch.rand(2, 4)**

**c = b.expand\_as(a)**

**print(c)**