**DEEP LEARNING ASSIGNMENT\_11**

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

Here's an example of how you can implement a single neuron in Python:

import numpy as np

class Neuron:

def \_\_init\_\_(self, inputs):

self.weights = np.random.randn(inputs)

self.bias = np.random.randn(1)

def sigmoid(self, x):

return 1 / (1 + np.exp(-x))

def feedforward(self, inputs):

total = np.dot(inputs, self.weights) + self.bias

return self.sigmoid(total)

neuron = Neuron(3)

inputs = np.array([1, 2, 3])

output = neuron.feedforward(inputs)

print(output)

In this example, the Neuron class takes the number of inputs as an argument when it is instantiated. The weights attribute is initialized with random values using np.random.randn(). The bias attribute is also initialized with a random value.

The sigmoid() method implements the sigmoid activation function, which is commonly used in neural networks. The feedforward() method takes an array of inputs and calculates the weighted sum of the inputs, adds the bias, and passes the result through the sigmoid activation function.

Finally, an instance of the Neuron class is created with 3 inputs and its feedforward output is calculated and printed. Note that this is a very simple implementation of a neuron and real-world neural networks usually consist of many neurons interconnected in a specific way.

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

Here's an example of how you can implement the ReLU activation function in Python:

import numpy as np

def relu(x):

return np.maximum(0, x)

inputs = np.array([-1, 2, -3, 4, -5, 6])

output = relu(inputs)

print(output)

The relu() function takes an input array and returns an array with the same shape, where each element is the maximum of 0 and the corresponding input element. The np.maximum() function from the numpy library is used to calculate the maximum of 0 and each input element.

Finally, an example input array is passed through the relu() function and its output is printed. Note that this implementation is simple and efficient, but there are other ways to implement the ReLU activation function, including element-wise operations in a deep learning library such as TensorFlow or PyTorch.

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

Here's an example of how you can implement a dense layer using matrix multiplication in Python:

import numpy as np

class DenseLayer:

def \_\_init\_\_(self, inputs, units, activation=None):

self.weights = np.random.randn(inputs, units)

self.bias = np.zeros((1, units))

self.activation = activation

def feedforward(self, inputs):

output = np.dot(inputs, self.weights) + self.bias

if self.activation is not None:

output = self.activation(output)

return output

layer = DenseLayer(3, 4, activation=np.tanh)

inputs = np.array([[1, 2, 3]])

output = layer.feedforward(inputs)

print(output)

In this example, the DenseLayer class takes the number of inputs, the number of units, and an optional activation function as arguments when it is instantiated. The weights attribute is initialized with random values using np.random.randn() and the bias attribute is initialized with zeros using np.zeros().

The feedforward() method takes an input array and calculates the dot product of the inputs and the weights, adds the bias, and passes the result through the activation function, if one was specified. The np.dot() function from the numpy library is used to perform the matrix multiplication.

Finally, an instance of the DenseLayer class is created with 3 inputs, 4 units, and the hyperbolic tangent activation function, and its feedforward output is calculated and printed.

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

Here's an example of how you can implement a dense layer using only built-in Python functions:

def relu(x):

return max(0, x)

def dense\_layer(inputs, weights, bias, activation=None):

output = sum(i \* w for i, w in zip(inputs, weights)) + bias

if activation is not None:

output = activation(output)

return output

weights = [0.1, 0.2, 0.3]

bias = 0.4

inputs = [1, 2, 3]

output = dense\_layer(inputs, weights, bias, activation=relu)

print(output)

In this example, the relu() function implements the ReLU activation function, which returns the maximum of 0 and its input. The dense\_layer() function takes a list of inputs, a list of weights, a bias, and an optional activation function as arguments.

The dense\_layer() function calculates the dot product of the inputs and the weights using a list comprehension, adds the bias, and passes the result through the activation function, if one was specified. The zip() function is used to iterate over the inputs and weights in parallel.

Finally, a list of weights, a bias, and a list of inputs are defined, and the dense\_layer() function is called with the ReLU activation function. The output is printed. Note that this is a simple implementation of a dense layer and real-world neural networks usually consist of many layers interconnected in a specific way.

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

The "hidden size" of a layer refers to the number of neurons or units in that layer. In the context of a neural network, each unit in a layer represents a single computation that takes in inputs from the previous layer, performs some mathematical operation, and produces an output that is passed on to the next layer.

The hidden size determines the number of parameters that the layer has, as well as the capacity of the layer to learn complex relationships between inputs and outputs. Increasing the hidden size typically increases the model's ability to fit the data, but also increases the risk of overfitting. Decreasing the hidden size reduces the risk of overfitting but may also limit the model's ability to fit the data.

In practice, the hidden size is a hyperparameter that is usually set experimentally, based on the size and complexity of the data being modeled, the architecture of the network, and the computational resources available. The optimal hidden size can depend on many factors and may vary greatly from one problem to another.

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

In PyTorch, the t() method is used to transpose a tensor, which is an array-like object in PyTorch that is used to store and manipulate multi-dimensional arrays.

The transpose operation swaps the axes of a tensor, effectively flipping it along one or more dimensions. In PyTorch, the t() method can be used to transpose a tensor along all dimensions by default, or along specific dimensions by passing a tuple of axis permutation to the method.

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

Matrix multiplication in plain Python using basic list comprehensions can be very slow due to several reasons:

Interpreted language: Python is an interpreted language, which means that the code is executed line by line, without the use of a compiler that can optimize the code. This results in slower execution times compared to compiled languages like C or Fortran.

Loops: Matrix multiplication involves a lot of repeated computations, and in Python, these computations are often performed using loops. Loops in Python are slow compared to vectorized operations that can be performed using libraries like NumPy or PyTorch.

Dynamic typing: Python is a dynamically typed language, which means that the type of a variable is determined at runtime, not at compile time. This results in slower execution times, since the interpreter has to do additional work to determine the type of each variable.

Overhead of list comprehensions: List comprehensions are a convenient and readable way to write code in Python, but they come with a performance overhead, since the interpreter has to evaluate the expression for each element in the list.

To achieve better performance for matrix multiplication in Python, it's recommended to use optimized libraries like NumPy or PyTorch, which are specifically designed for numerical computing and can perform matrix multiplication much faster than plain Python.

**8. In matmul, why is ac==br?**

In matrix multiplication, the product of two matrices A (of shape m x n) and B (of shape p x q) is only defined if n is equal to p. If this condition is satisfied, then the result of the matrix multiplication is a matrix C (of shape m x q), where each element c\_ij is given by the dot product of the i-th row of A and the j-th column of B.

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

You can measure the time taken for a single cell to execute in Jupyter Notebook using the time module in Python. To do this, you can use the following code:

import time

start = time.time()

# Your code here

end = time.time()

print("Time taken: {:.2f} seconds".format(end - start))

This code measures the current time before and after the execution of the cell and calculates the difference, which is the time taken for the cell to execute. The result is then printed in seconds, rounded to two decimal places.

You can place this code at the beginning and end of the cell you want to measure, and it will print the time taken for that cell to execute.

**10. What is elementwise arithmetic?**

Elementwise arithmetic refers to mathematical operations that are performed on individual elements of arrays or tensors, rather than on the entire arrays or tensors.

In the context of arrays or tensors, elementwise arithmetic operations are performed element-by-element, rather than on the whole arrays or tensors. For example, in elementwise addition, each element of one array or tensor is added to the corresponding element of another array or tensor, resulting in a new array or tensor with the same shape as the original arrays or tensors.

Elementwise arithmetic is a common operation in many machine learning algorithms and is often used in deep learning frameworks such as TensorFlow and PyTorch. These frameworks provide built-in functions for elementwise arithmetic operations, such as add, subtract, multiply, divide, and others, which make it easy to perform these operations on arrays and tensors.

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

**corresponding element of b.**

Here is an example of how you can test whether every element of a tensor a is greater than the corresponding element of tensor b in PyTorch:

import torch

a = torch.tensor([1, 2, 3, 4])

b = torch.tensor([0, 1, 2, 3])

result = (a > b).all().item()

if result:

print("All elements of a are greater than the corresponding elements of b.")

else:

print("Not all elements of a are greater than the corresponding elements of b.")

In this example, the comparison operation a > b returns a tensor of the same shape as a and b with elements that are either True or False depending on whether the corresponding element of a is greater than the corresponding element of b. The method all() is then called on this tensor, which returns a scalar tensor indicating whether all elements of the tensor are True or not. Finally, the method item() is called on this scalar tensor to obtain the actual boolean value.

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

A rank-0 tensor is a tensor with only one element, also known as a scalar tensor. It is considered a tensor of rank-0 because it has only one dimension, while tensors with more than one element have a higher rank (for example, a tensor with two elements would be rank-1, a tensor with three elements would be rank-2, and so on).

To convert a rank-0 tensor to a plain Python data type, you can use the item() method of the tensor. For example:

import torch

tensor = torch.tensor([1])

scalar = tensor.item()

print(scalar) # 1

print(type(scalar)) # <class 'int'>

In this example, the rank-0 tensor tensor has a single element with the value 1. The item() method is called on the tensor, which returns the scalar value 1 of type int.

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

Elementwise arithmetic does not directly help speed up matrix multiplication. Matrix multiplication is an inherently sequential operation that requires a series of dot products to be computed between the rows of the first matrix and the columns of the second matrix. There is no obvious way to speed up this operation using elementwise arithmetic.

However, many modern deep learning libraries (including PyTorch) take advantage of GPU acceleration for matrix multiplication, which can significantly speed up the operation. The GPU parallelizes the calculation across multiple cores, allowing many dot products to be computed in parallel. In this way, the GPU can significantly speed up the matrix multiplication operation compared to computing it on a CPU.

Additionally, many deep learning libraries (including PyTorch) also provide optimized implementations of matrix multiplication that are optimized for performance on modern hardware. These implementations can further speed up the calculation.

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

Broadcasting is a mechanism used in many numerical programming libraries, including PyTorch, to automatically expand the dimensions of arrays with smaller shapes so that they can be used in arithmetic operations with arrays with larger shapes. Broadcasting rules define how the smaller arrays are expanded to match the larger arrays.

The broadcasting rules can be summarized as follows:

If two arrays have different numbers of dimensions, then the shape of the array with fewer dimensions is padded with ones on the left.

If the shapes of two arrays along a given dimension do not match, and one of the shapes along that dimension is equal to 1, then the shape of the array with a shape of 1 is stretched to match the other array.

For example, consider two arrays a and b with shapes (2, 3) and (3,), respectively. The shape of b is broadcast to (1, 3), and then both arrays have the same shape of (2, 3). This allows the elementwise addition a + b to be performed.

The broadcasting rules can be a powerful tool for writing more concise and efficient code, but they can also lead to unexpected results if not used correctly. It is important to understand the broadcasting rules and to carefully check the shapes of arrays before using them in arithmetic operations.

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

**broadcasting.**

expand\_as is a function in PyTorch that is used to expand the dimensions of a tensor to match the shape of another tensor. The function can be used to mimic the behavior of broadcasting, but with greater control over the expansion process.

Here is an example of how expand\_as can be used to match the results of broadcasting:

import torch

a = torch.tensor([1, 2, 3])

b = torch.tensor([[1], [2], [3]])

# Using broadcasting

result\_broadcast = a + b

print(result\_broadcast)

# tensor([[2, 3, 4],

# [3, 4, 5],

# [4, 5, 6]])

# Using expand\_as

result\_expand\_as = a.unsqueeze(1).expand\_as(b) + b

print(result\_expand\_as)

# tensor([[2, 3, 4],

# [3, 4, 5],

# [4, 5, 6]])

In this example, a and b are two tensors with different shapes. Broadcasting would automatically expand the dimensions of a to match the shape of b, but using expand\_as allows us to explicitly control the expansion process. The unsqueeze method is used to add a new dimension to a with shape (1,), and the expand\_as method is used to expand this new dimension to match the shape of b. The result of the expansion is a tensor with the same shape as b, allowing us to perform the elementwise addition.