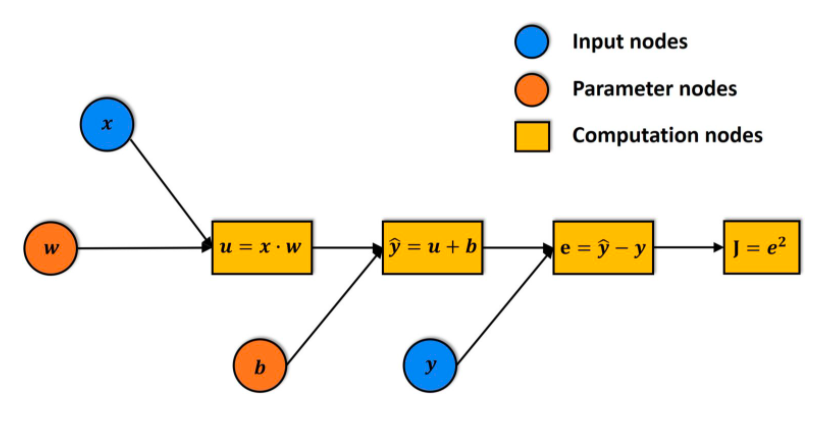
Pytorch

**Computation Graph**

import torch

import numpy as np

# Tensor

#   A generalize name for any dimensional matrices(nd) including scalar values(1d).

#     Types of tensors

#       1. Scalar Tensors (Tensors having only 1 dimension)

#       2. N dimentional tensor(Tensors having dimension more than 1 that is 2x2,3x3,4x4,nxm matrices)

#

#

# Difference between dimensions and Order/Size/Shape

# Dimensions --> The total number of opening '[' square brackets before any closing square bracket is called the dimension of a tensor.

# Size --> Total rows , Total columns

# Example [[[[[        [1,2],[3,4],[5,6]            ]]]]]   // here dimensions --> 6  and  shape --> 3,2

# x = torch.tensor([[[ [1,2,3] ]]])

# print(x.ndimension()) // output ---> 4

# Ways to create tensors

# Empty Function

# torch.empty(provide size OR order(if a multidimentional tensor)

# Empty function with all Keyword Arguments,

#   torch.empty(\*size, \*, out=None, dtype=None, layout=torch.strided, device=None, requires\_grad=False, pin\_memory=False)

#     1. out --> gives the output of described tensor in given variable

#           torch.empty(5,out=x)

#     2. 'requires\_grad' if set to True, then it allows calculating the derivative/gradient of the computation graph.

#           a = torch.tensor(3, requires\_grad=True, dtype=float) --> creates a scalar tensor 'a' with value 3.

#           x = 3\*a\*\*2 + 4\*a + 2 --> Computation graph

#           x.backward() --> if backward(back propagation) is called then a gradient/slope/derivative is calculated. It calculates dx/da.

#           a = torch.tensor(3, requires\_grad=True, dtype=float)

#           x = 3\*a\*\*2 + 4\*a + 2

#           print(a.grad) Gradient ---> None    because backward was not called at this point

#           x.backward(retain\_graph=True)

#           print(a.grad) Gradient ---> 22    i = 1      calling backward ith time will return gradient\*i

#           x.backward(retain\_graph=True)

#           print(a.grad) Gradient ---> 44    i = 2      calling backward ith time will return gradient\*i

#           x.backward(retain\_graph=True)

#           print(a.grad) Gradient ---> 66    i = 3      calling backward ith time will return gradient\*i

#           we are not allowed to call backward twice because once we call backward(), the intermediary results don’t exist

#           and the backward pass cannot be performed again setting retain\_graph backward(retain\_graph=True) to make a backward

#           pass that will not delete intermediary results, and so we are now allowed to call backward function again but

#           the gradient we get will be --> Gradient = (Gradient in first backward call \* ith call )

#     3. dtype --> it is used to specify the data type of the tensor if it is not set then default type 'float' will be set.

#     4. device --> we can specify the device used by the tensor for computations. There are three type of devices,

#                       1. CPU

#                       2. GPU

#                       3. TPU

#         example,

#            mydev = "cuda" if torch.cuda.is\_available() else "cpu"

#            x = torch.tensor([   [1,2,3], [4,5,6], [7,8,9]    ]   , device=mydev) // now this tensor will use cude if available else use cpu.

#     5. size(n,m) --> to specify dimensions/size of the tensor

# Tensor Function

# torch.tensor([  [1,2,3],[4,5,6],[7,8,9]  ]) OR torch.tensor( "Numpy Array") OR torch.tensor("list of values of any type or mixture of types")

# // All the keyword parameters are same as described for empty function

# Rand Function

# torch.rand((3,3)) //rand is used to make tensors of random values. (3,3) represents the size/shape/dimension of the tensor.

# // All the keyword parameters are same as described for empty function

# Ones Function

# torch.ones((3,3)) //ones is used to make a tensor having values only 1s. (3,3) represents the size/shape/dimension of the tensor.

# // All the keyword parameters are same as described for empty function

# Eye Function

# torch.eye(5) //eye is used to make identity matrices/tensors. '5' here indicates that its a 5x5 tensor having 5 ones in diagonal.

# // All the keyword parameters are same as described for empty function.

# Zeros Function

# torch.rand((3,3)) //zeros is used to make tensors of values 0. (3,3) represents the size/shape/dimension of the tensor.

# // All the keyword parameters are same as described for empty function

# x = torch.arange(start,stop,stepsize) // used to create only 1 dimensional tensors with start->end range and a stepsize.

# x = torch.arange(0,50,5)

# print(x)   // tensor([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])

# print(x.bool())  // tensor([False,  True,  True,  True,  True,  True,  True,  True,  True,  True]) coverts all >0 values to true and 0 as false

# print(x)

# print(x.short()) // converts dtype to int16 bit

# print(x.int())  // converts dtype to int32 bit

# print(x.long()) // converts dtype to int64 bit

# print(x.half()) // converts dtype to float16 bit

# print(x.float()) // converts dtype to float32 bit

# print(x.double()) // converts dtype to float64 bit

# Note: These functions does not modify the original tensors but in order to modify the original

# tensor or to convert a tensor from one type to another then we do that by assignment,

# Example,

# x = x.bool()

# print(x) // tensor([False,  True,  True,  True,  True,  True,  True,  True,  True,  True])

# Arrays to tensor conversion

# x = np.array([1,2,3,4,5])

# print(x.dtype)

# x = torch.from\_numpy(x)

# print(x.dtype)

# List to tensor

# data = [[1, 2],[3, 4]]

# x\_data = torch.tensor(data)

# x = ["1","Tashik", "Fakhiha", "Hassan", "23.5"]

# y =  torch.tensor(x) // A tensor can not contain strings it can only contain int,floats,double etc but not string type.

# print(y) // error

# x = [1,2,3]

# y =  torch.tensor(x) // Allowed

# print(y) // no error

# x = [1,2.2,3]

# y =  torch.tensor(x) // if a single value is a float then all will be treated as float

# print(y) // no error

# Tensor Math, Comparison and basic functions

# Note: In order to do operations between two tensors, the size or order or the dimension of tensors must be same.

# x =  torch.tensor([10,20,30,50])

# y = torch.tensor([10,10,10,10])

# x = torch.add(x,y)

# OR

# x = x+y

# print(x) // tensor([20, 30, 40, 60])

# x =  torch.tensor([10,20,30,50])

# y = torch.tensor([10,10,10,10])

# x = torch.sub(x,y)

# OR

# x = x-y

# print(x) // tensor([ 0, 10, 20, 40])

# x =  torch.tensor([10,20,30,50])

# y = torch.tensor([10,10,10,10])

# x = torch.div(x,y)

# OR

# x = x/y

# print(x) // tensor([1., 2., 3., 5.])

# x =  torch.tensor([10,20,30,50])

# y = torch.tensor([10,10,10,10])

# x = torch.mul(x,y)

# OR

# x = x\*y

# OR

# x = x.mm(y) # mm --> matrix multiplication

# print(x) // tensor([1., 2., 3., 5.])

# x =  torch.tensor([[10,20],[30,50]])

# y = torch.tensor([[10,10],[10,10]])

# x.add\_(y) #// same as x = x+y this style of adding with a post \_ is called inplace operation

# x.sub\_(y) #// same as x = x-y this style of adding with a post \_ is called inplace operation

# x.mul\_(y) #// same as x = x\*y this style of adding with a post \_ is called inplace operation

# x.div\_(y) #// inplace division is not allowed

# x =  torch.tensor([1,2,3])

# x = x.pow(2)

# print(x) // tensor([1, 4, 9])

# OR

# x =  torch.tensor([1,2,3])

# x = x\*\*2

# print(x)

# x =  torch.tensor([1,2,3,4,5])

# x = x>2

# print(x) // tensor([False, False,  True,  True,  True])

# x =  torch.tensor([1,2,3,4,5])

# x = x-2

# print(x) // tensor([-1,  0,  1,  2,  3])

# x =  torch.tensor([1,2,3,4,5])

# x = x\*2

# print(x) // tensor([ 2,  4,  6,  8, 10])

# etc

# Matrix power function ( it works only on floats and doubles)

# x =  torch.tensor([ [1.1,2,3],[4,5,6], [7,8,9] ])

# x = x.matrix\_power(1) // matrix it self

# x = x.matrix\_power(0) // identity matrix of given matrix 'x'

# x = x.matrix\_power(2) // matrix multiplied by it self 2 times

# x = x.matrix\_power("any value 'n' >1") // matrix multiplied by itself n times where n > 1

# x = x.matrix\_power(-1) // matrix inverse

# x = x.matrix\_power("any value 'n' <-1 ") // matrix inversed n times

# x =  torch.tensor([ [1.1,2,3],[4,5,6], [7,8,9] ])

# x = x.matrix\_exp() // calculates exponential of all the elements of the matrix

# print(x)

# tensor([[1143472.0000, 1396604.3750, 1656827.6250],

#         [2574169.2500, 3144020.7500, 3729830.2500],

#         [4011957.5000, 4900094.5000, 5813107.0000]])

# x =  torch.tensor([ 1,2,3 ])

# y =  torch.tensor([ 4,5,6 ])

# x = x.dot(y) // Works only on 1 dimensional tensors. It calculates dot product of 2 tensors (1)(4)+(2)(5)+(3)(6) = 32

# print(x) // tensor(32)

# batch matrix multiplication

# batch = 40

# m = 20

# n = 30

# p = 25

# x =  torch.rand(batch,m,n)

# y =  torch.rand(batch,n,p)

# x =  torch.bmm(x,y) #// output will be same as normal matrix multiplication

# Note: batch matrix multiplication only works on '3 dimensional' matrices the three dimensions are (batch, rows, columns).

# In batch multiplication, two tnsors are multiplied with each other in batches (small chunks of given batch size) and

# then final result is calculated. This is done in order to improve the efficiency.

# print(x)

# Note: Dimension value tells in which dimension max should be searched. max function returns values and indices of max values in specified dimensions.

# when dimension parameter is not set then a single value is returned.

# x = torch.tensor([[ [1,2,3], [4,5,6], [7,8,9] ]])

# values, indices = torch.max(x,dim=0)

# print(values)

# print(indices)

# Output

# Values([[1, 2, 3],

#         [4, 5, 6],

#         [7, 8, 9]])

# Indices([[0, 0, 0],

#         [0, 0, 0],

#         [0, 0, 0]])

# values, indices = torch.max(x,dim=1)

# print(values)

# print(indices)

# Output

# Values([[7, 8, 9]])

# Indices([[2, 2, 2]])

# values, indices = torch.max(x,dim=2)

# print(values)

# print(indices)

# Output

# Values([[3, 6, 9]])

# Indices([[2, 2, 2]])

# same for min,sum etc

# x = torch.tensor([[ [1,2,3], [4,5,6], [7,8,9] ]])

# y = torch.argmax(x,dim=1)

# z = torch.argmin(x,dim=1)

# when the size of arrays becomes too large we use argmax and argmin to find minimum and maximum value indices more efficiently.

# x = torch.tensor([1,2,3,4,5,6,7,8,9,10])

# x.clamp(min=4, max = 8)

# if thre exist any value lesser than the min value of clamp then clamp function sets all the values < min to minimum value given

# similarly, all the values > than the given max value will become the max value.

# output --> tensor([4, 4, 4, 4, 5, 6, 7, 8, 8, 8])

# x = torch.tensor([0,1,2,3,4,5,6,7,8,9,10])

# print(x.any()) # if a single non zero value exist in the array then it returns true else return false

# print(x.all()) # if all the values are non zero then only it returns true else returns false.

# x = torch.tensor([[0,1,2],[3,4,5],[6,7,8]])

# x = x.t() // used for calculating the transpose

# print(x)

# Output

# tensor([[0, 3, 6],

#         [1, 4, 7],

#         [2, 5, 8]])

# Tensor Indexing

# x = torch.tensor([0,1,2,3,4,5,6,7,8,9,10])

# indices = [1,2,3]

# y = x[indices] // we can also directly print the values of provided indices without assigning to new variable.

# print(y) // only prints the values of provided indices

# x = torch.tensor([0,1,2,3,4,5,6,7,8,9,10])

# y = x[(x > 3) & (x < 8)] // only gives values > 3 and < 8. we can also directly print it without assigning to a new variable y

# print(y)

# x = torch.tensor([0,1,2,3,4,5,6,7,8,9,10])

# print(x[(x.remainder(2) == 0) & (x.remainder(4) == 0)]) // only gives elements who has 0 remainder when divided by 2 and 4

# where function

# where(condition, if part if condition satisafies, els part if condition does not satisfies)

# x = torch.tensor([0,1,2,3,4,5,6,7,8,9,10])

# print(torch.where((x>3) & (x<7),5,x)) // if condition satisfy, assign 5 else let x as it is.

# Output --> tensor([ 0,  1,  2,  3,  5,  5,  5,  7,  8,  9, 10])

# x = torch.tensor([[[ [1,2,3] ]]])

# print(x.ndimension()) // gives total dimensions of tensor

# x = torch.tensor([0,1,2,3,4,5,6,7,8,9,10])

# print(x.numel()) // returns total "number of elements"

# Tensor Reshaping

# Reshaping using view function

# Note: when using view, the original tensor should have contiguous memory. if memory is non contiguous, then first call contiguous

# function then call view like this --> x.contiguous().view() this will work and will give no errors.

# Also, when reshaping using view/reshap make sure that assignment to new or the same value should be done in order to make difference

# in the original tensor's shape.

# x = torch.tensor([0,1,2,3,4,5,6,7,8])

# print(x.shape) // Output torch.Size([9])

# x = x.view(3,3) // The reshaping value should not exceed the size of the original tensor it should cover all i.e 3x3 = 9(original tensor size)

# print(x.shape) // Output torch.Size([3, 3])

# Reshaping using reshape function

# no contiguous memory needed

# x = torch.tensor([0,1,2,3,4,5,6,7,8])

# x = x.reshape(3,3)

# print(x.shape)

x = torch.rand((2,5))

y = torch.rand((2,5))

x = torch.cat(x,y)

print(x)