CSE 673 COMPUTATIONAL VISION

VENU GOVINDARAJU
DEEN DAYAL MOHAN

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Covid-19 Guidelines

- Effective Aug. 3, the University at Buffalo will require all students, employees and visitors – regardless of their vaccination status – to wear face coverings while inside campus buildings. This includes classrooms, hallways, libraries and other common spaces, as well as UB buses and shuttles.

- Students are expected to wear mask in class during lectures (unless you have a UB approved exception)

- Public Health Behavior Expectations https://www.buffalo.edu/studentlife/who-we-are/departments/conduct/coronavirus-student-compliance-policy.html

Project Details

- Team of 3-4 depending upon final enrollment. 40% of the grade
- Set of topics provided by us
- Two presentations
- Reports of sample projects posted on Piazza
- Satisfies Masters Project requirement

Agenda

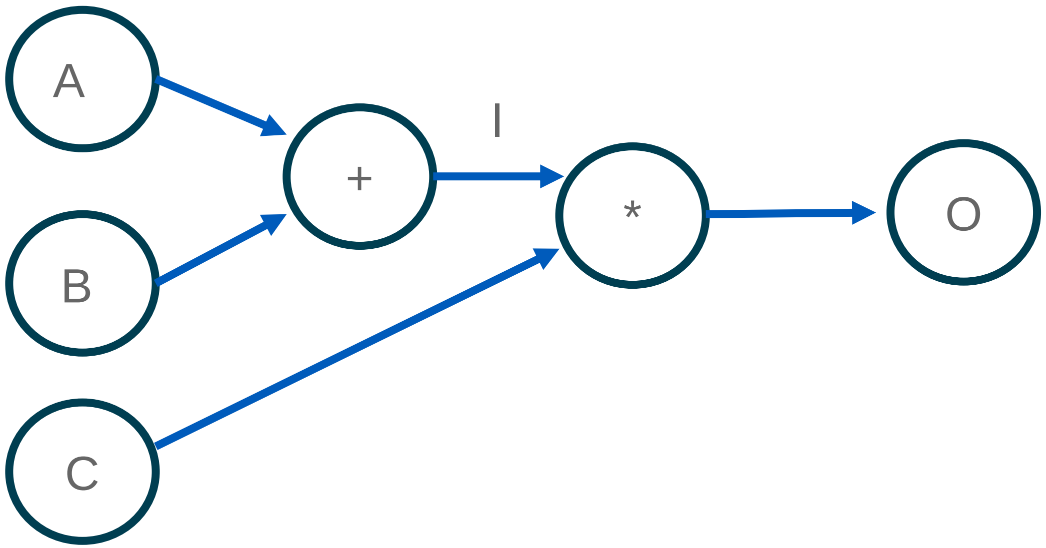
- Computational Graph
- Backpropagation
- How is it implemented?
- Assignment 1

COMPUTATIONAL GRAPH

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

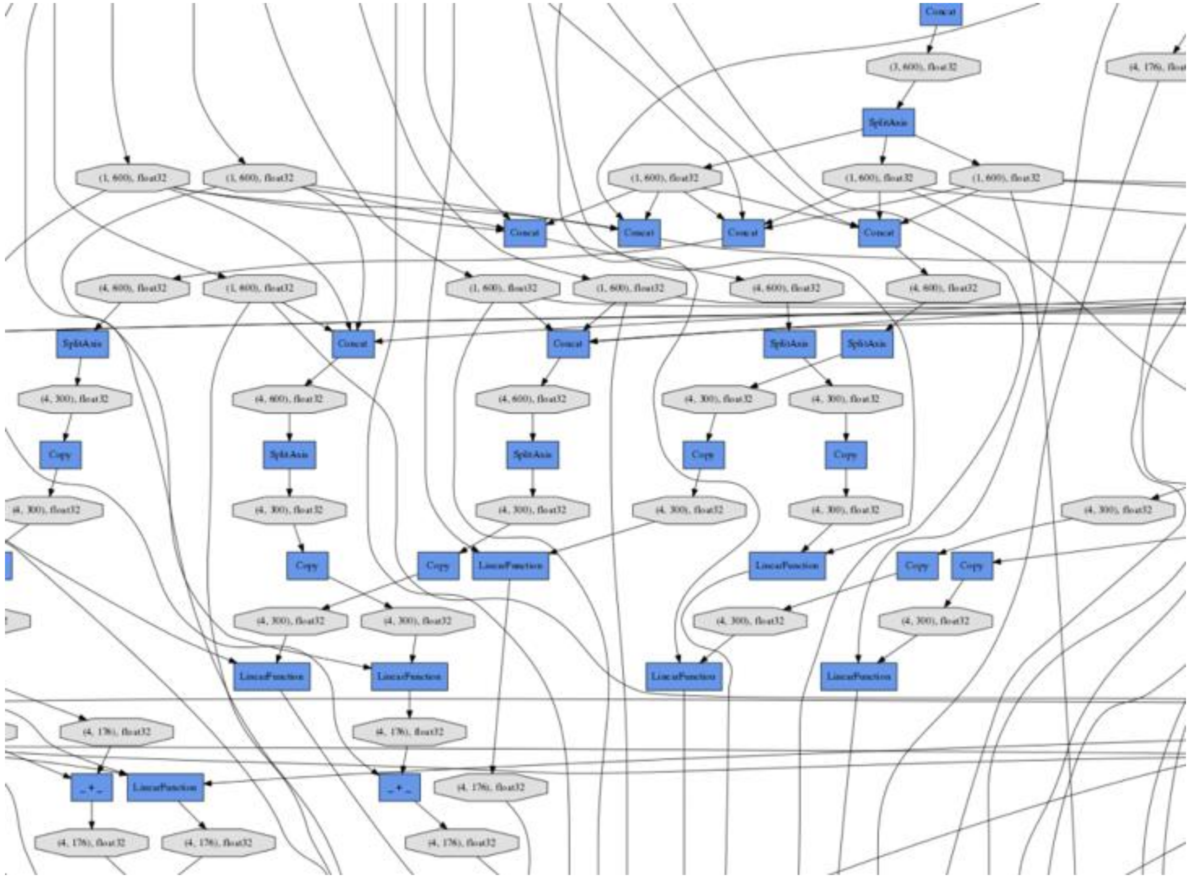
Computational Graph

- Computational graphs are a nice way to think about mathematical expressions.
- For example consider O = (A + B) * C. The computational graph for the function would look like this:



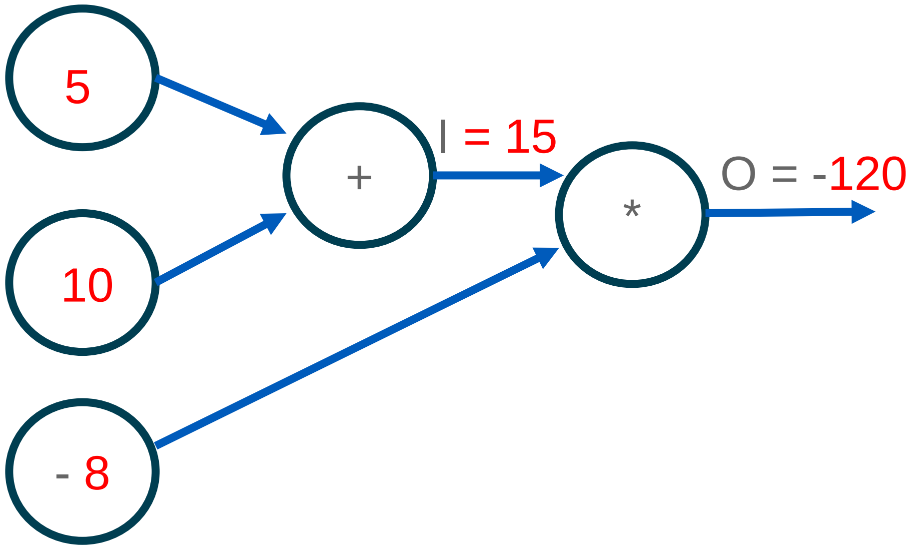
Computational Graph

- These computational graphs can get large.



Small section of the computational graph formed from an RNN based model

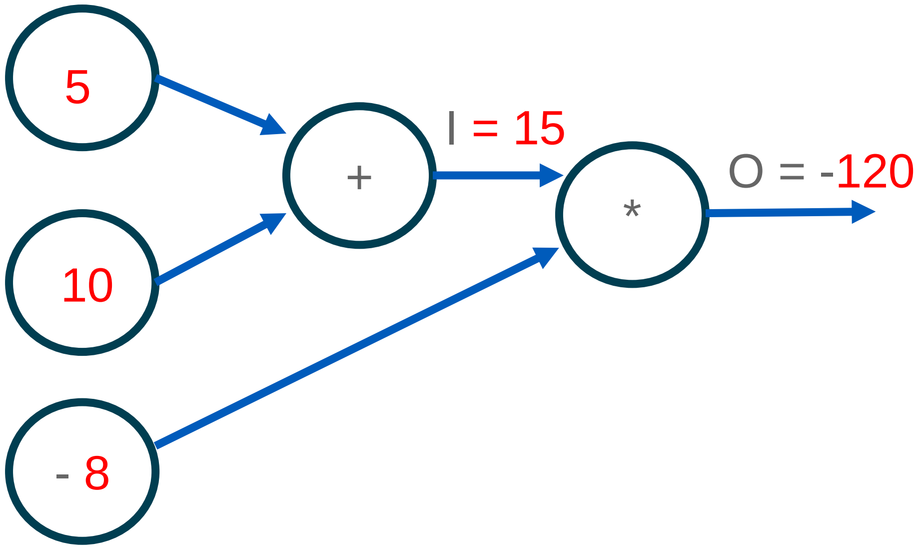
Computational Graph



- These values are calculated when the computation is executed
- In the above diagram values flow from left to right

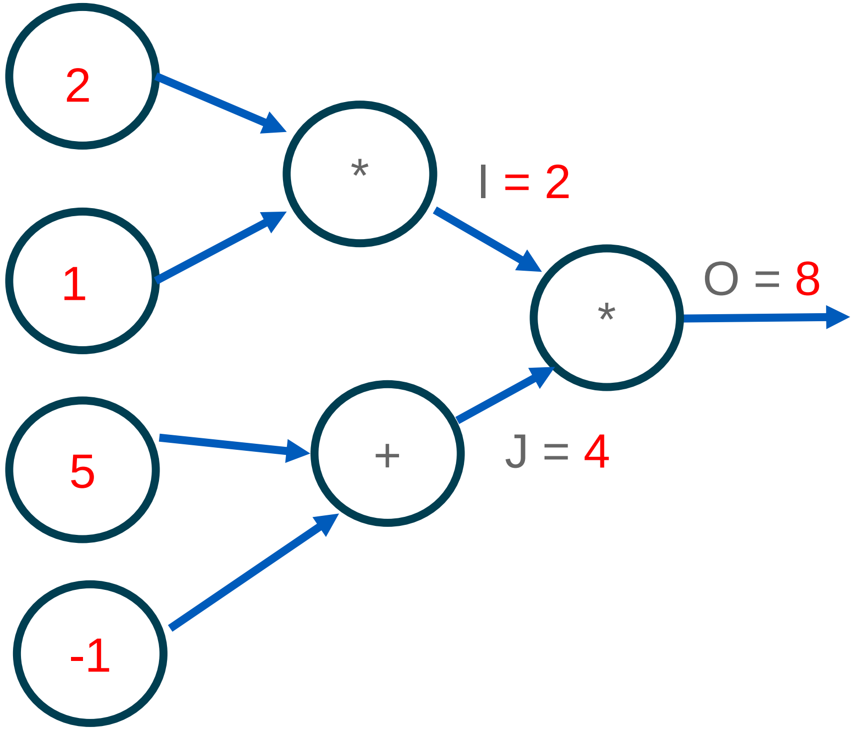
Computational Graph

- The computational graph is directed and acyclic in nature
- The values are computed and fed in based on the topological order of the nodes
- The flow of values in the context of neural networks are called forward propagation



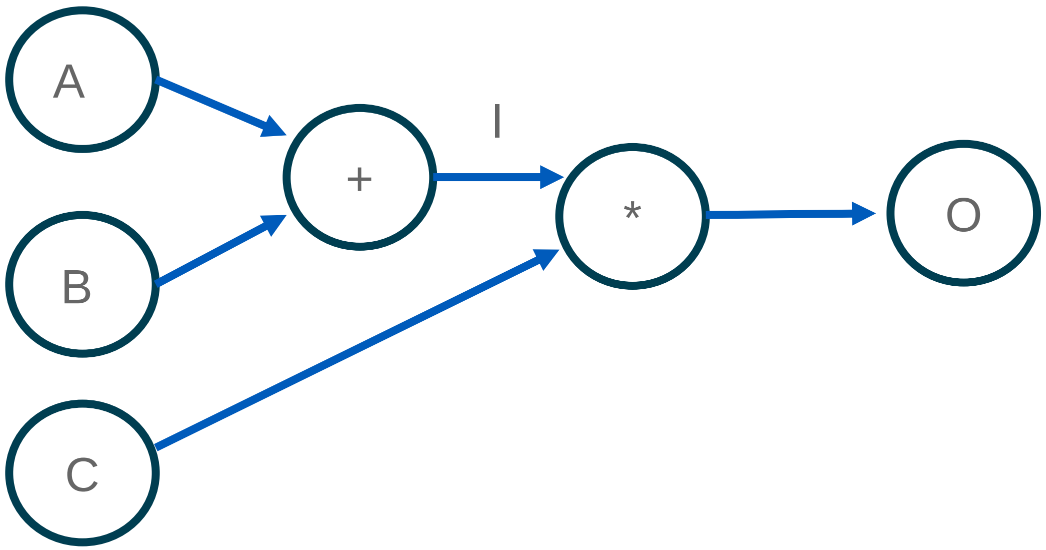
Computational Graph

- Let us look at another example



Computational Graph

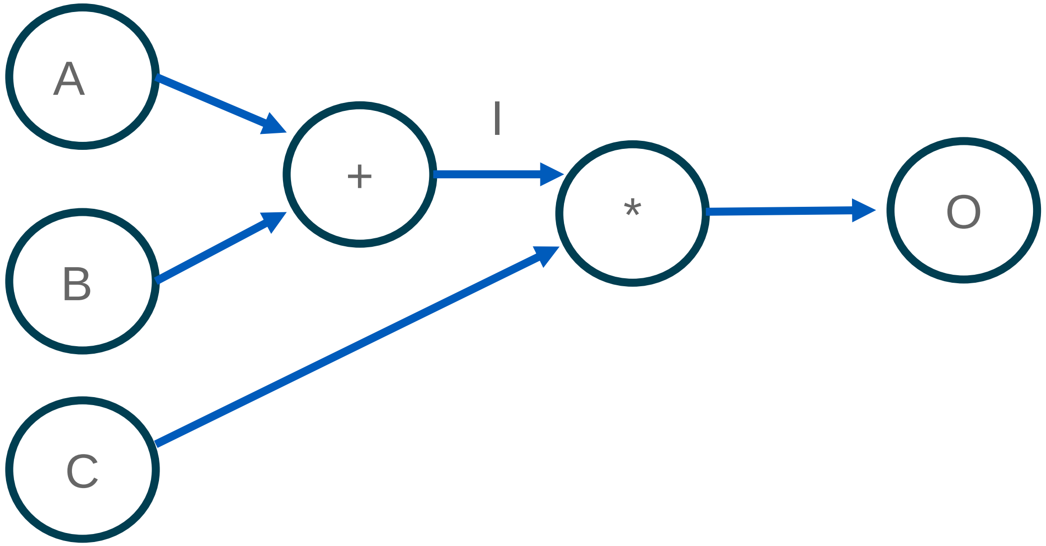
- Now let us look at how to compute the influence of each input on the output



Computational Graph

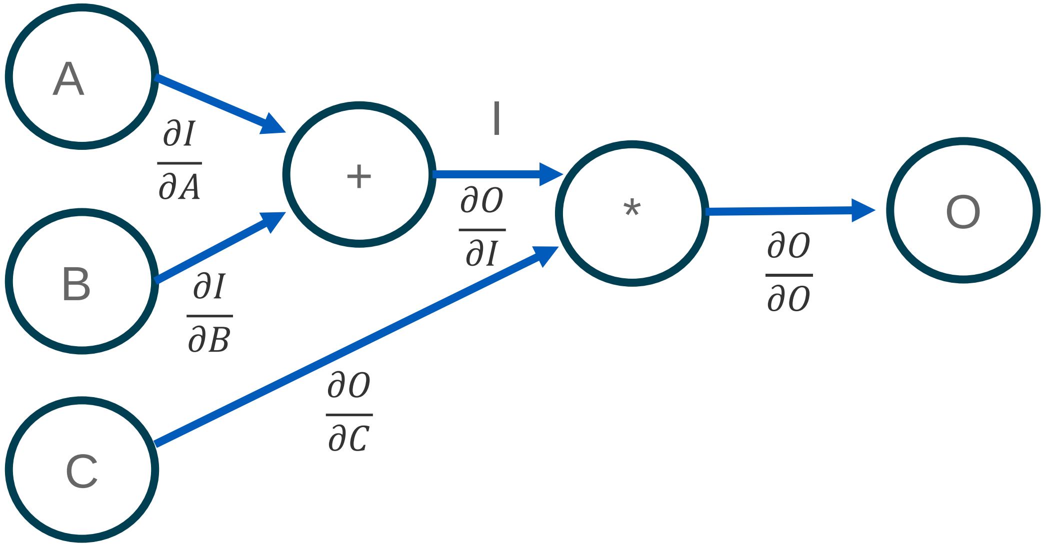
- What is the mathematical tool that we should use to find the influence of the inputs on the output ?
  - Partial derivatives to find the gradient

- So for the function O = (A + B) * C , we need to compute
  - $\frac{\partial O}{\partial A}, \frac{\partial O}{\partial B}, \frac{\partial O}{\partial C}$



Computational Graph

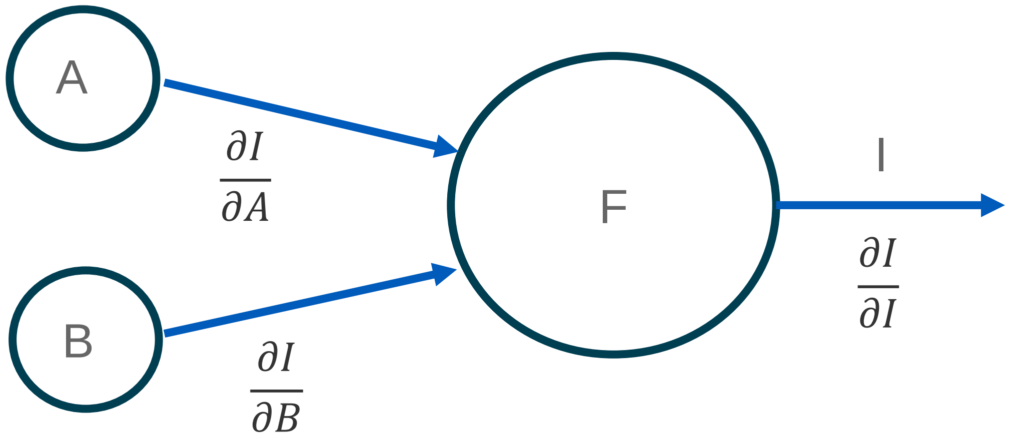
- These derivatives are not directly inferable. Let us see what are the derivatives that are directly inferable



- This derivatives are called local derivatives

Computational Graph

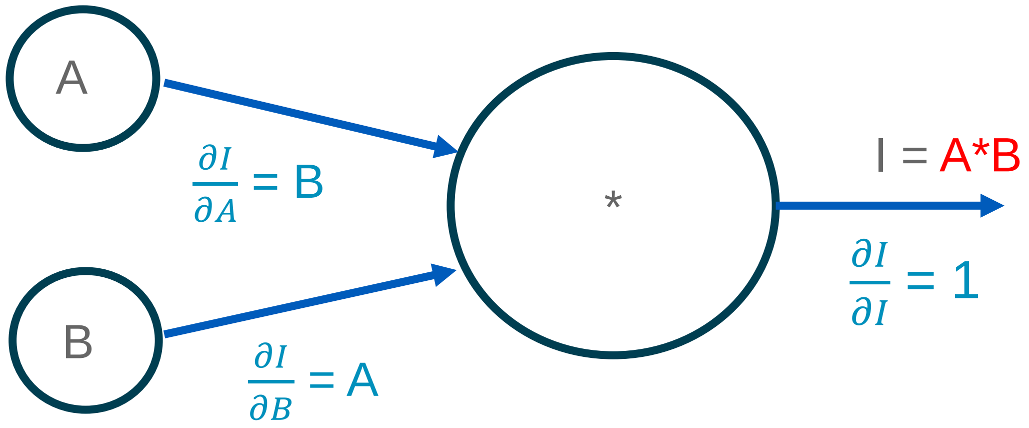
- Let us think about the local derivatives a bit more



- F can be any mathematical function that is differentiable

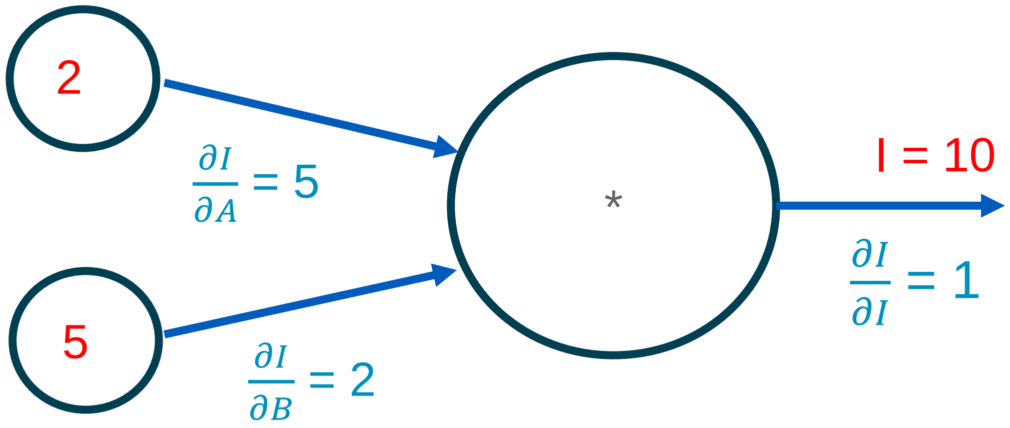
Computational Graph

- Let us look at an example



Computational Graph

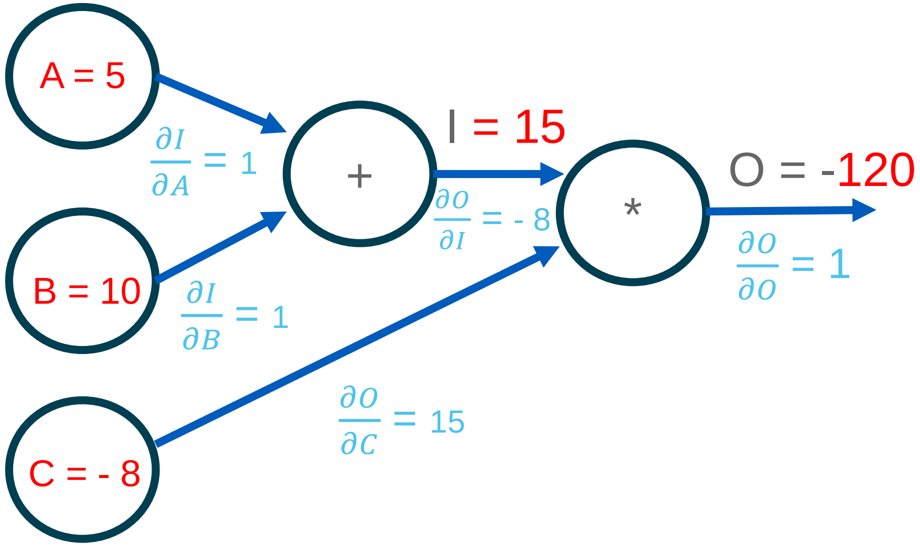
- Lets use real values instead



- The take away here should be, If input B is increased by x times it would change the output by a factor of 2x

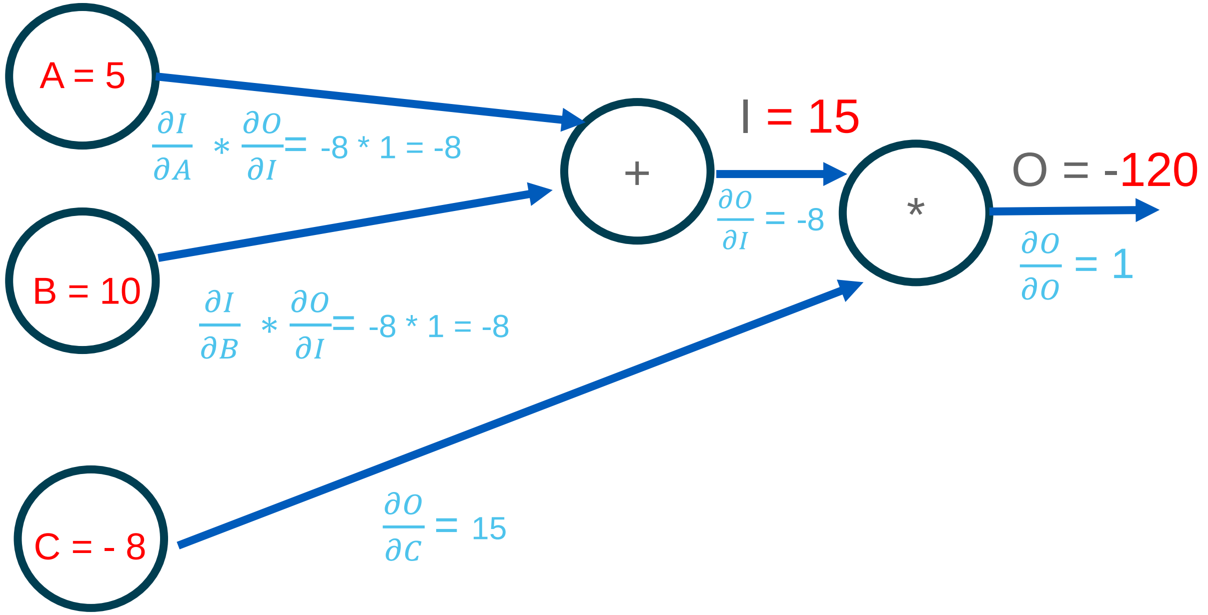
Computational Graph

- Let us look at another example



Computational Graph

- Now we need to find the gradient with respect to the output. It happens that multiplying these gradients (applying chain rule) is the solution.



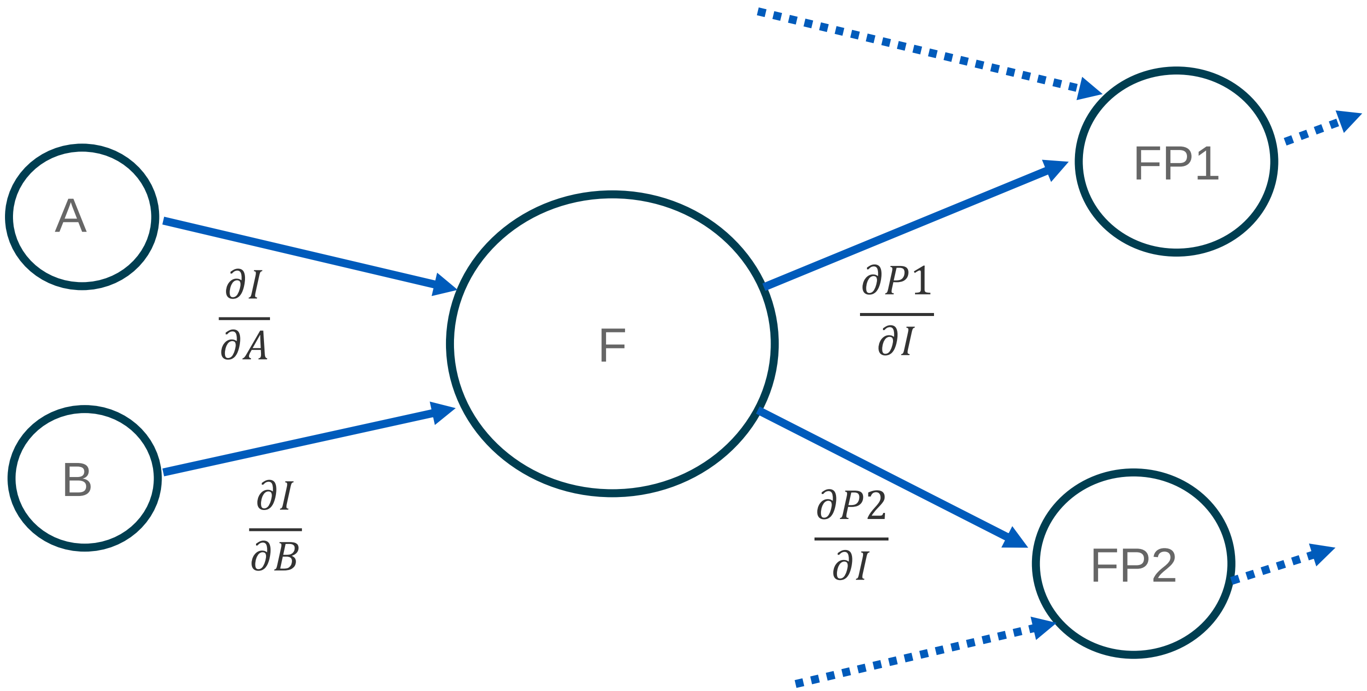
Computational Graph

- Some additional details

- Imagine if the intermediate result of the function is used in two different computations FP1 and FP2

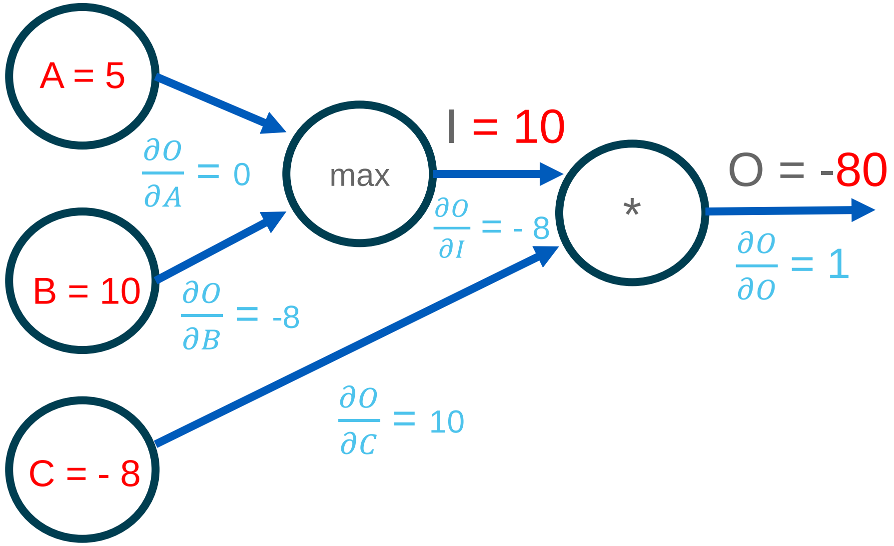
- In order to compute the gradient $\frac{\partial output}{\partial A}$ and $\frac{\partial output}{\partial B}$, we will sum up all the gradients at F

- $$\frac{\partial Output}{\partial A} = \frac{\partial I}{\partial A} * \left(\frac{\partial P1}{\partial I} + \frac{\partial P2}{\partial I}\right)$$
  $$\frac{\partial Output}{\partial B} = \frac{\partial I}{\partial B} * \left(\frac{\partial P1}{\partial I} + \frac{\partial P2}{\partial I}\right)$$



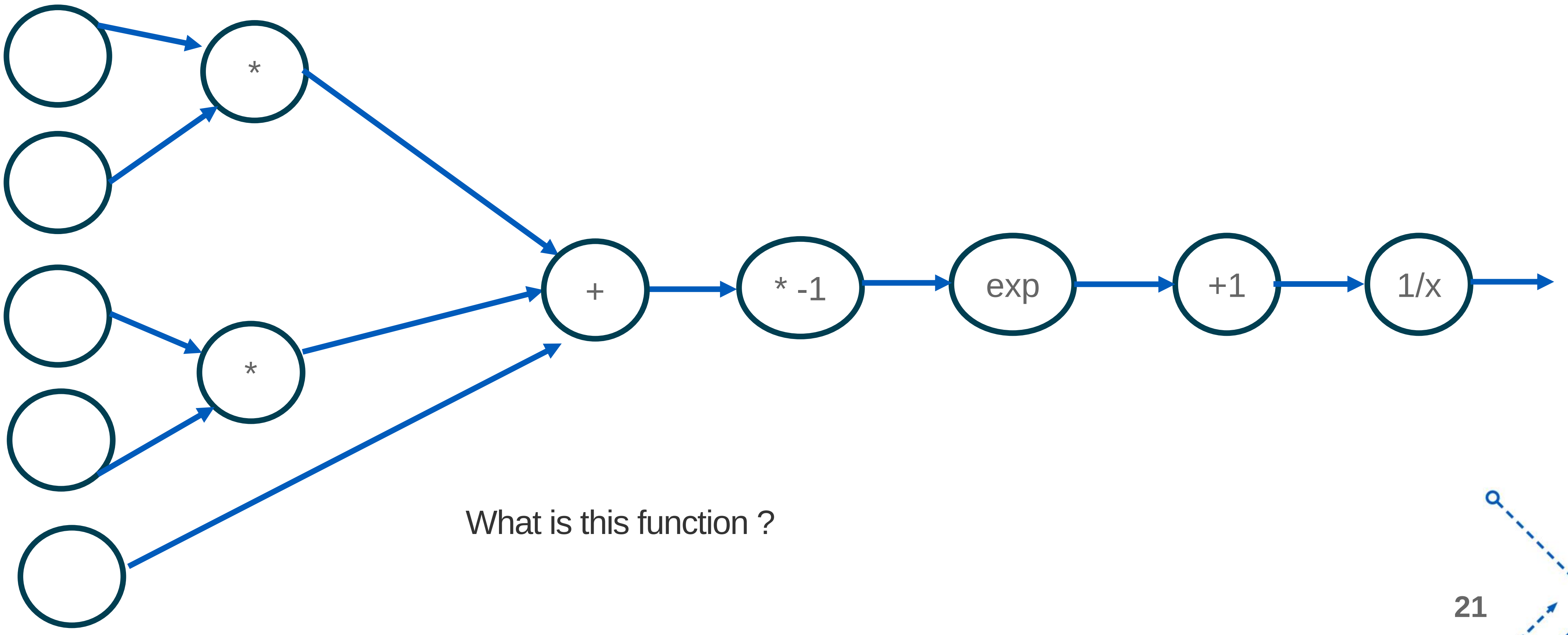
Computational Graph

- Let us look at another example



Computational Graph

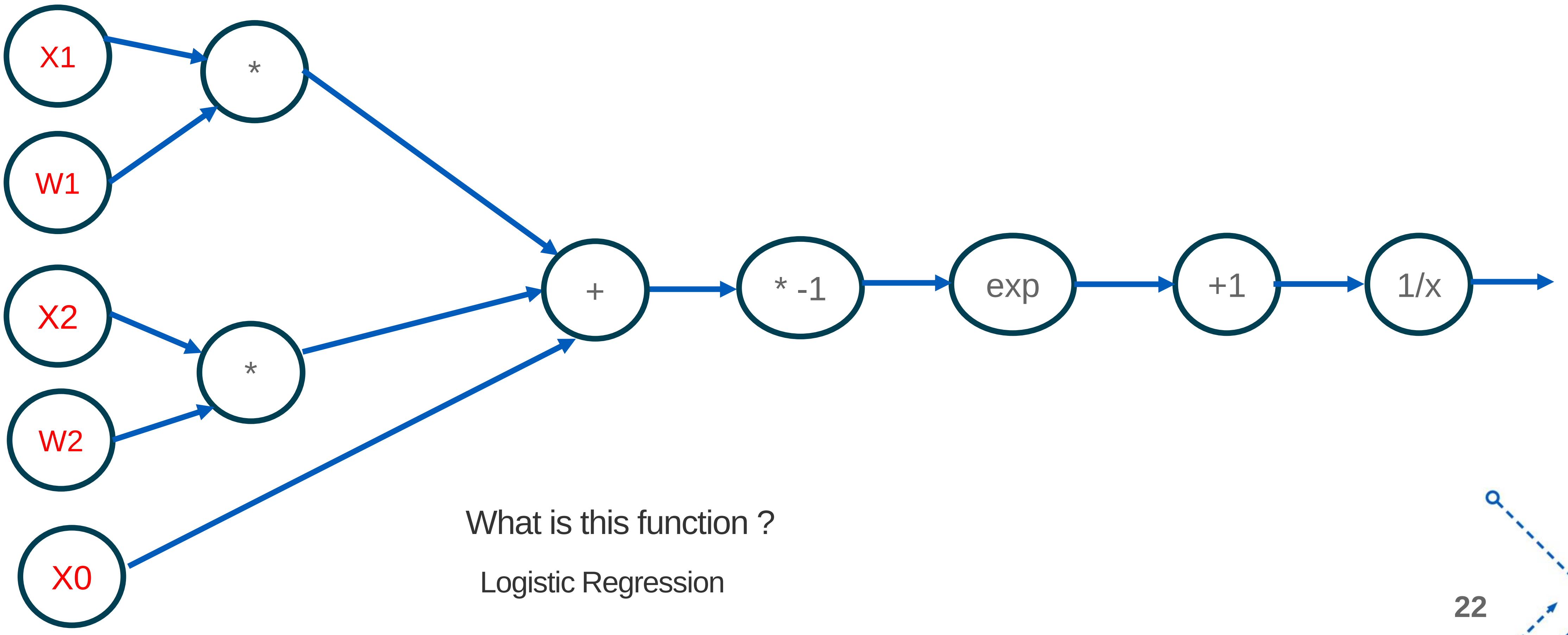
- Let us look at another example



What is this function ?

Computational Graph

- Let us look at another example

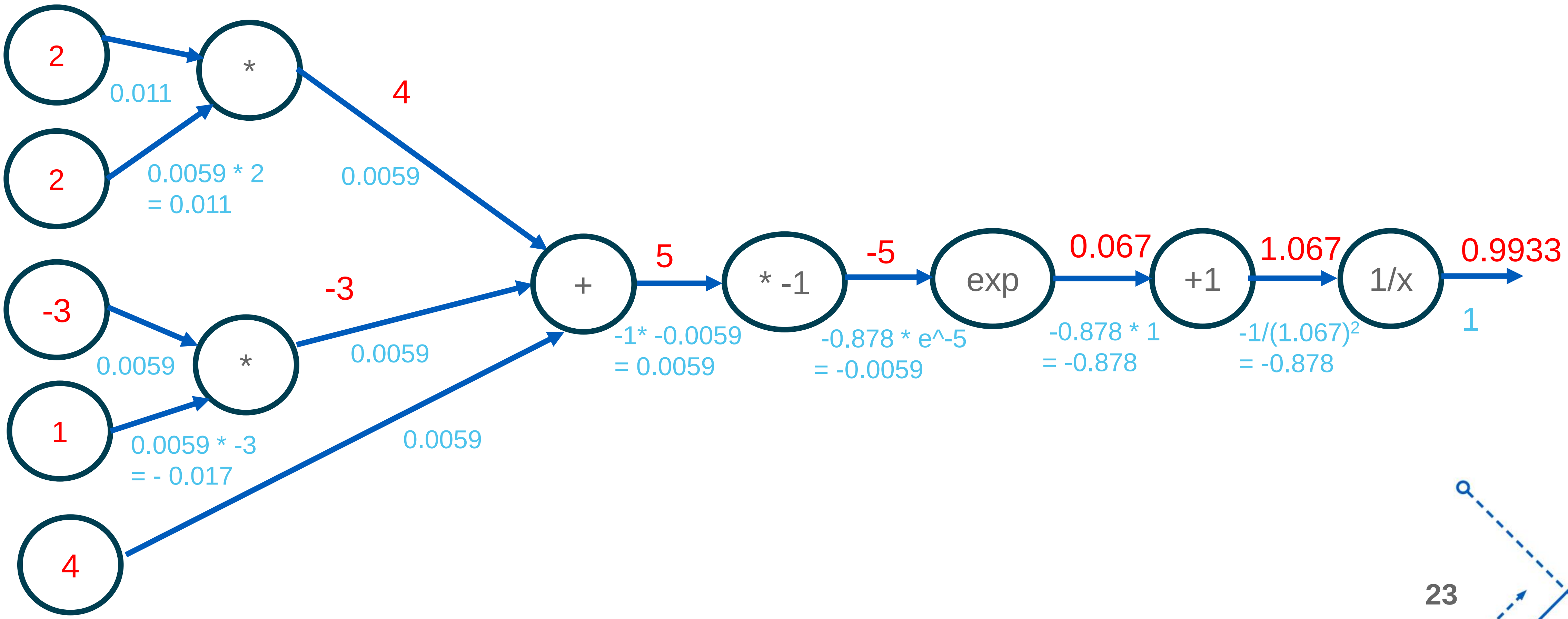


What is this function ?

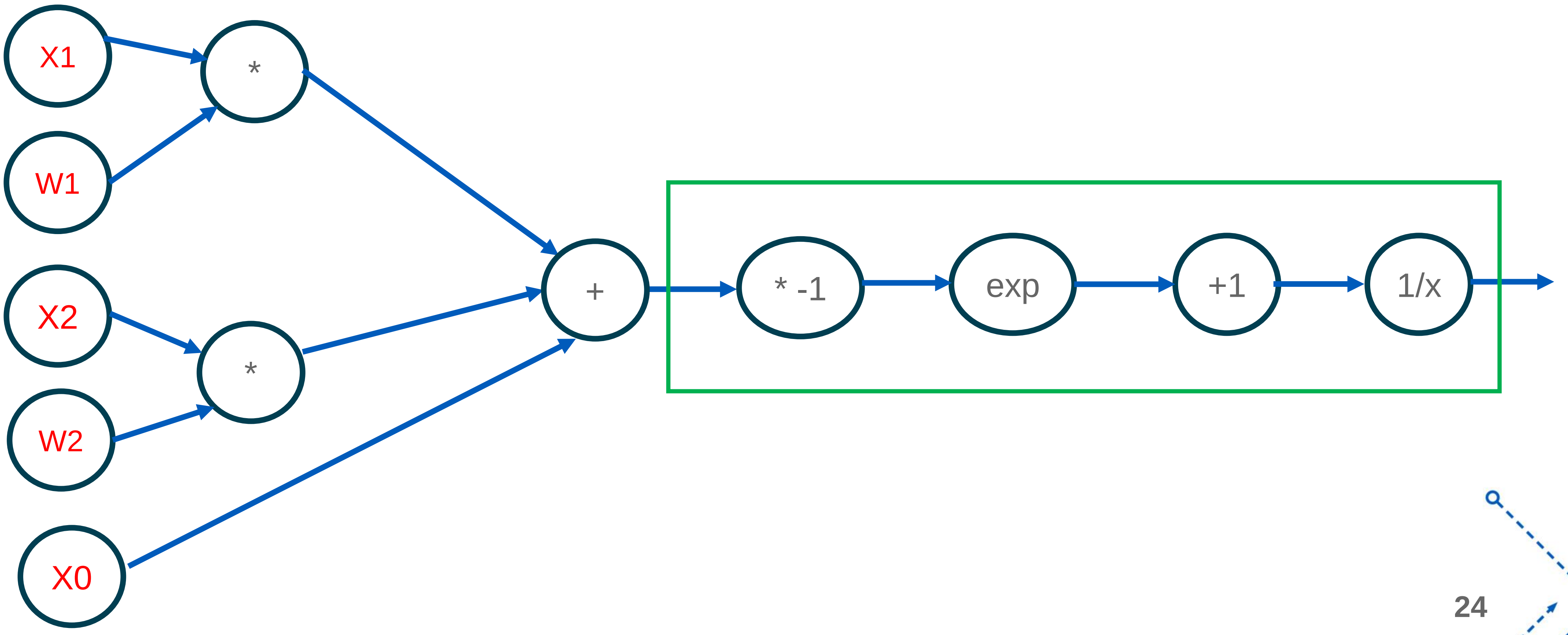
Logistic Regression

Computational Graph

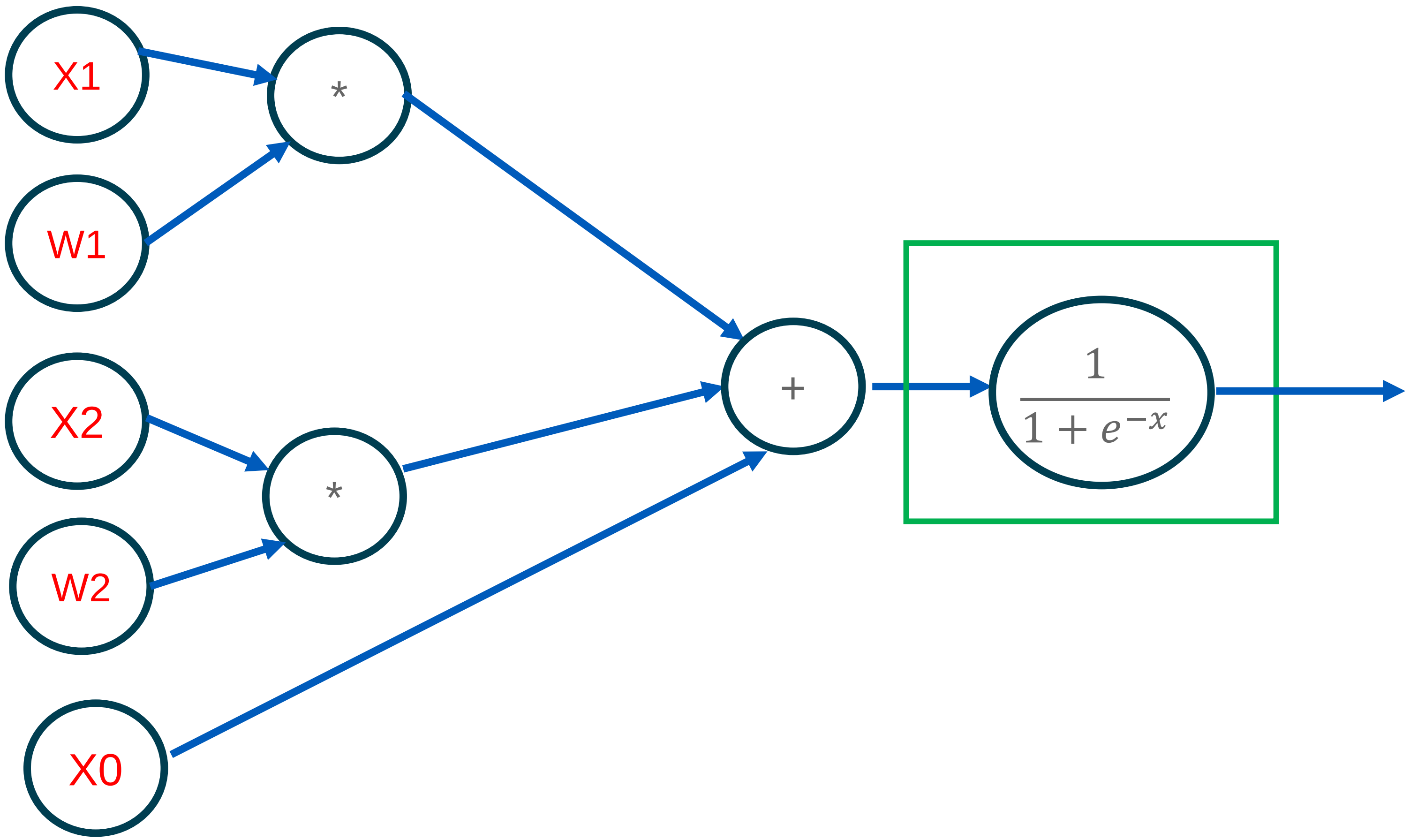
- Let us look at another example



Computational Graph



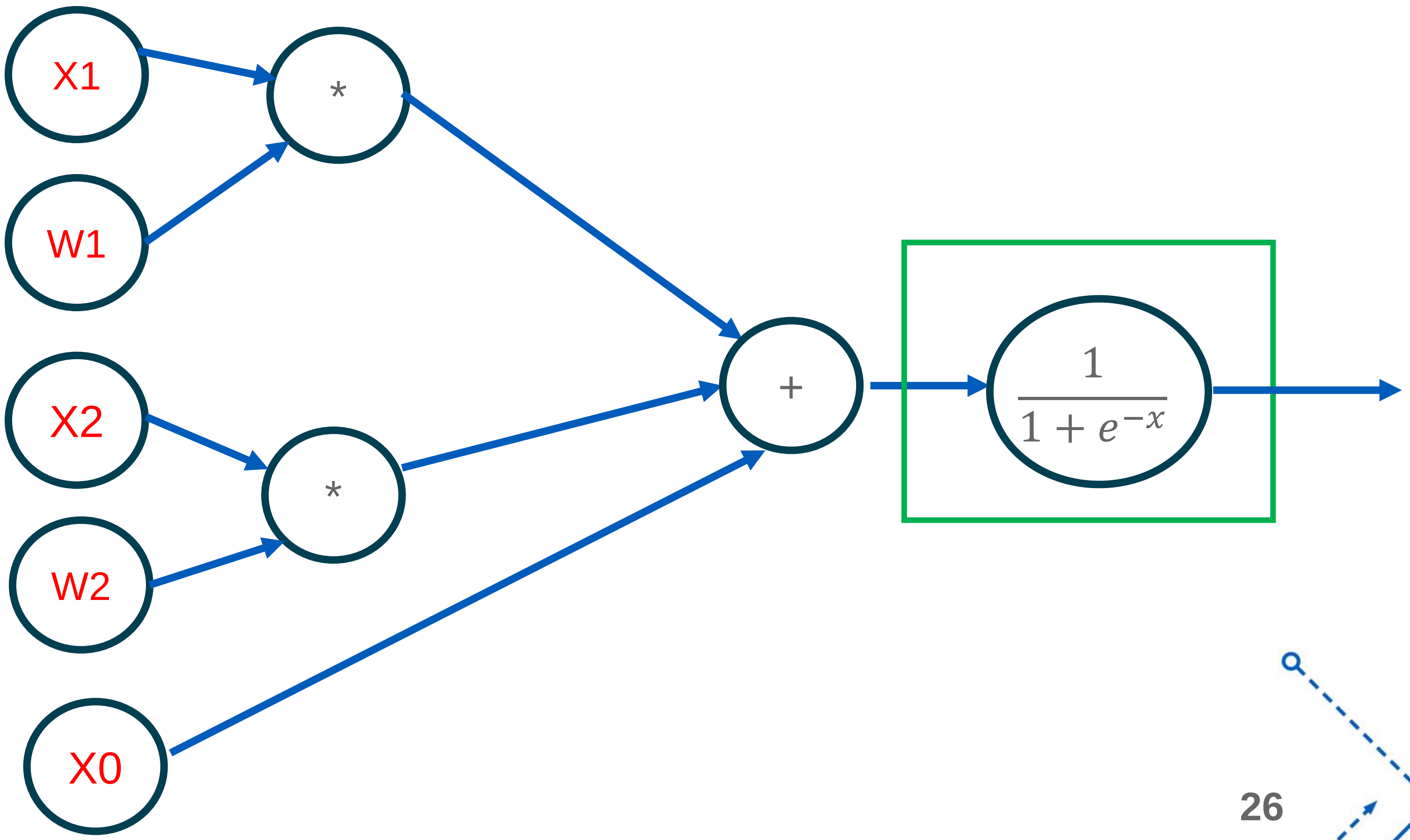
Computational Graph



Computational Graph

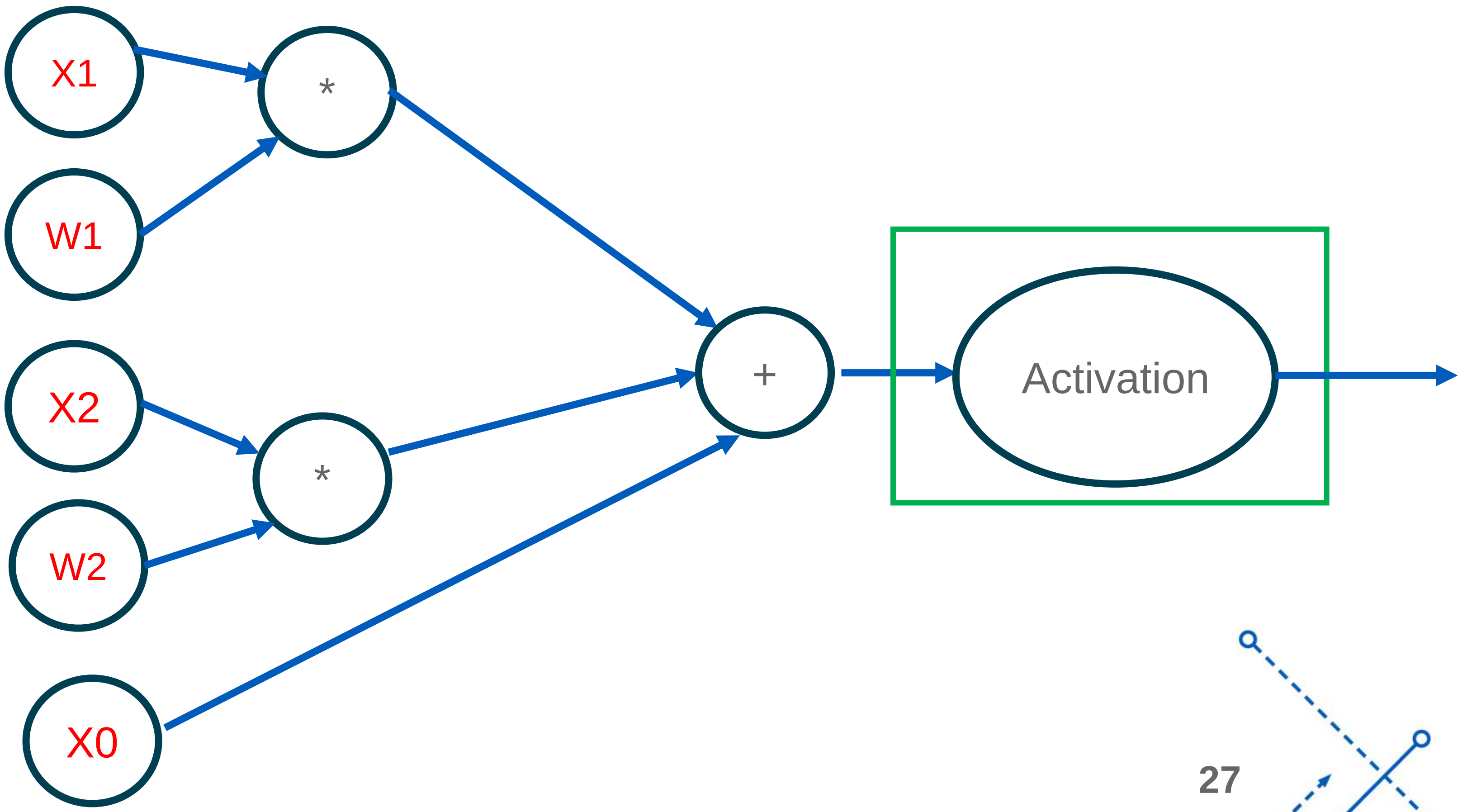
- The granularity of the nodes can be custom
- Need to find the derivative of the custom function
- For example, the derivative of $\sigma(x)$ is

$$= \sigma(x)\big(1 - \sigma(x)\big)$$



Computational Graph

- This is often called a perceptron model
- When training the model, there would be an expected output Y associated with each data point
- Objective/Loss function is designed to estimate the error associated with the prediction and the expected value

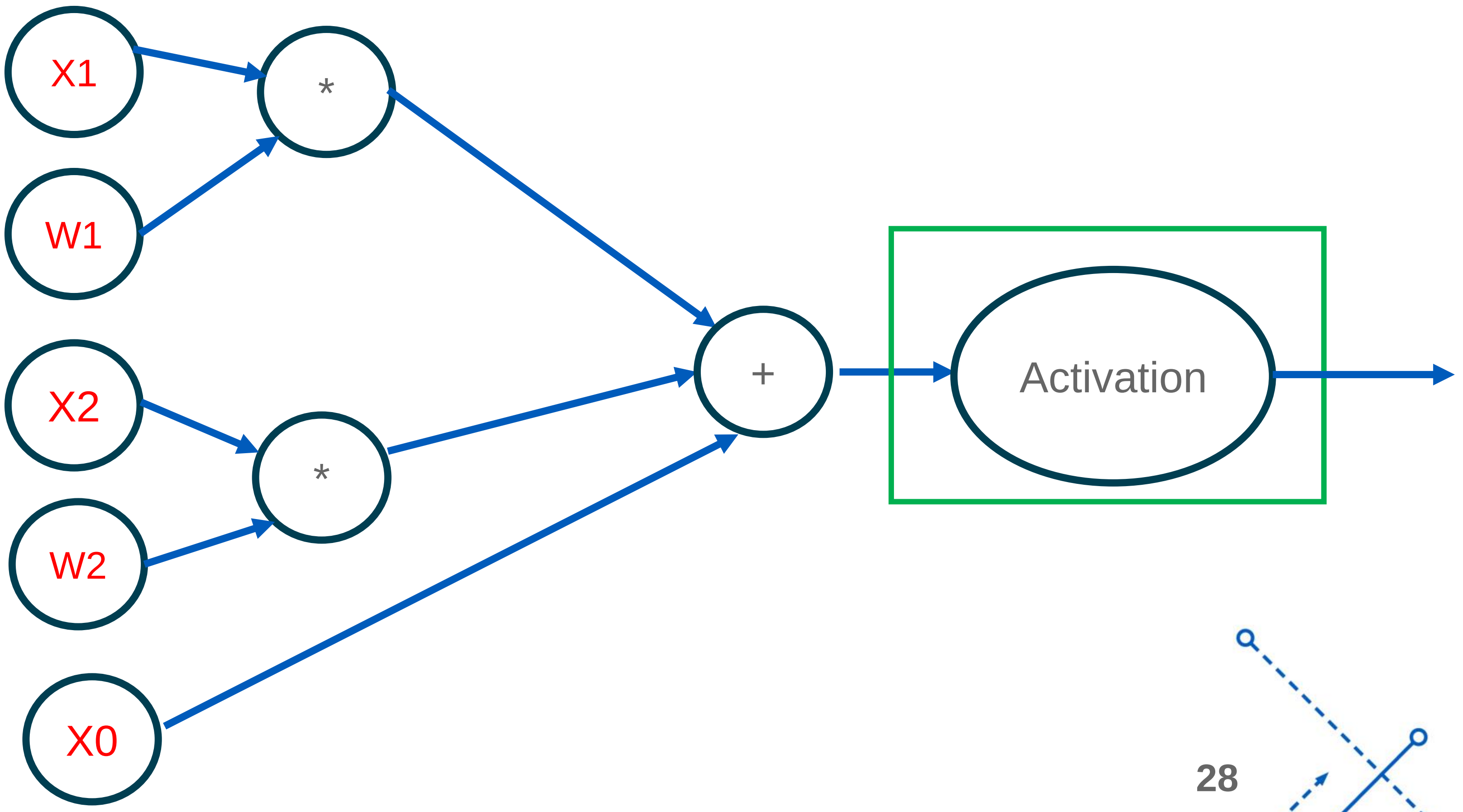


Computational Graph

- Some of the common loss functions you might know are MSE, Cross Entropy, Hinge etc.

- The derivative of the objective function with respect to the prediction is computed

- This is the first step in the backpropagation

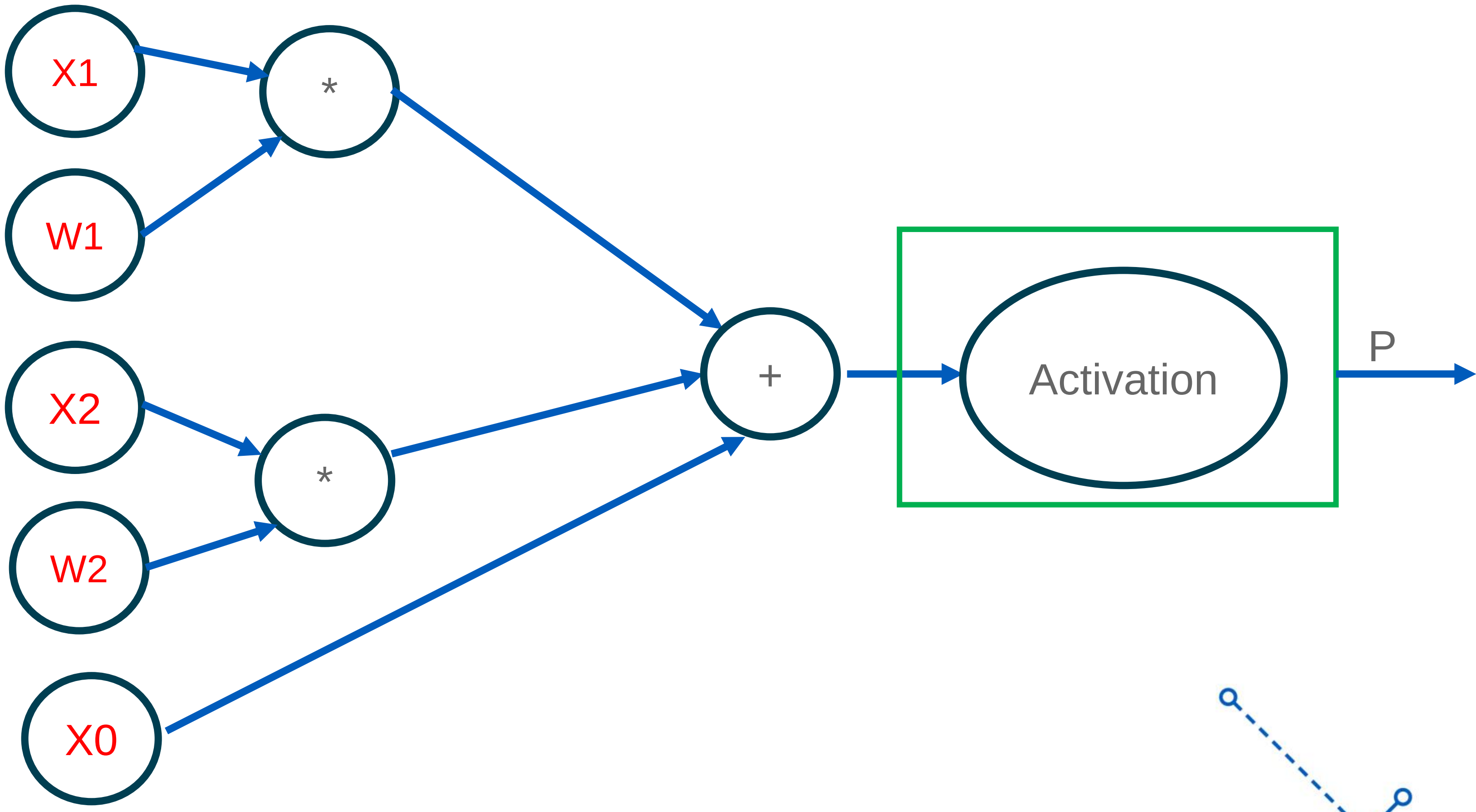


Computational Graph

- If we consider P as the prediction and MSE loss
- The loss formulation would be

  $J = \frac{1}{N} \sum_{i=1}^{N} (y - p)^2$

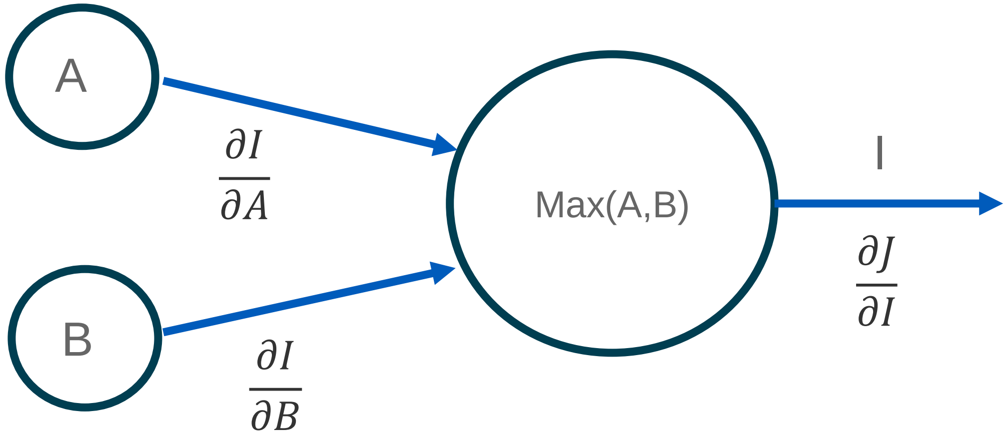
- The first derivative found during the backward pass is $\frac{\partial J}{\partial p}$



Computational Graph

- Let us look at some code

```
class MaxNode:

    def forward(A,B):

        I = max(A,B)
        return I

    def backward(dI):

        dA = ..... # derivative of I with respect to A * dI
        dB = ..... # derivative of I with respect to B * dI

        return [dA,dB]
```



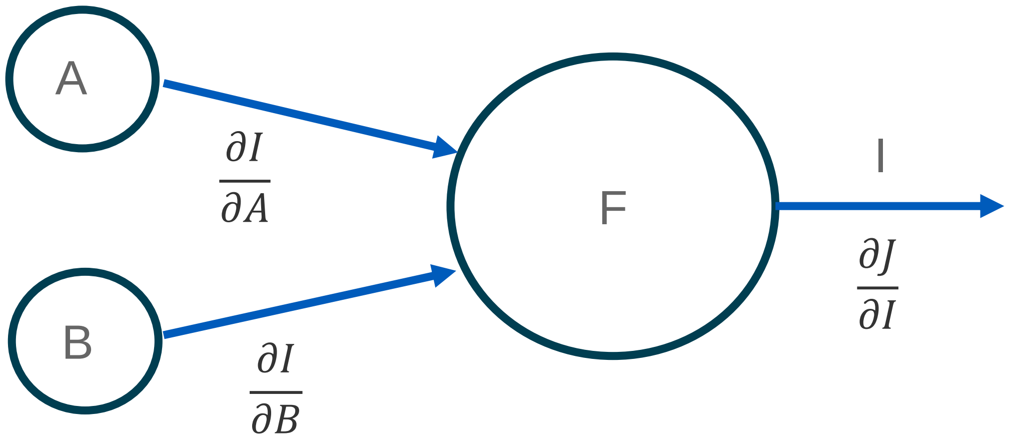
Computational Graph

- We need one more thing to get it to work

```
class GraphNet:

    def __init__(self,graphnodes):
        self.graphnodes = graphnodes # Some representation of nodes in the computational graph
        ...
        ...
        ...

    def forward(input):
        for eachNode in topologically_sorted(self.graphnodes):
            eachNode.forward()
            ...
            ...  # logic to manipulate the output produced in the forward pass of each node
            ...
        return loss

    def backward():
        for eachNode in reversed(self.topologically_sorted(self.graphnodes)):
            eachNode.backward()
            ...
            ... # chain the gradients produced in the backward of each node
            ...

        return gradients
```

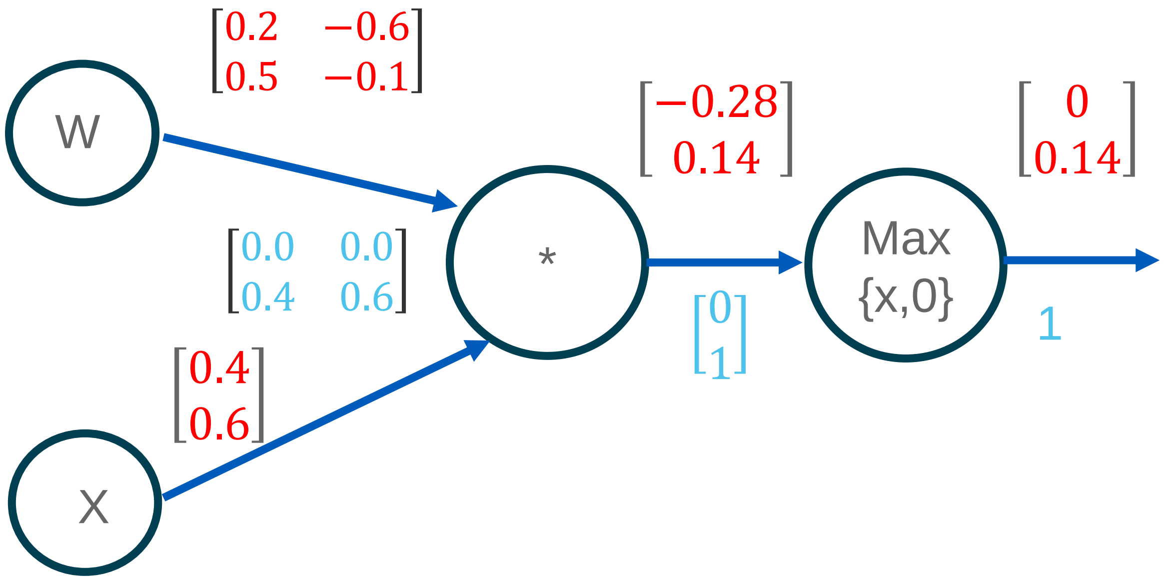
Computational Graph

- The examples we saw until now consider X and W to be scalars
- But in real computations, X and W are vectors and the local gradients are Jacobians
- Jacobians are matrices
- The final computation for chained gradient will be a vector-matrix multiplication.



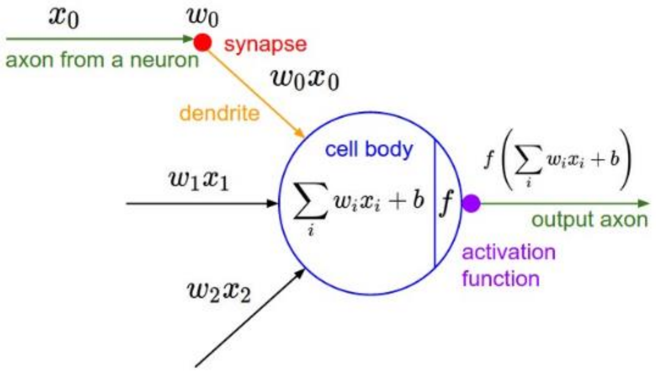
Computational Graph

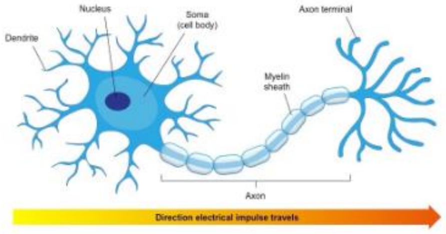
- Let us look at an example with matrices



Artificial Neural Networks

- Inspired from the biological neural network in the brain
- Perceptron architecture was inspired from the working of a neuron



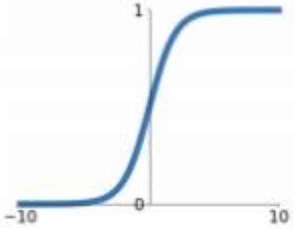


Artificial Neural Networks

- Today there are many activations

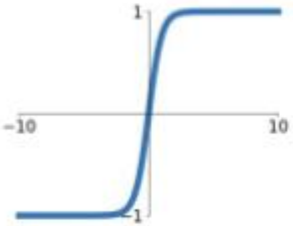
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



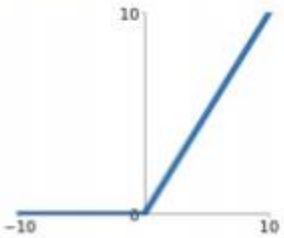
tanh

$\tanh(x)$



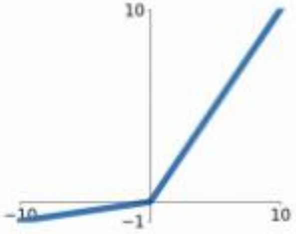
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

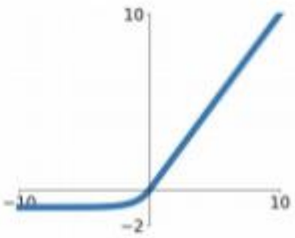


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

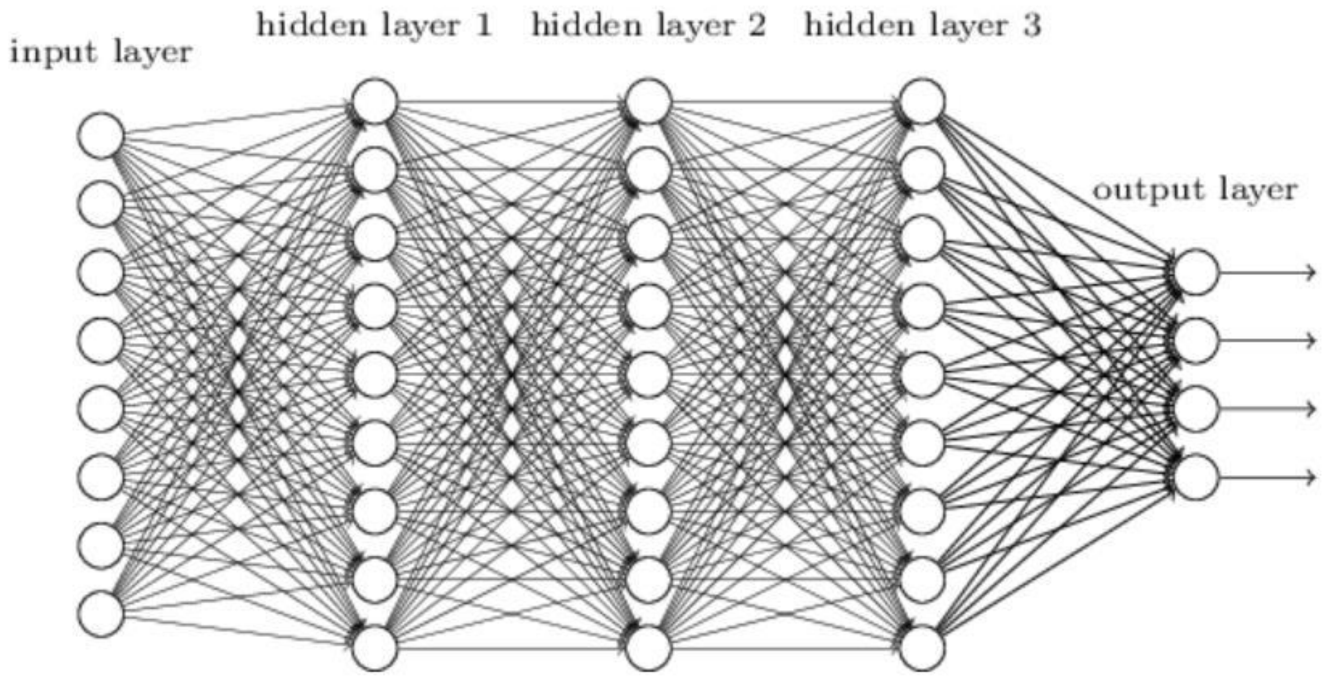
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Artificial Neural Networks

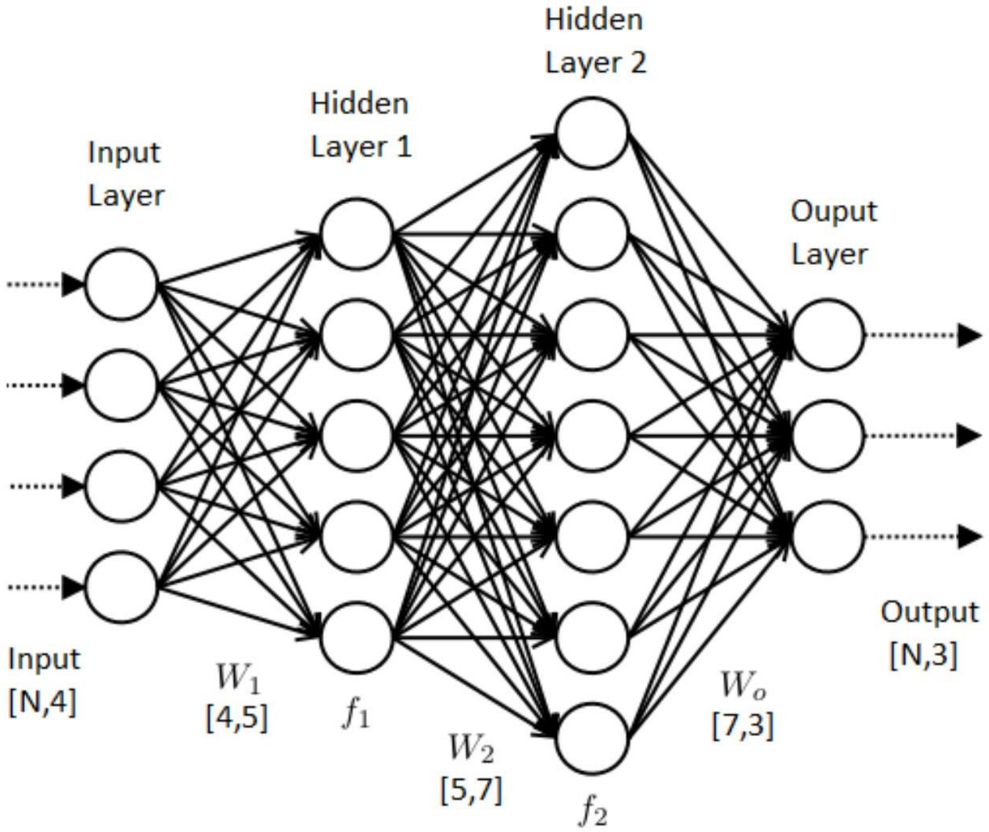
- Lets look at how the neural network is setup



- Why is this setup like this?

Artificial Neural Networks

- A closer look



- Each column look like a stack of computational nodes. These stacks are called layers

Artificial Neural Networks

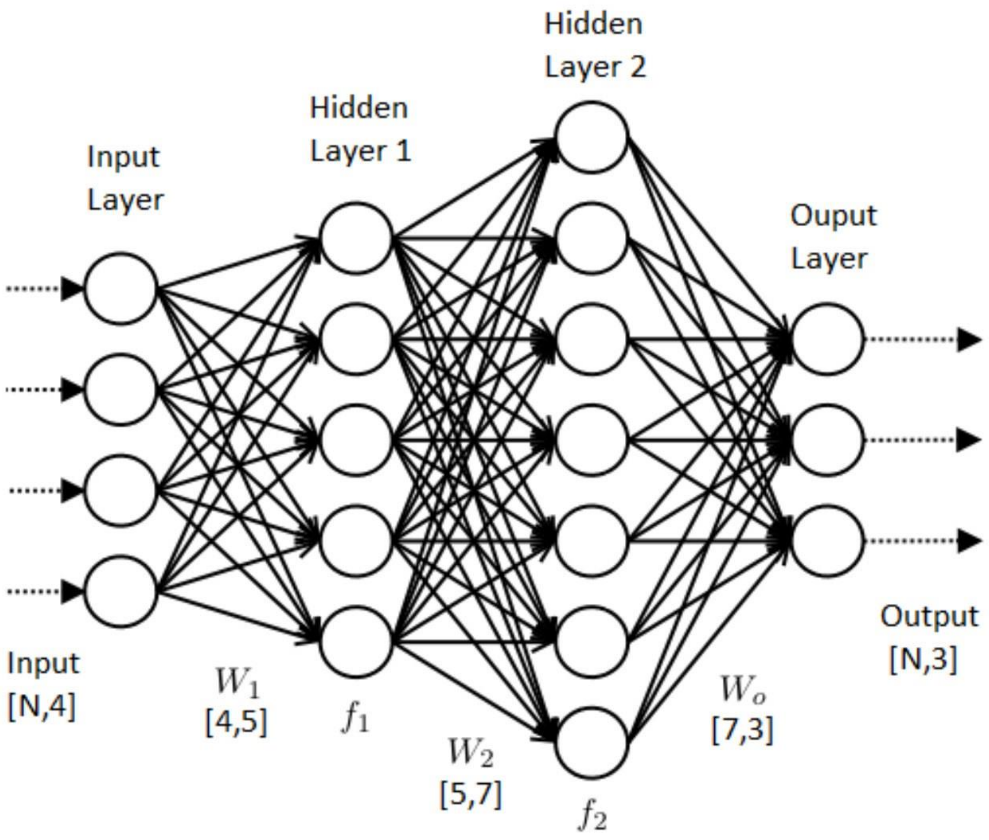
- This neural network can be mathematically written as

$$\text{Output} = w_0^T\left(f_2\left(w_2^T\left(f_1(w_1^T x)\right)\right)\right)$$

- Where f_1, f_2 are activation functions and w_1, w_2, w_0 are the parameters of the network

- This can be modelled as a computational graph

- In order to improve computational efficiency, the basic computational unit can be the layer



Artificial Neural Networks

- Let us look at some code

```
class GraphNet:

    def __init__(self,graphnodes):
        self.graphnodes = graphnodes # Some representation of layers in the computational graph
        ...
        ...
        ...

    def forward(input):
        for eachNode in topologically_sorted(self.graphnodes):
            eachNode.forward()
            ...
            ...  # logic to manipulate the output produced in the forward pass of each layer
            ...
        return loss

    def backward():
        for eachNode in reversed(self.topologically_sorted(self.graphnodes)):
            eachNode.backward()
            ...
            ... # chain the gradients produced in the backward of each layer
            ...

        return gradients
```

Artificial Neural Networks

- But now instead of the nodes we have layers

```
class BaseLayer:

    def __init__(self,):

        ...
        ...   # initialize some parameters common to all the layers
        ...

    def forward():
        pass

    def backward():
        pass
```

Artificial Neural Networks

- Then inherit the base class for implementing a layer with some specific functionality

```
class DenseLayer(BaseLayer):

    def __init__(self,input_param):

        ...
        ... # initialize the base class constructor
        ... # initialize layer specific parameters

    def forward(input):

        ...
        ... # perform foward propagation
        ...
        return layerOutput

    def backward(dz):

        ...
        ... # perform backprop
        ...
        return gradient
```

Artificial Neural Networks

Consider the derivative of error function with respect to the weight from the hidden unit j to unit l where $j = 1, 2, \cdots, M+1$ and $l = 1, \cdots, K$:

$$
\begin{aligned}
\frac{\partial J_i}{\partial w_{lj}^{(2)}} &= \frac{\partial J_i}{\partial o_l} \frac{\partial o_l}{\partial b_l} \frac{\partial b_l}{\partial w_{lj}^{(2)}} \\
&= \delta_l z_j
\end{aligned}
$$

On the other hand, the derivative of error function with respect to the weight from the p^{th} input to hidden unit j where $p = 1, 2, \cdots, D+1$ and $j = 1, \cdots, M$ can be computed as follow:

$$
\begin{aligned}
\frac{\partial J_i}{\partial w_{jp}^{(1)}} &= \sum_{l=1}^{K} \frac{\partial J_i}{\partial o_l} \frac{\partial o_l}{\partial b_l} \frac{\partial b_l}{\partial z_j} \frac{\partial z_j}{\partial a_j} \frac{\partial a_j}{\partial w_{jp}^{(1)}} \\
&= \sum_{l=1}^{K} \delta_l w_{lj}^{(2)} (1 - z_j) z_j x_p \\
&= (1 - z_j) z_j \left(\sum_{l=1}^{k} \delta_l w_{lj}^{(2)}\right) x_p
\end{aligned}
$$

Artificial Neural Networks

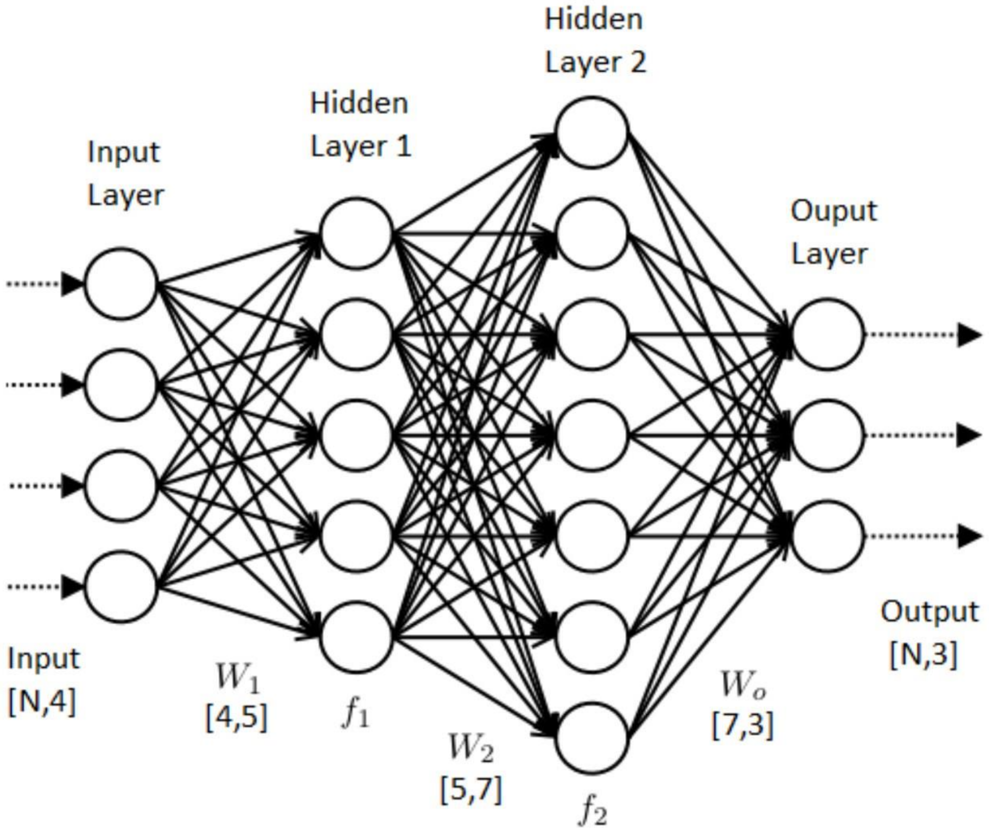
- Once the gradients with respect to the necessary parameters are computed, update step is performed

- $$w_{new} = w_{Old} - \eta \frac{\partial J}{\partial w_{Old}}$$

- Here η is the learning rate

- All the parameters are updated after the gradients are computed

- We will look at other methods of parameter update in a later lecture



Assignment 1

- Create a Toy Deep Learning framework called “PyTorchNano”.

- PyTorchNano should support the following layers
  - Dense Layer
  - Convolutional Layer
  - Max and Average Pooling Layer
  - Flatten Layer

- It should also support the following activations
  - ReLU
  - Softmax
  - Sigmoid

Assignment 1

- The user should be able to train the model with following loss functions:
  - Cross-Entropy
  - Hinge Loss
  - MSE

- The framework should allow the user to create a custom network architecture (not more than 10 layers deep) and train it on the MNIST dataset.

- The Implementation of Layers and Activation functions should be using classes having the forward and backward method.

- The Layers should be customizable (number of hidden neurons, filter size etc.)

- The framework should also provide the capability to measure the accuracy of the model.

Assignment 1

- Framework should provide capability to visualize filter responses of a convolutional layer
- Bonus points will be awarded if the framework has Batch Norm Layer and Dropout Layer implemented.
- The implementation should not use any Deep Learning frameworks like Pytorch, Keras , Tensorflow etc.
- More specific details will be mentioned in the assignment document, which will be released before next class
- The numbers of members in a team might be from 1-2 depending upon final enrollment


ANY QUESTIONS ?

References

- This lecture is inspired from cse 231n https://www.youtube.com/watch?v=i94OvYb6noo&t=2051
- http://neuralnetworksanddeeplearning.com/chap5.html
- https://mlcourse-ub.readthedocs.io/en/latest/
- https://developer.nvidia.com/blog/recursive-neural-networks-pytorch/
- http://cs231n.stanford.edu/
- https://www.google.com/imgres?imgurl=https%3A%2F%2Fmiro.medium.com%2Fmax%2F1400%2F1*Di4V69e4gC16ooF6PZPt-A.png&imgrefurl=https%3A%2F%2Ftowardsdatascience.com%2Feverything-you-need-to-know-about-neural-networks-and-backpropagation-machine-learning-made-easy-e5285bc2be3a&tbnid=PFKzBNejYXM4hM&vet=12ahUKEwism4aItO3yAhVrqnIEHSHUCNkQMyhDegQIARBf..i&docid=OXeL--Z4fRwo6M&w=1250&h=1057&q=neural%20networks%20with%20math&hl=en&client=firefox-b-1-d&ved=2ahUKEwism4aItO3yAhVrqnIEHSHUCNkQMyhDegQIARBf