SDSU CS 549 Spring 2024 Machine Learning Lecture 8: Neural Networks

IRFAN KHAN

References

SDSU CS549 Lecture Notes by Prof Yang Xu, Spring 2023. Some slides used here

Coursera machine learning course by Dr Andrew Ng, Oct 2023

Introduction

- We start with Artificial Neural Networks (ANN) used for Regression and Logistic Regression
- These ANNs fall in the category of Feed-Forward Neural Networks (FNN)
  - In FNNs, flow is uni-directional => information in the model flows in only one direction—forward—from the input nodes, through the hidden nodes and to the output nodes, without any cycles or loops
  - In contrast recurrent neural networks, which we will cover later, have a bi-directional flow.
- Many of these concepts carry over to other types of neural networks, e.g. Convolutional Neural Networks used in Object Detection

Outline

Neural Network Architecture

Back Propagation Intro

Computational Graph

Computational Graph and Basic NN Code Examples

Activation Functions

Derivatives of Vectors and Matrices and Back propagation

Generalize Back propagation for Deeper ANNs Using Vectors and Matrices

Backpropagation Code Examples

Initialization

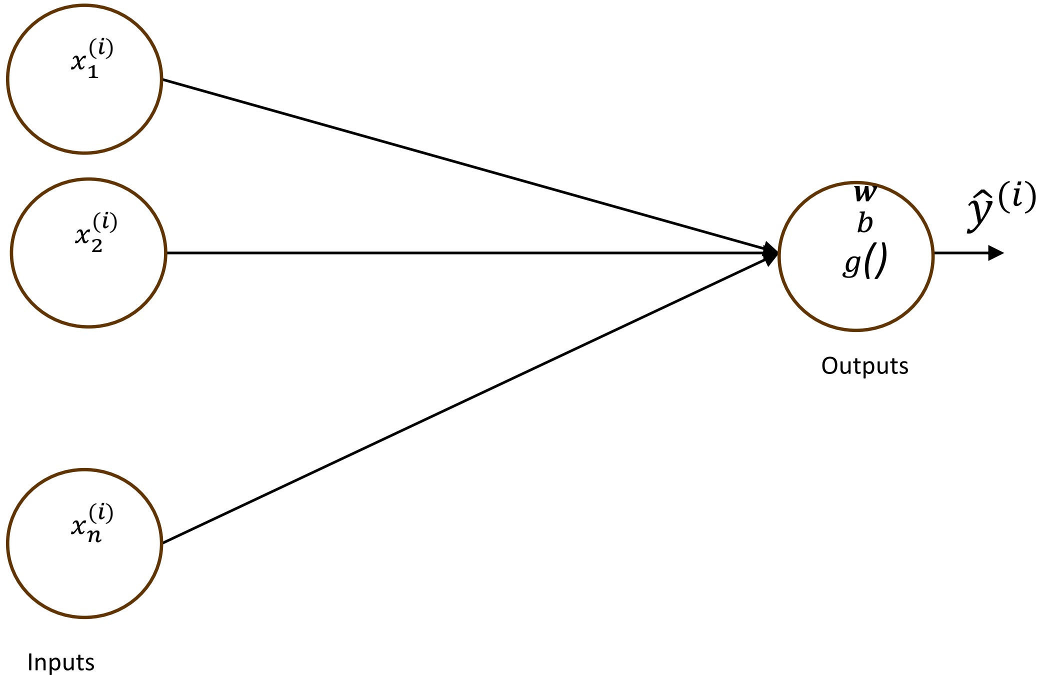
Neural Network Architecture

Recap – Linear Regression and Logistic Regression

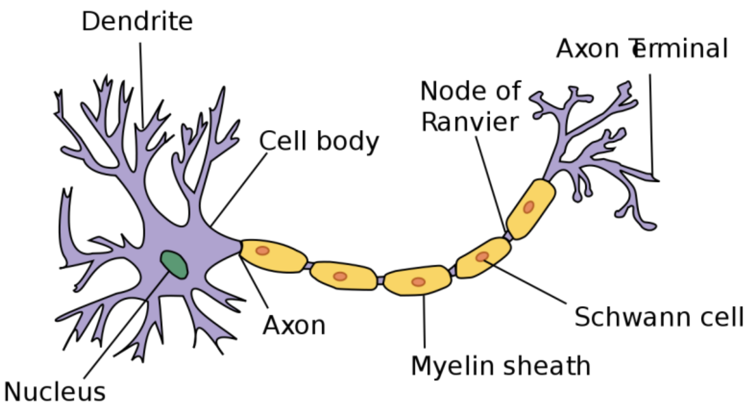
$$\begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} = g\left(\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_2^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_1^{(m)} & \cdots & x_1^{(m)} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix} \right)$$

g() = pass through function for Linear Regression

g() = Sigmoid function for Logistic Regression

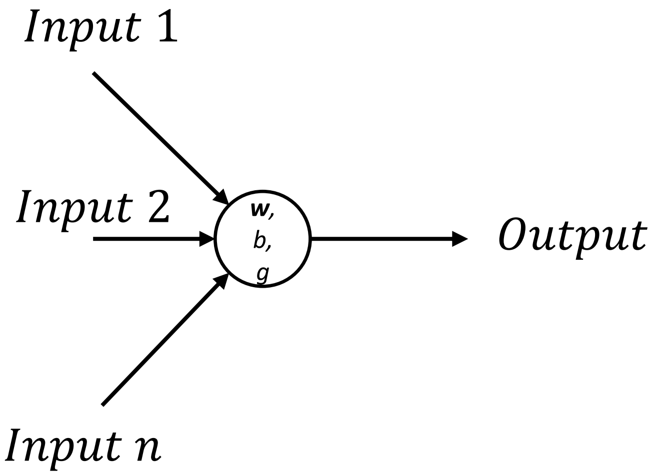


Artificial Neuron



Neuron

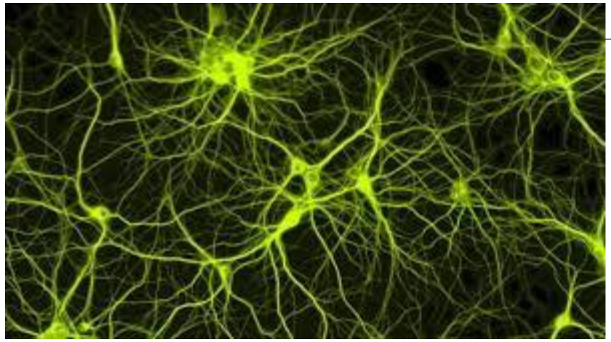
Inspired from biological neuron



Artificial Neuron

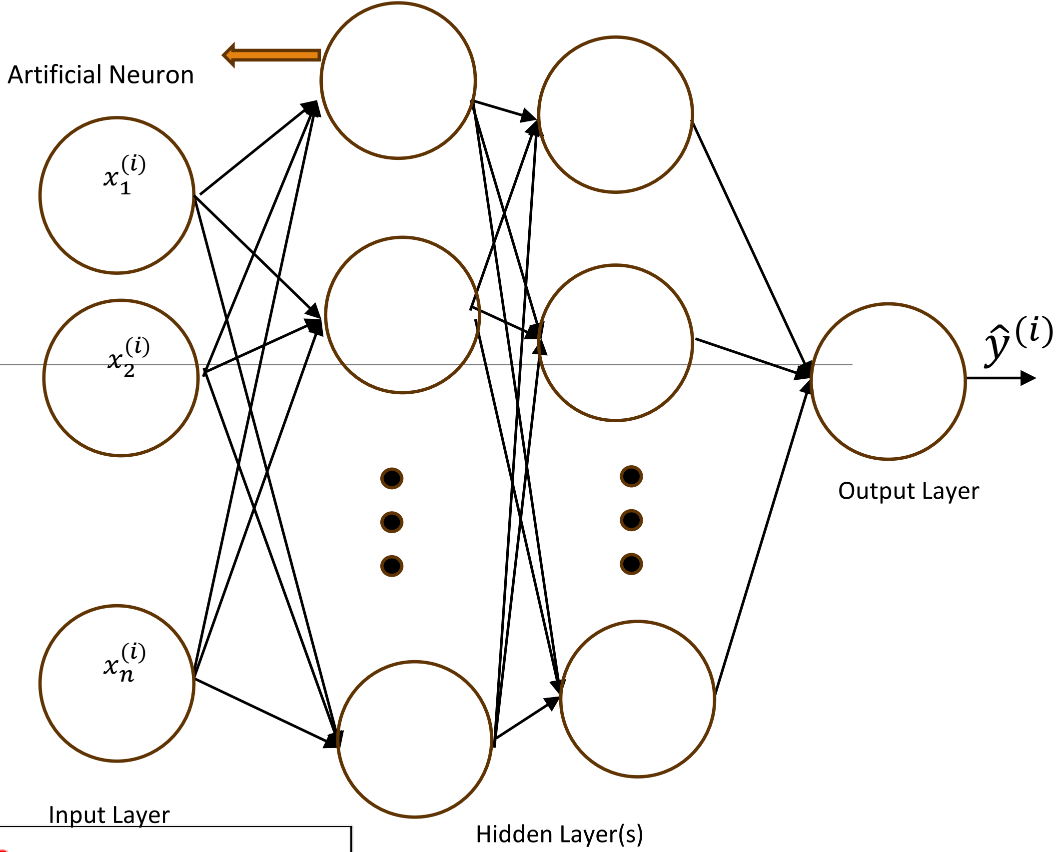
w: “weight”, *b*: “bias”, *g*: activation function applied to ***wx***+*b*

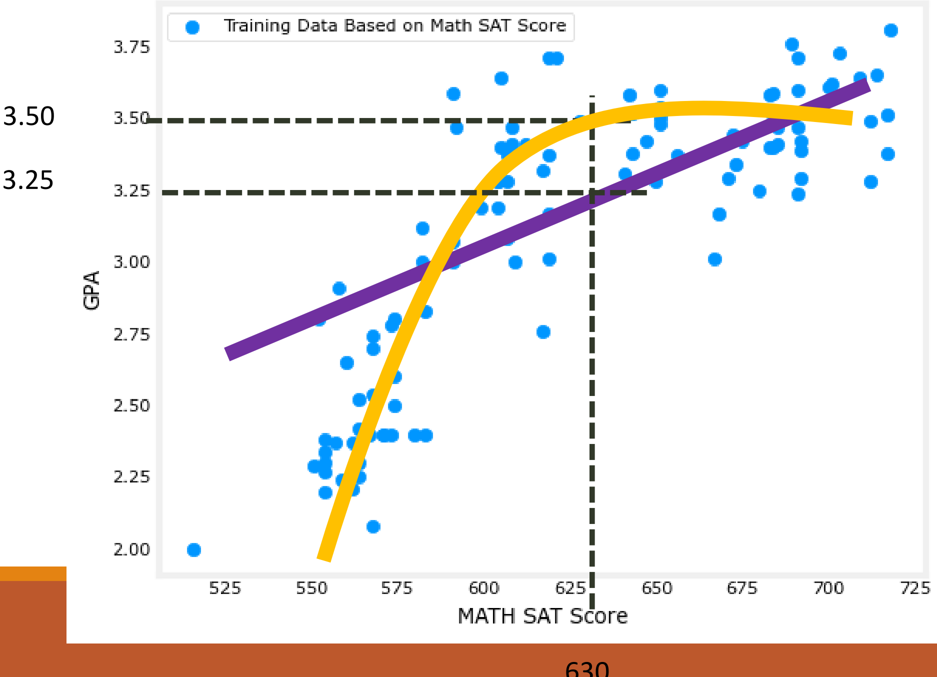
Artificial Neural Network

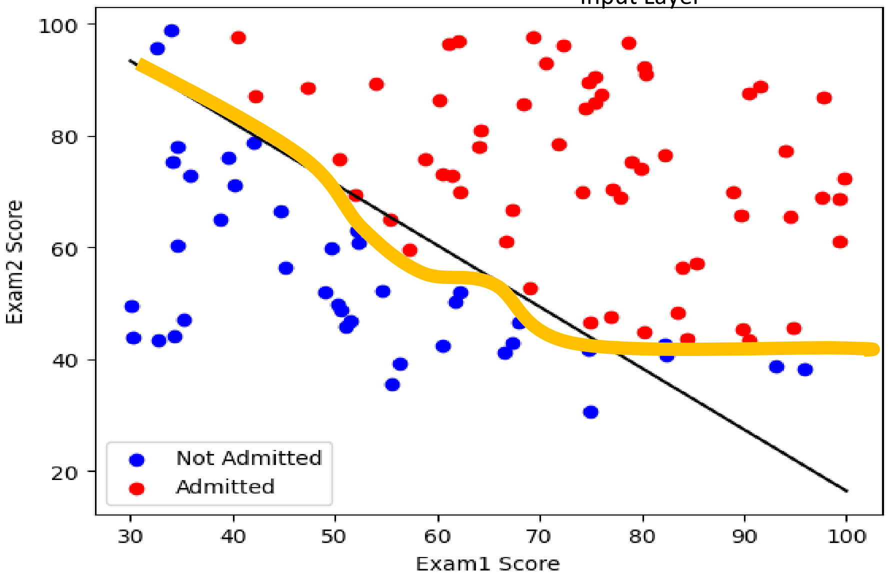


Neural Network: A dense interconnection of neurons

Inspired from biological neural network => ANN





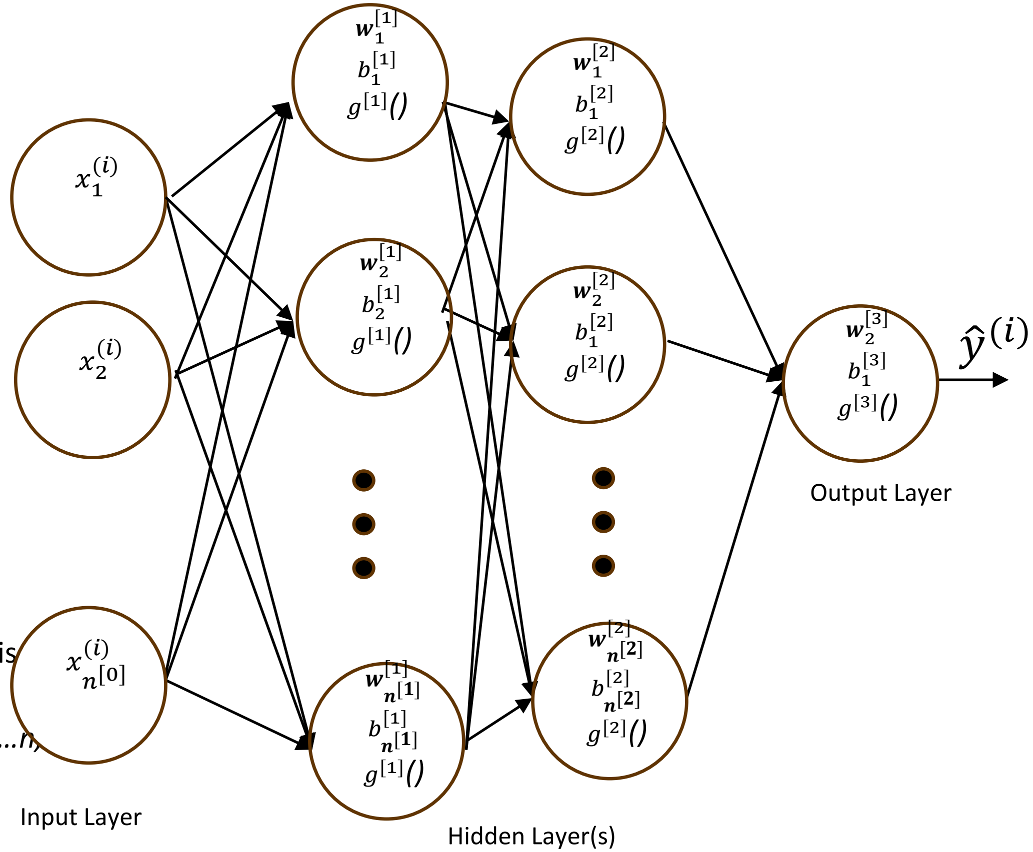


- Artificial Neural Network (ANN): A dense interconnection of neurons with *introduced non-linearities*
  - What happens if activation functions in the hidden layers are linear?
- Automatically creates more complex data fitting functions like the orange curve on the left
- The more the number of hidden layers, the “deeper” the ANN

ANN - Parameters

- Layers in a neural network are composed of neurons
  - $l = 0,1,2,\ldots L$ where *l=0* denotes the input layer and *L* denotes the output layer.
  - So total number of "hidden" layers = *L-1*
- $n^{[l]}$ represents the number of neurons in the *l*th layer
- $w_{ij}^{[l]}$: weight parameter in the i^{th} neuron in layer *l* applied to output of the j^{th} neuron of the previous layer, *l=1,2,..L*
  - Define the row vector at the i^{th} neuron in layer *l* as: $\boldsymbol{w}_i^{[l]} = \begin{bmatrix} w_{i1}^{[l]} & w_{i2}^{[l]} & \cdots & w_{in_{l-1}}^{[l]} \end{bmatrix}$
- $b_i^{[l]}$: bias parameter in the i^{th} neuron in layer *l* , *l=1,2,..L*
- $g^{[l]}$*()*: Activation function at all neurons in layer *l*
  - Activation functions help create non-linearities
  - Number of choices are available, e.g. ReLU
- For the output layer:
  - For regression, a pass-through (linear) function is used
  - For classification, sigmoid layer is used

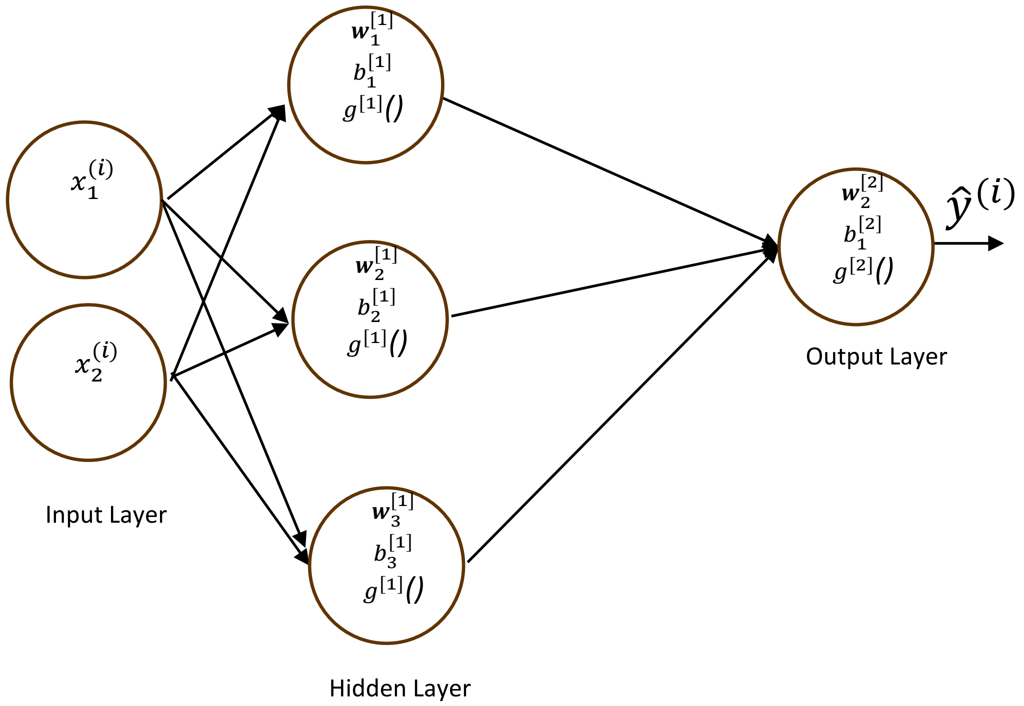
- Layer *0* consists just of the input data and no weights or bias or activation function are associated with this layer
- As usual, input data is represented by $x_j^{(i)}$, where *i = 1,2,…m, m* being the number of samples, *and j= 1,2,…n, n* being the number features in the input data
- Layer *L*, the output layer, predicts the output, $\hat{y}$. So, only one neuron in the output layer.
  - In general, $\hat{y}$ can be a vector with multiple dimensions we may want to predict, in which case the output layer also will have multiple neurons, one for each output dimension



Total Parameters = $\sum_{l=1}^{L} n^{[l]}\, n^{[l-1]} + n^{[l]}$

Total Parameters for an ANN

- $L = 2$, $n = 2$ for the simple/"shallow" ANN shown on the right
- Total Parameters $=\sum_{l=1}^{L} n^{[l]}\, n^{[l-1]} + n^{[l]}$ = 3*2+3 +1*3+1 = 13
- Number of parameters can grow rapidly with "deep" ANNs
- ResNet-50: deep CNN with over 23 million parameters, designed for image classification tasks.
  - We will cover CNNs in a later lecture
- One metric for HW hosting neural networks is TOPs (trillions of Operations per second!)
  - Entry level Nvidia A2 GPU: 36 INT8 TOPs
  - https://www.nvidia.com/content/dam/en-zz/solutions/data-center/a2/pdf/a2-datasheet.pdf
- How is all this possible?
  - Matrix manipulation that can be parallelized in hardware



Matrix Operations and Parallelization Scope in Hardware

- $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$
  - *3 x 2*
- $B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}$
  - *2 x 3*
- *C=A*B =* $\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$

- *Using 'for' loops:*
- *for i in range (3):*

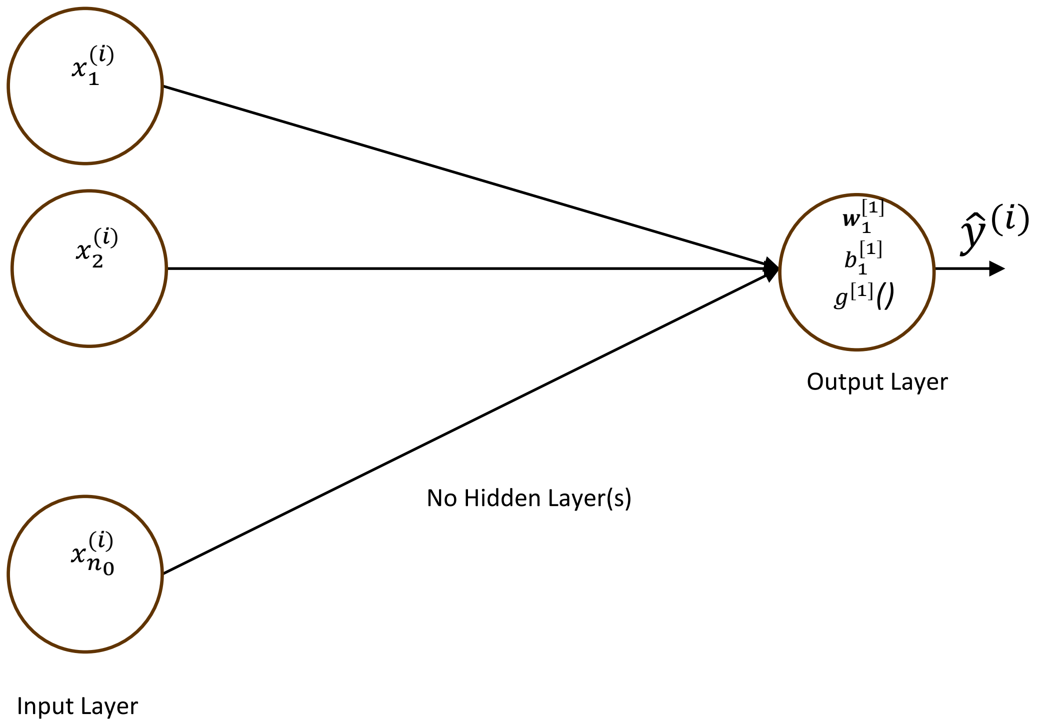
```
        for j in range (3):
                c[I,j]=0
                for k in range(2):
                        C[i,j]=C[i,j]+A[i,k]*B[k,j]
```

- *Using matrix function in NumPy*
- *C=np.dot (A,B)*

- All $a_{ik} * b_{kj}$ simple multiplications can be executed in hardware in parallel.
- All that remains next is a simple addition using terms computed above: $c_{ij} = \sum_{k=1}^{2} a_{ik} * b_{kj}$ which also can be computed in hardware in parallel for each element of C

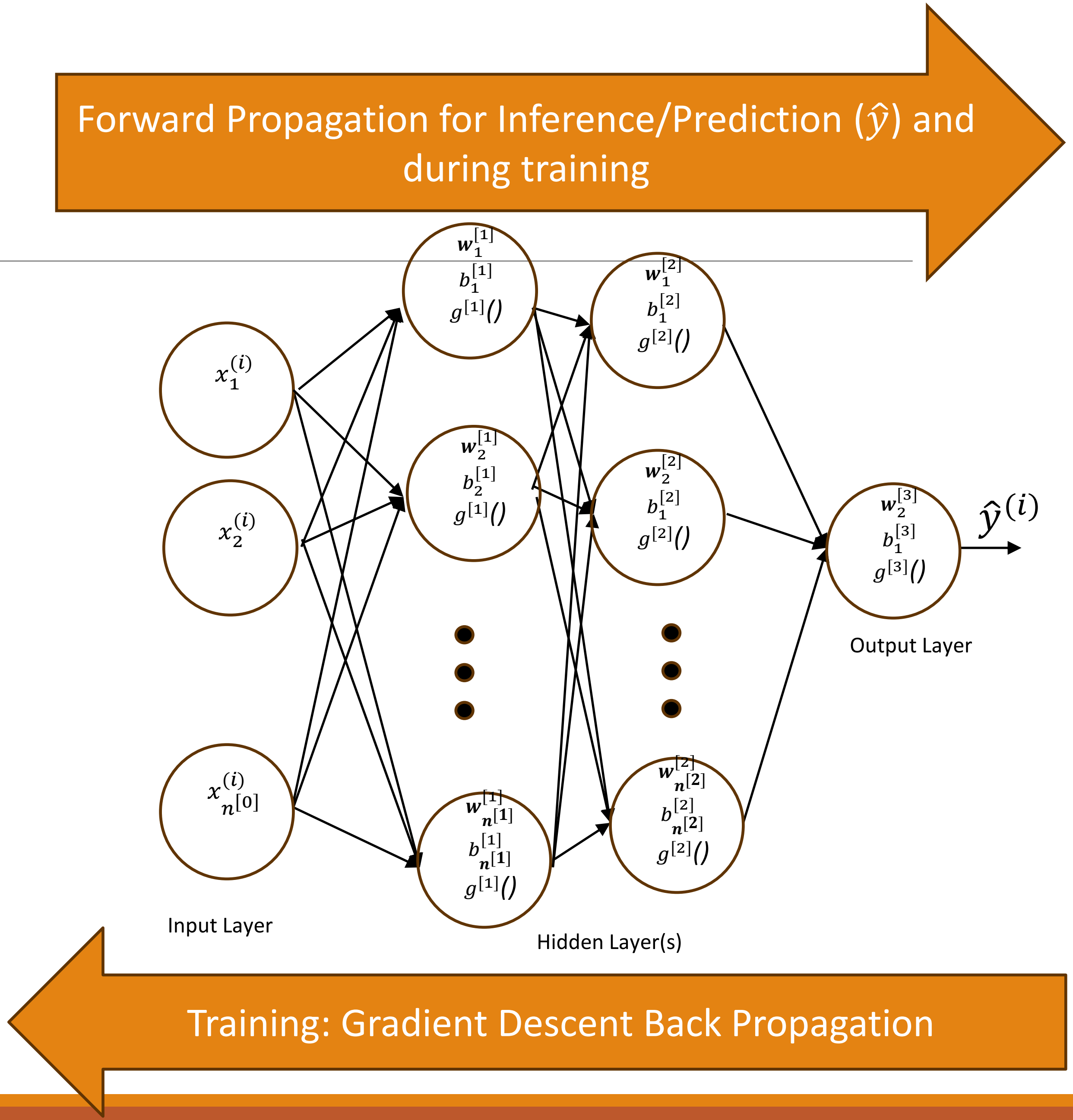
L=1 corresponds to Legacy Regression Models

- *Zero* hidden layers
- Total Parameters $=\sum_{l=1}^{L} n^{[l]}\, n^{[l-1]} + n^{[l]} = n_0 * 1 + 1 = n_0 + 1$



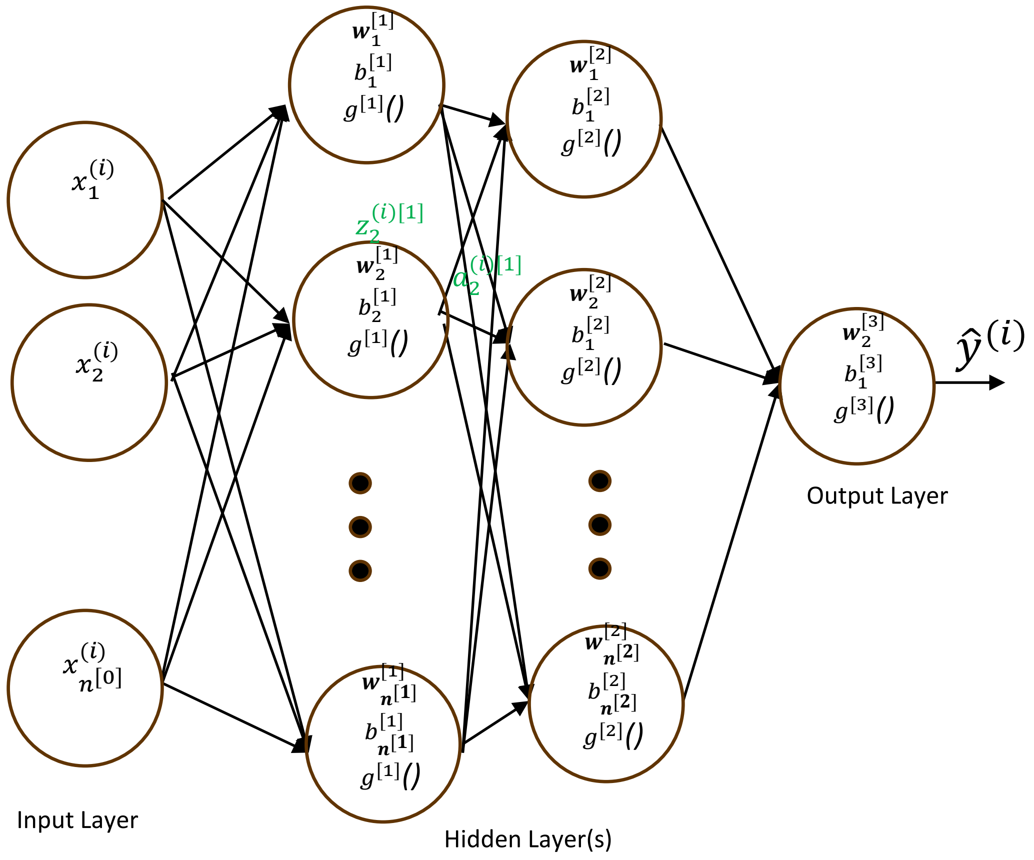
ANN in Practice

- How do we train the ANN => how do we learn values for $\boldsymbol{w}_j^{[l]}$ and $b_j^{[l]}$ which will minimize the cost function *J*?
  - Gradient Descent Back Propagation
  - Needs Forward Propagation to update parameters and to compute cost function, *J* iteratively
- Once trained, forward propagation, allows us to compute the prediction, $\hat{y}$
- Considerations:
  - How many hidden layers?
  - How many neurons/hidden layer?
  - Which activation functions to use?



Forward Propagation/Inference: Computing $\hat{y}^{(i)}$

- Introducing two new variables at neuron *j* in the *l**th* layer
  - $z_j^{(i)[l]}$ which computes an interim value after applying the weights and bias
  - $a_j^{(i)[l]}$ representing the output at neuron *j* after applying the activation function
- $z_j^{(i)[l]} = \sum_{k=1}^{n_{l-1}} a_k^{(i)[l-1]} w_{kj}^{[l]} + b_j^{[l]}, l = 1,2, \ldots L, j = 1,2, \ldots n^{[l]}$
- $a_j^{(i)[l]} = g^{[l]}(z_j^{(i)[l]})$
- $a_j^{(i)[0]} = x_j^{(i)} j = 1,2, \ldots n^{[0]}$
- $\hat{y}^{(i)} = a_j^{(i)[L]}(j = 1)$, for the single output case



Vector/Matrix Formulation for Computing $\hat{y}^{(i)}$

- Input data matrix: $X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} m \ x \ n$

- Output prediction: $\hat{y} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(m)} \end{bmatrix} m \ x \ 1$

- Weight matrix at the l^{th} layer: $W^{[l]} =$
$$\begin{bmatrix} w_{11}^{[l]} & w_{12}^{[l]} & \cdots & w_{1n^{[l-1]}}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & \cdots & w_{2n^{[l-1]}}^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[l]}1}^{[l]} & w_{n^{[l]}2}^{[l]} & \cdots & w_{n^{[l]}n^{[l-1]}}^{[l]} \end{bmatrix} n^{[l]} \ x \ n^{[l-1]}$$

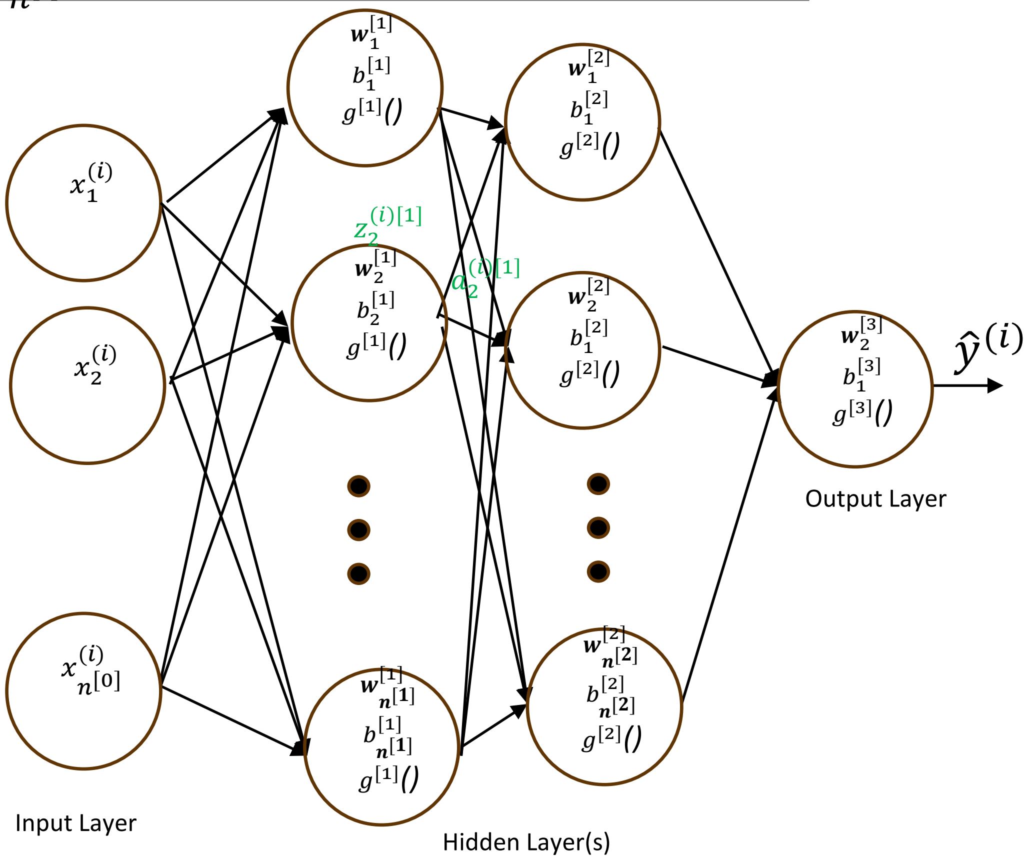
  - by convention used in PyTorch, could have been a $n^{[l-1]} \ x \ n^{[l]}$ matrix instead.

- Bias vectors at the l^{th} layer: $B^{[l]} = \begin{bmatrix} b_1^{[l]} & b_2^{[l]} & \cdots & b_{n^{[l]}}^{[l]} \end{bmatrix} 1 \ x \ n^{[l]}$

- Intermediate value at the l^{th} layer:

- $Z^{[l]} = \begin{bmatrix} z_1^{(1)[l]} & z_2^{(1)[l]} & \cdots & z_{n^{[l]}}^{(1)[l]} \\ z_1^{(2)[l]} & z_2^{(2)[l]} & \cdots & z_{n^{[l]}}^{(2)[l]} \\ \vdots & \vdots & \ddots & \vdots \\ z_1^{(m)[l]} & z_2^{(m)[l]} & \cdots & z_{n^{[l]}}^{(m)[l]} \end{bmatrix} m \ x \ n^{[l]}$

- Output values (activations) at the l^{th} layer: $A^{[l]}=$
$$\begin{bmatrix} a_1^{(1)[l]} & a_2^{(1)[l]} & \cdots & a_{n^{[l]}}^{(1)[l]} \\ a_1^{(2)[l]} & a_2^{(2)[l]} & \cdots & a_{n^{[l]}}^{(2)[l]} \\ \vdots & \vdots & \ddots & \vdots \\ a_1^{(m)[l]} & a_2^{(m)[l]} & \cdots & a_{n^{[l]}}^{(m)[l]} \end{bmatrix} m \ x \ n^{[l]}$$
- $A^{[0]} = X: m \ x \ n = m \ x \ n^{[0]}$
- $A^{[l]} = g^{[l]}(Z^{[l]}): \ m \ x \ n^{[l]}$
- $Z^{[l]} = A^{[l-1]} W^{[l]T} + B^{[l]}: m \ x \ n^{\{l]}$
  - Check matrix dimensions:
  - $m \ x \ n^{\{l]} = m \ x \ n^{\{l-1]} * n^{[l-1]} \ x \ n^{[l]} + m \ x \ n^{\{l]}$
- $\hat{y} = A^{[L]}: \ m \ x \ n^{[L]} : m \ x \ n^{[L]}$: $m \ x \ 1$ when there is only one output.

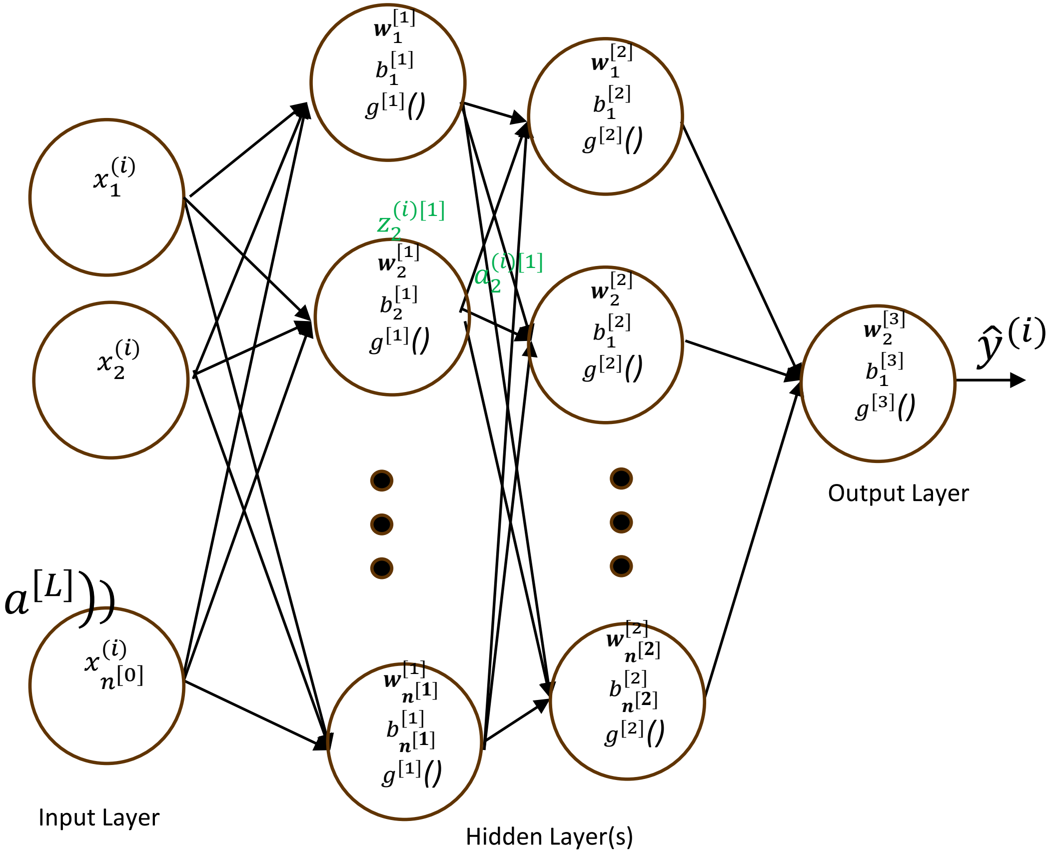


Computing the Cost Function J

- Let the sample data output $\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}^{(1)} \\ \boldsymbol{y}^{(2)} \\ \vdots \\ \boldsymbol{y}^{(m)} \end{bmatrix}$

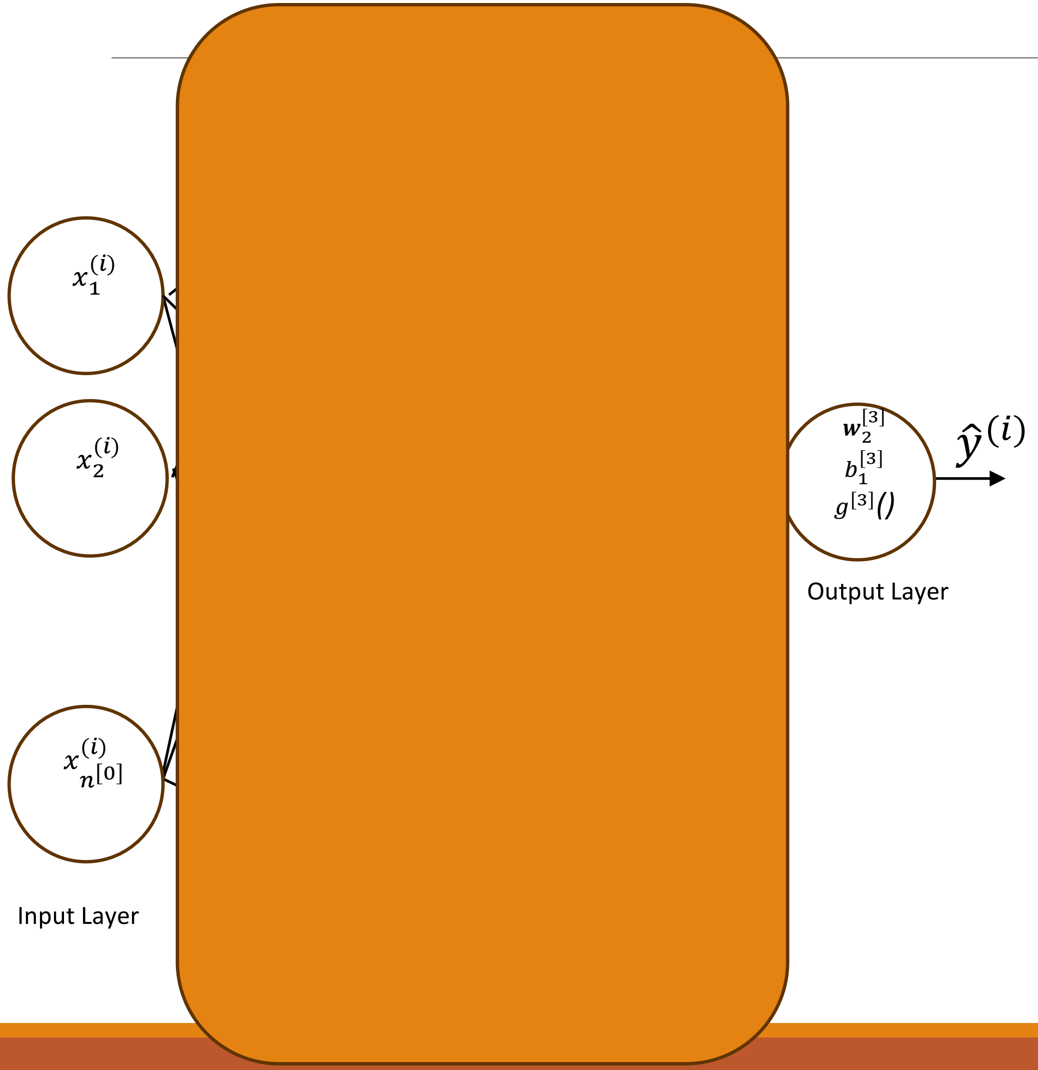
- For a regression problem,
  - $J = \frac{1}{2m}\sum_{i=1}^{m}(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)})^2 = \frac{1}{2m}(\boldsymbol{y} - \hat{\boldsymbol{y}})^T(\boldsymbol{y} - \hat{\boldsymbol{y}})$

- For a Logistic Regression problem,
  - $J = -\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{y}^{(i)} log(\boldsymbol{a}^{(i)[L]}) + (1 - \boldsymbol{y}^{(i)})\log(1 - (\boldsymbol{a}^{(i)[L]})$
  - $J = -\frac{1}{m}(Rowsum(y * \log(a^{[L]} + (1 - y) * log(1 - a^{[L]}))$



Gradient Descent Back Propagation Intro

Gradient Descent Back Propagation for an ANN



Recall Gradient Descent steps

- Take derivative of cost function wrt W and wrt b
- Move W and b opposite to the direction of the gradient scaled by a learning factor of α
- Continue until convergence

How do we back-propagate gradients through various layers?

Computation Graph

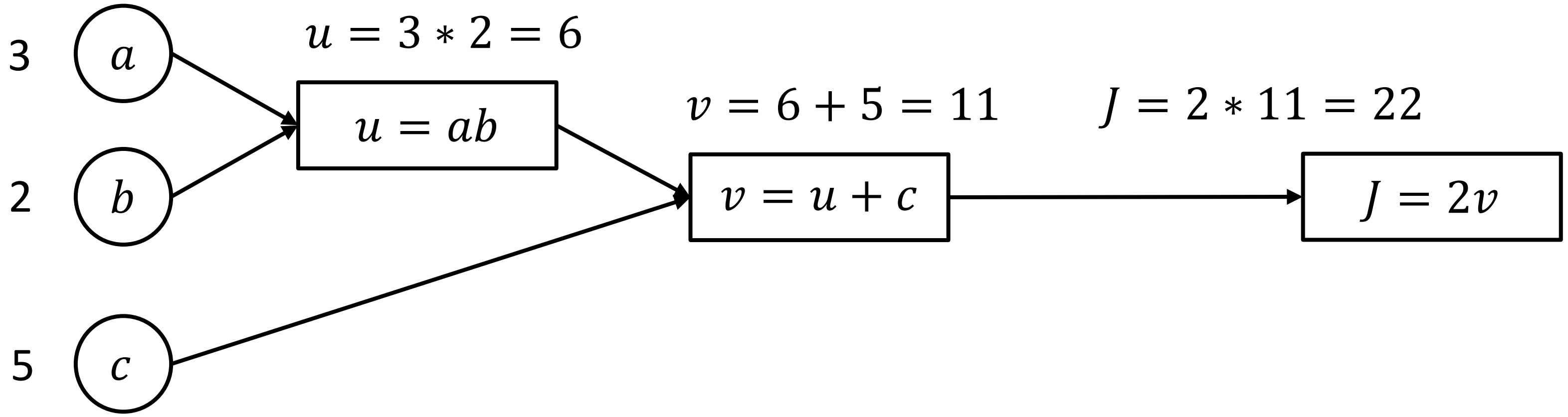
Computation Graph

What is it?

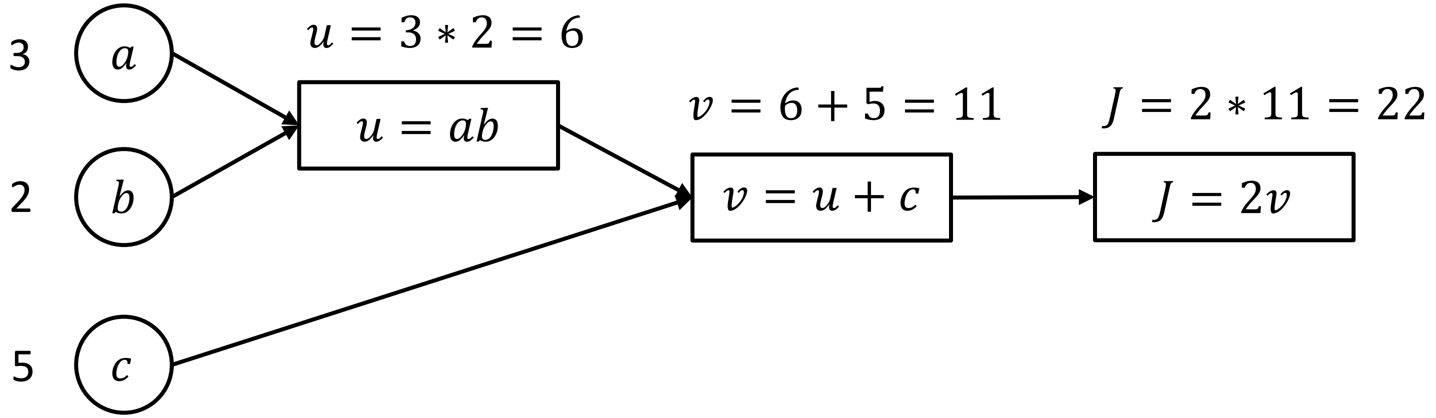
A useful tool to understand/visualize ML models

Simple arithmetic example

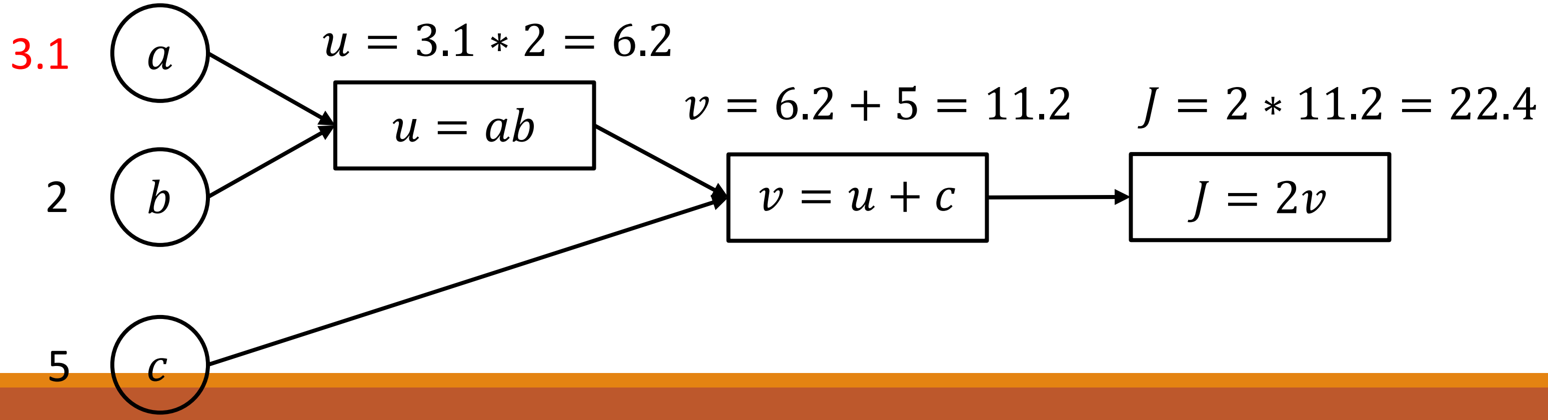
$J(a,b,c) = 2(ab + c)$ Set intermediate variables $u = ab$
$v = u + c$



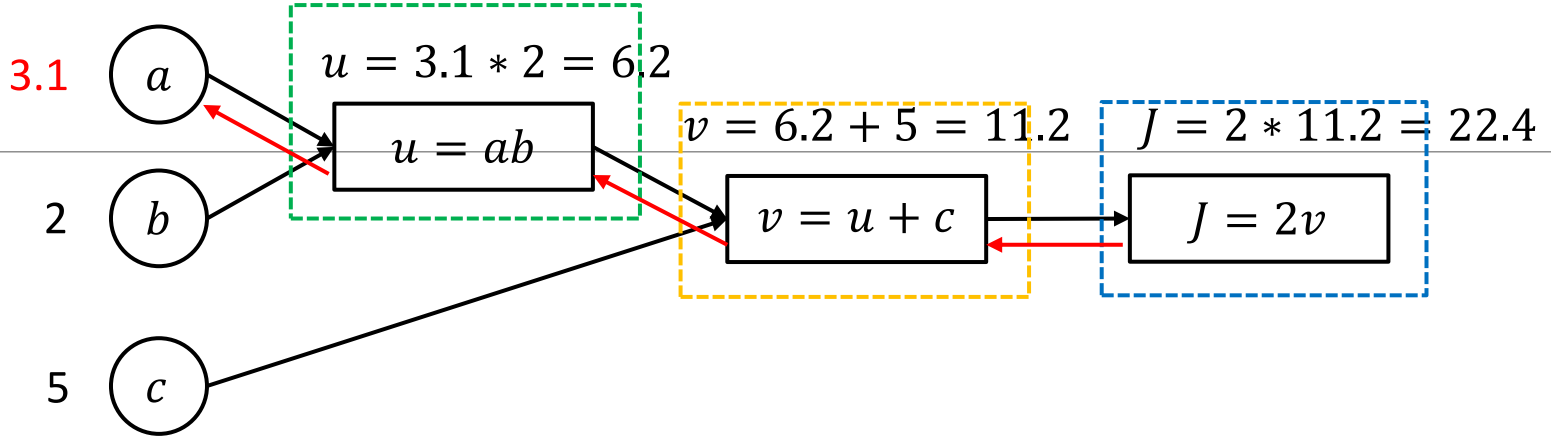
Input change leads to output change



How much will J change if a changes a little bit?



Derivative of J w.r.t a



$$\frac{\partial J}{\partial a} = \frac{22.4 - 22}{3.1 - 3} = 4$$

J will change by 4 units if a changes by 1

What about going Backward ***step by step***?

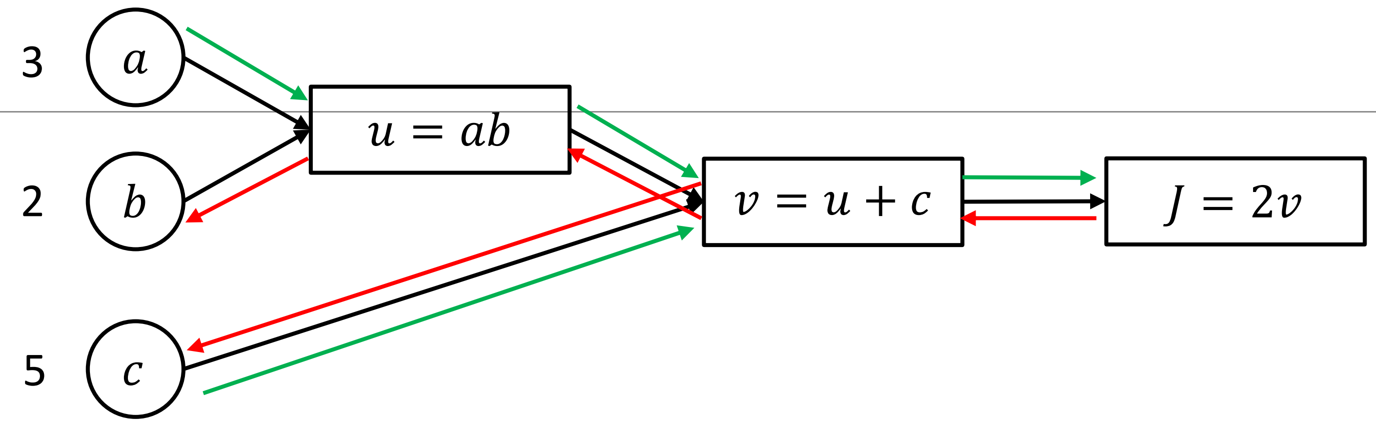
$$\frac{\partial u}{\partial a} = \frac{6.2 - 6}{3.1 - 3} = 2 \qquad \frac{\partial v}{\partial u} = \frac{11.2 - 11}{6.2 - 6} = 1 \qquad \frac{\partial J}{\partial v} = \frac{22.4 - 22}{11.2 - 11} = 2$$

Apply chain rule:

$$\frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial a} = 2 * 1 * 2 = 4 = \frac{\partial J}{\partial a}$$

Same result

Derivative of J w.r.t all inputs



$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 2 * 1 * 3 = 6 \qquad \frac{\partial J}{\partial c} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial c} = 2 * 1 = 2$$

Forward pass (green), compute the cost J

Backward pass (red), compute the derivative $\frac{\partial J}{\partial ?}$

Chain rule between nodes

$$N_{i+1} = g(N_i)$$

$$\begin{tikzcd} {} & \boxed{N_i} \arrow[l, "dN_i"'] \arrow[r, shift left, "g"] & \boxed{N_{i+1}} \arrow[l, shift left, "dN_{i+1}"] \arrow[r] & {} \end{tikzcd} \quad \dots \quad \text{Output } J$$

$$\frac{\partial J}{\partial N_i} = ? \qquad \boxed{\frac{\partial J}{\partial N_{i+1}}}$$

$$= \frac{\partial J}{\partial N_{i+1}} \cdot \frac{\partial N_{i+1}}{\partial N_i}$$ dot-product for vectors/matrices

$$\boxed{dN_i = \frac{\partial J}{\partial N_{i+1}} g'(N_i)} \qquad = dN_{i+1} g'(N_i)$$

The derivative of one node can be inferred from that of the adjacent node, and the ***function*** that connects them

Notation for **derivatives**:

dN_{i+1} is for $\dfrac{\partial J}{\partial N_{i+1}}$

Practice

$$N_{i+1} = g(N_i)$$

$$\boxed{N_i} \xrightarrow{g} \boxed{N_{i+1}} \longrightarrow$$

$dN_i = ?$ $\qquad dN_{i+1}$

Sigmoid $g(z) = \frac{1}{1+e^{-z}}$

$$dN_i = dN_{i+1} g'(N_i) = ?$$

Answer: $g'(z) = g(z)(1 - g(z))$

$$dN_i = dN_{i+1} g(N_i)(1 - g(N_i)) = dN_{i+1} N_{i+1}(1 - N_{i+1})$$

$$N_{i+1} = g(N_i)$$

$$\boxed{N_i} \xrightarrow{g} \boxed{N_{i+1}} \longrightarrow$$

$dN_i = ?$ $\qquad dN_{i+1}$

Tanh $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

$$dN_i = dN_{i+1} g'(N_i) = ?$$

Answer: $g'(z) = 1 - g^2(z)$

$$dN_i = dN_{i+1}(1 - g^2(N_i)) = dN_{i+1}\left(1 - N_{i+1}^2\right)$$

Computation graph for logistic regression

- Review:

$$z = x\boldsymbol{w}^{T} + b$$

$$\hat{y} = \underline{a} = \sigma(z) = \frac{1}{1+e^{-z}}$$

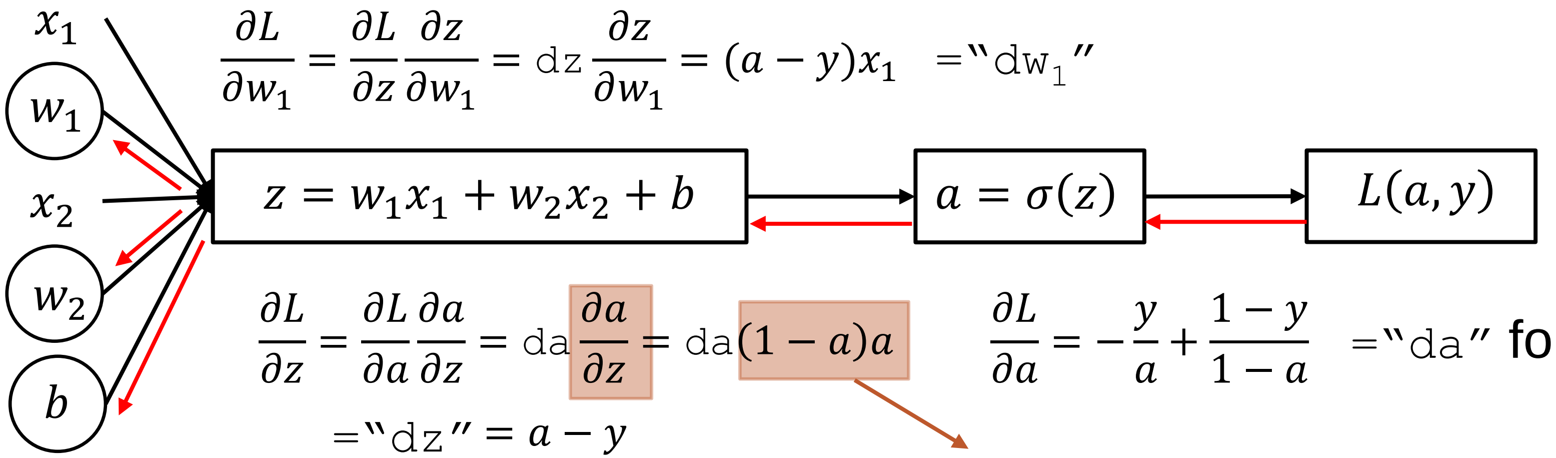
$$\underline{L(a,y)} = -(y\log(a) + (1-y)\log(1-a))$$

$\boldsymbol{a}$ for "activation"

loss function $L(a,y)$

= The **cost** of a single training example

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \mathtt{dz}\frac{\partial z}{\partial w_1} = (a-y)x_1 \quad = \text{“}\mathtt{dw_1}\text{”}$$



$\mathtt{dw_2} = (a-y)x_1$

$\mathtt{dw_2} = (a-y)x_2$

$\mathtt{db} = (a-y)$

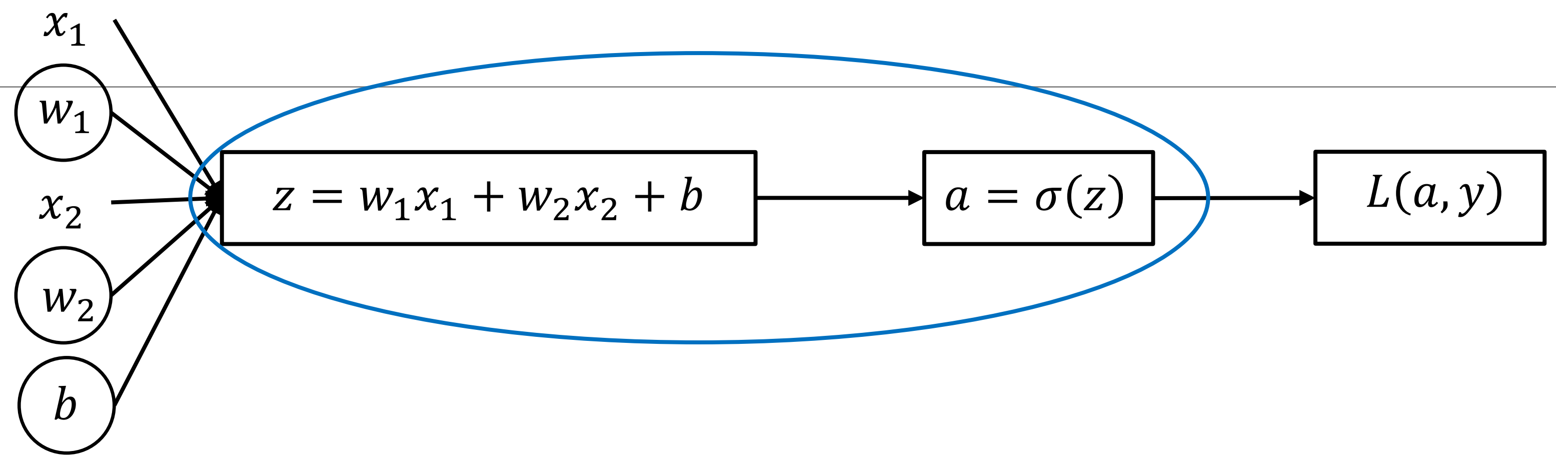
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = \mathtt{da}\boxed{\frac{\partial a}{\partial z}} = \mathtt{da}\boxed{(1-a)a}$$

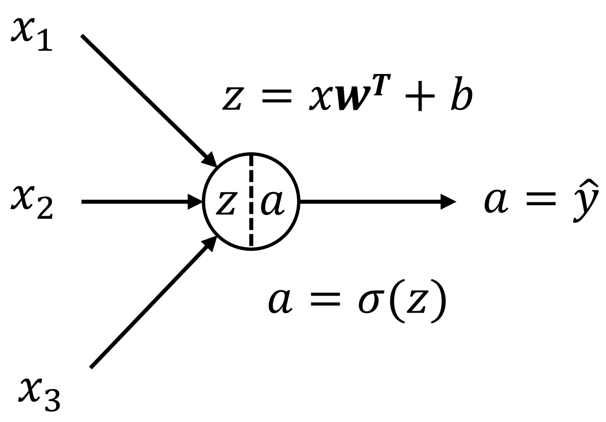
$$= \text{“}\mathtt{dz}\text{”} = a - y$$

$$\frac{\partial L}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a} \quad = \text{“}\mathtt{da}\text{” for short}$$

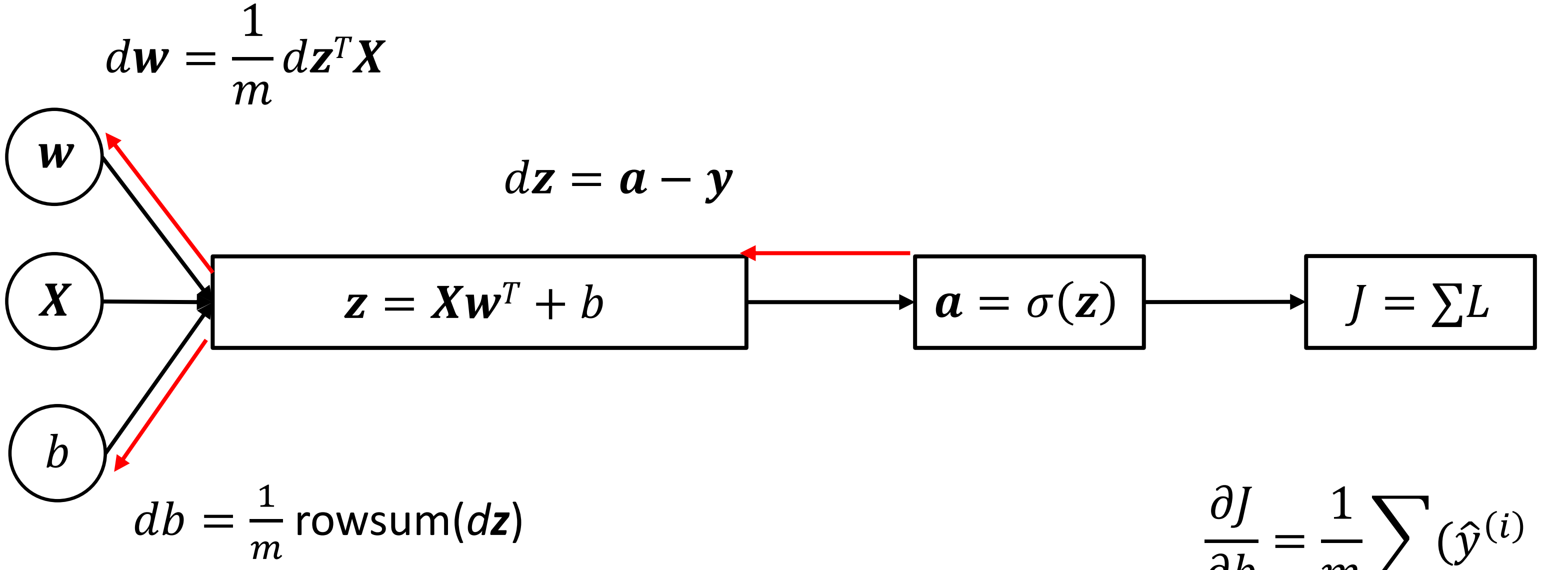
The derivative of sigmoid function (previous slide)

Wrapped into a single unit





Vectorizing logistic regression (recap)



$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_i (\hat{y}^{(i)} - y^{(i)})$$

$$\frac{\partial J}{\partial w_k} = \frac{1}{m} \sum_i (\hat{y}^{(i)} - y^{(i)})\, \mathrm{x_k}^{(i)}$$, *k=1,2,…n*

Code Examples

Computational graph in PyTorch

Computational graph is implemented by `torch.autograd` in PyTorch, and automatic differentiation engine

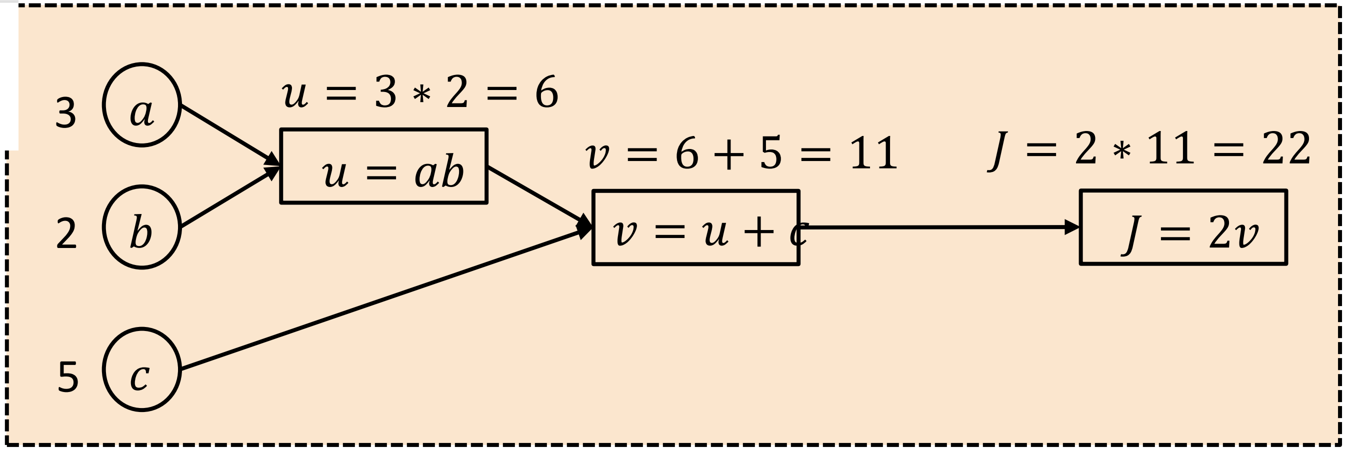
```
import torch
```

```
a = torch.tensor([3.], requires_grad=True)
b = torch.tensor([2.], requires_grad=True)
c = torch.tensor([5.], requires_grad=True)
print(a)
print(b)
print(c)
```

```
tensor([3.], requires_grad=True)
tensor([2.], requires_grad=True)
tensor([5.], requires_grad=True)
```

```
# define the output function
J = 2 * (a * b + c)
print(J)
```

```
tensor([22.], grad_fn=<MulBackward0>)
```



“requires_grad=True” to indicate they are leaf tensor of the computational graph

What is a Tensor?

- A tensor is a mathematical object that generalizes the concept of scalars, vectors, and matrices to higher-dimensional spaces. In the context of machine learning and deep learning, tensors are fundamental data structures used to represent and store multi-dimensional data.

- Key concepts: Dimension/Rank, Shape, Dtype

- Dimension/Rank
  - [[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]] has three dimensions

- Here are some types of tensors:
  - **Scalar:** A scalar is a single numerical value. In tensor notation, a scalar is considered a tensor of rank 0.
  - **Vector:** A vector is an ordered collection of scalars. It is a one-dimensional tensor. For example, a vector with three elements could represent a point in 3D space.
  - **Matrix:** A matrix is a two-dimensional array of scalars. It is a two-dimensional tensor. Matrices are often used to represent linear transformations, often used to carry image data or tabular data
  - **Tensor:** A tensor is a multi-dimensional array of numerical values. It is a generalization of scalars, vectors, and matrices. Tensors can have any number of dimensions, and each dimension is called a mode or an axis. In deep learning, tensors with three or more dimensions are commonly used to represent complex data structures, such as images, sequences, and volumes. A tensor with rank 3 can be visualized as a cube or a stack of matrices

- Shape of a tensor is related to its dimensions. The number of entries in each dimension
  - [[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]] has shape (2,2,3)

- Dtype refers to the data type of the elements, e.g. int32, float32, and boolean

- In libraries like PyTorch, tensors provide a convenient way to perform operations on multi-dimensional data, making them a foundational concept in the field of deep learning.

Computational graph in PyTorch

Let the output J change by 1.0, and let the gradient “flow back”

```
external_grad = torch.tensor([1.0])
J.backward(gradient = external_grad)
```

Remember to clear the gradients before you run the next backward step

```
# clear all gradients
a.grad.data.zero_()
b.grad.data.zero_()
c.grad.data.zero_()
```

The gradients for all variables in the graph are automatically computed

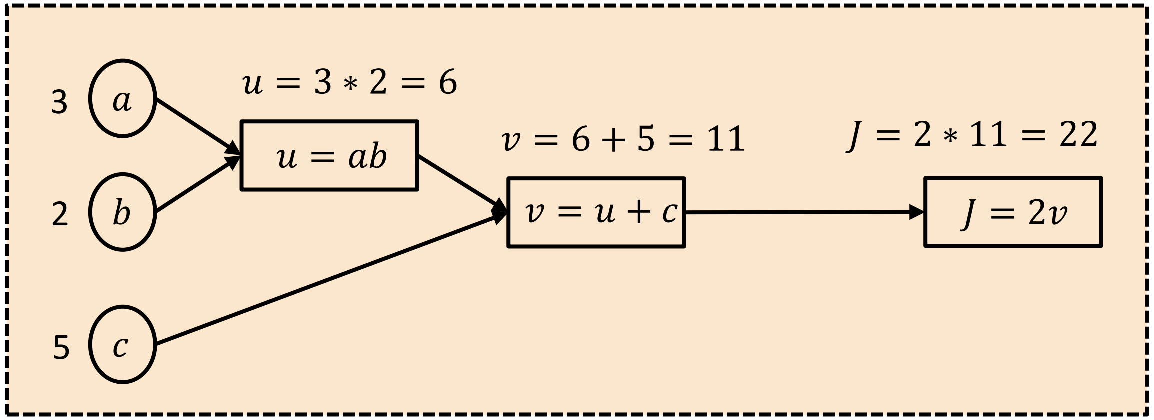
```
print(a.grad)
print(b.grad)
print(c.grad)
```

```
tensor([4.])
tensor([6.])
tensor([2.])
```

$$\frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial a} = 2*1*2 = 4 = \frac{\partial J}{\partial a}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 2*1*3 = 6$$

$$\frac{\partial J}{\partial c} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial c} = 2*1 = 2$$



Basic neural network in PyTorch

torch.nn.Linear (pass-through activation fn)

1. Use the `torch.nn` module

```
import torch
import torch.nn as nn
import numpy as np
torch.manual_seed(0)
```

2. Define a Linear layer

```
m = nn.Linear(4, 3)
# Equivalent to nn.Linear(in_features = 4, out_features = 3, bias = True
```

3. Let's look at the initial weight and bias:

```
print(m.weight)
print(m.bias)
```

```
Parameter containing:
tensor([[ 0.3742, -0.0806,  0.0529,  0.4527],
        [-0.4638, -0.3148, -0.1266, -0.1949],
        [ 0.4320, -0.3241, -0.2302, -0.3493]], requires_grad=True)
Parameter containing:
tensor([-0.4683, -0.2919,  0.4298], requires_grad=True)
```

Recall wt matrix: $n_l \times n_{l-1}$
Bias vector: $1 \times n_l$

Basic neural network in PyTorch

4. Set some dummy values

```
# Manually set weight and bias
m.weight.data = torch.tensor(np.arange(1,13).reshape((3,4))).float()
print(m.weight)
m.bias.data = torch.ones_like(m.bias.data).float()
print(m.bias)
```

```
Parameter containing:
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.],
        [ 9., 10., 11., 12.]], requires_grad=True)
Parameter containing:
tensor([1., 1., 1.], requires_grad=True)
```

Basic neural network in PyTorch

5. Prepare a single input data example

```
x = torch.randn(1, 4)
print(x.shape)
```

```
torch.Size([1, 4])
```

Get the output by ***calling*** the model

```
out = m(x)
print(out.shape)
```

```
torch.Size([1, 3])
```

x is 1×4:

By default, the first dimension of a tensor stands for batch size

- out is 1×3:
- Pytorch calculates XW^T+B. Check the dimensions:
- X is 1 x 4, W^T is 4 x 3, B is 1 x 3, hence the output is 1 x 3

Basic neural network in PyTorch

6. Prepare a batch of data examples

```
x1 = torch.randn(100, 4)
out1 = m(x1)
print(out1.shape)
```

```
torch.Size([100, 3])
```

x is 100×4:

By default, the first dimension of a tensor stands for batch size

- out is 100×3:
- Pytorch calculates XW^T+B. Check the dimensions:
- X is 100 x 4, W^T is 4 x 3, B is 1 x 3, hence the output is 100 x 3

Basic neural network in PyTorch

7. Let's manually check what computation is done

Prepare an input vector: [1,1,1,1]

```
x2 = torch.ones(1,4)
print(x2)
out2 = m(x2)
print(out2)
```

```
tensor([[1., 1., 1., 1.]])
tensor([[11., 27., 43.]], grad_fn=<AddmmBackward>)
```

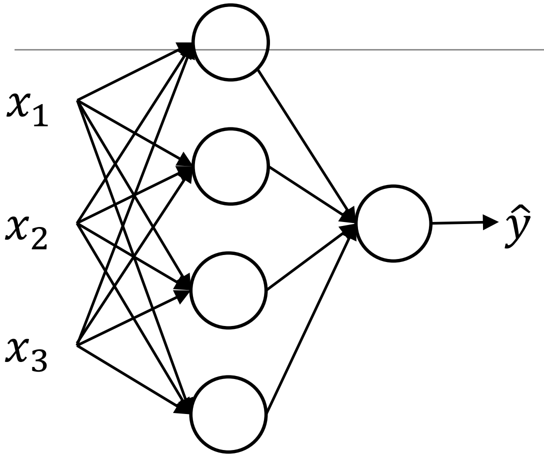
```
weight = m.weight.data.numpy()
bias = m.bias.data.numpy()
x = x2.numpy().squeeze()

print(np.dot(weight[0,:], x) + bias[0])
print(np.dot(weight[1,:], x) + bias[1])
print(np.dot(weight[2,:], x) + bias[2])

out2_manual = np.sum(weight, axis=1) + bias

print(np.equal(out2_manual, out2.data.numpy()))
```

```
11.0
27.0
43.0
[[ True  True  True]]
```

$$\text{out2} = XW^T + B = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 11 & 27 & 43 \end{bmatrix}$$

Activation Functions

Activation functions



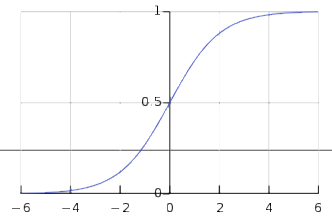
Single input example x

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \cancel{\sigma(z^{[1]})} \quad g^{[1]}(z^{[1]})$$

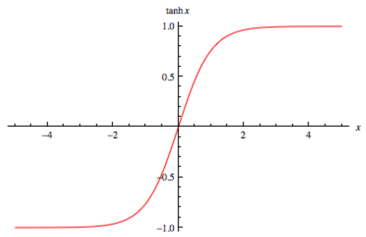
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \cancel{\sigma(z^{[2]})} \quad g^{[2]}(z^{[2]})$$



Sigmoid

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

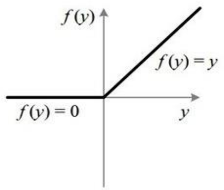


$$g(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

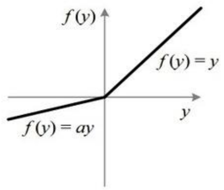
"hyperbolic tangent". Almost always works better for hidden layers than sigmoid.

Downside for both: small gradient for extreme z values

ReLU



LeakyReLU



Rectified linear unit (ReLU)

$$g(z) = \max(0, z)$$

Leaky ReLU

$$g(z) = \max(\alpha z, z)$$

Rule-of-thumb for choosing activation functions

- Binary classification, $y \in \{0,1\}$, use sigmoid for output layer.
- Regression, use a linear pass-through function for output layer
- For all the other units, use ReLU (increasingly the default choice) or tanh
  - ReLU learns much faster than tanh.

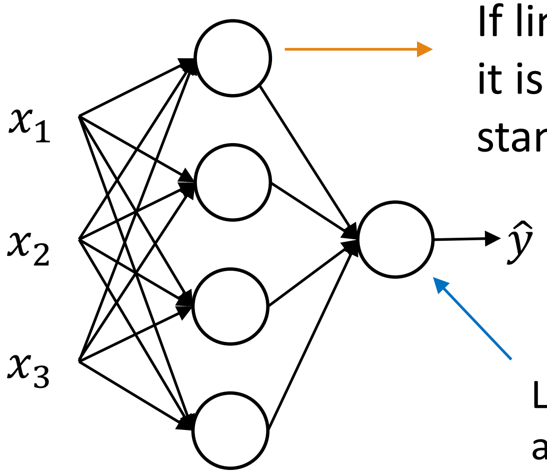
Why do we need non-linear activation functions?

Why not linear activation?

$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = z^{[1]}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = z^{[2]}$$

$$\Rightarrow$$

$$\begin{aligned} a^{[2]} &= W^{[2]}\left(W^{[1]}x + b^{[1]}\right) + b^{[2]} \\ &= \left(W^{[2]}W^{[1]}\right)x + \left(W^{[2]}b^{[1]} + b^{[2]}\right) \\ &= W'x + b' \end{aligned}$$



If linear activation used in hidden layer, it is no more expressive than a standard logistic regression

Linear activation can be used at output layer as a *regression* model

Derivatives of activation functions

Sigmoid $g(z) = \frac{1}{1 + e^{-z}} = a$ $g'(z) = g(z)\big(1 - g(z)\big) = a(1 - a)$

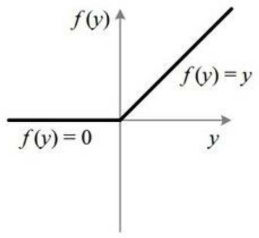
Tanh $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = a$ $g'(z) = 1 - \big(1 - g(z)\big)^2 = 1 - a^2$

ReLU $g(z) = \max(0, z)$ $g'(z) = \begin{cases} 1, & z > 0 \\ 0, & z < 0 \end{cases}$

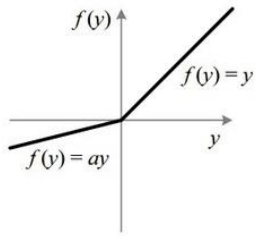
LeakyReLU $g(z) = \max(\alpha z, z)$ $g'(z) = \begin{cases} 1, & z > 0 \\ \alpha, & z < 0 \end{cases}$

α is a tiny positive value

ReLU



LeakyReLU



Activation functions in PyTorch

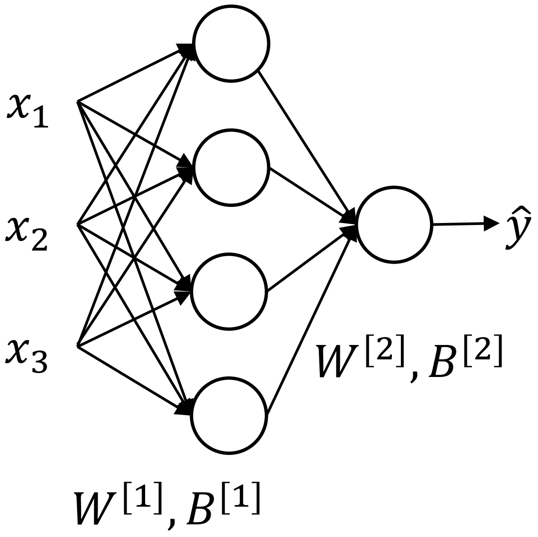
torch.nn.Sigmoid, torch.nn.LogSigmoid

torch.nn.Tanh, torch.nn.HardTanh

torch.nn.ReLU, torch.LeakyReLU, torch.ReLU6, torch.RReLU, …

Derivatives of Vectors and Matrices/Backpropagation

Gradient descent for neural networks



Input X dimension: $n^{[0]}$

# of units in layer 1: $n^{[1]}$ # of units in layer 2: $n^{[2]}$

$W^{[1]}$ dimension: $n^{[1]} \times n^{[0]}$ $B^{[1]}$ dimension: $1\ x n^{[1]}$

$W^{[2]}$ dimension: $n^{[2]} \times n^{[1]}$ $B^{[2]}$ dimension: $1\ x\ n^{[2]}$

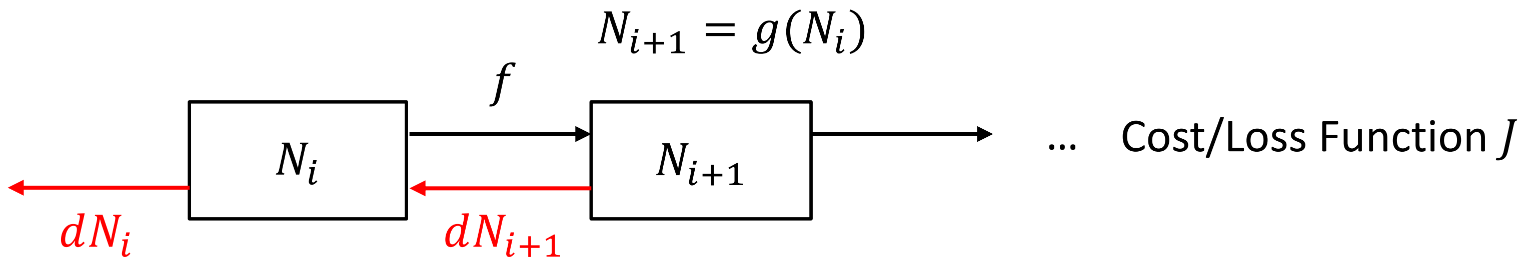
Binary classification case

$$J\left(W^{[1]}, B^{[1]}, W^{[2]}, B^{[2]}\right) = \frac{1}{m}\sum\nolimits_i L(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m}\sum\nolimits_i L(\boldsymbol{a}^{(i)[2]}, y^{(i)})$$

$$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}} \quad dB^{[1]} = \frac{\partial J}{\partial B^{[1]}} \quad dW^{[2]} = \frac{\partial J}{\partial W^{[2]}} \quad dB^{[2]} = \frac{\partial J}{\partial B^{[2]}}$$

How to compute the ***gradients*** (derivatives)?

Recall – Chain Rule Between Nodes



Notation for **derivatives**:

dN_{i+1} is for $\dfrac{\partial J}{\partial N_{i+1}}$

$$dN_i = dN_{i+1} \,.\, f'(N_i)$$

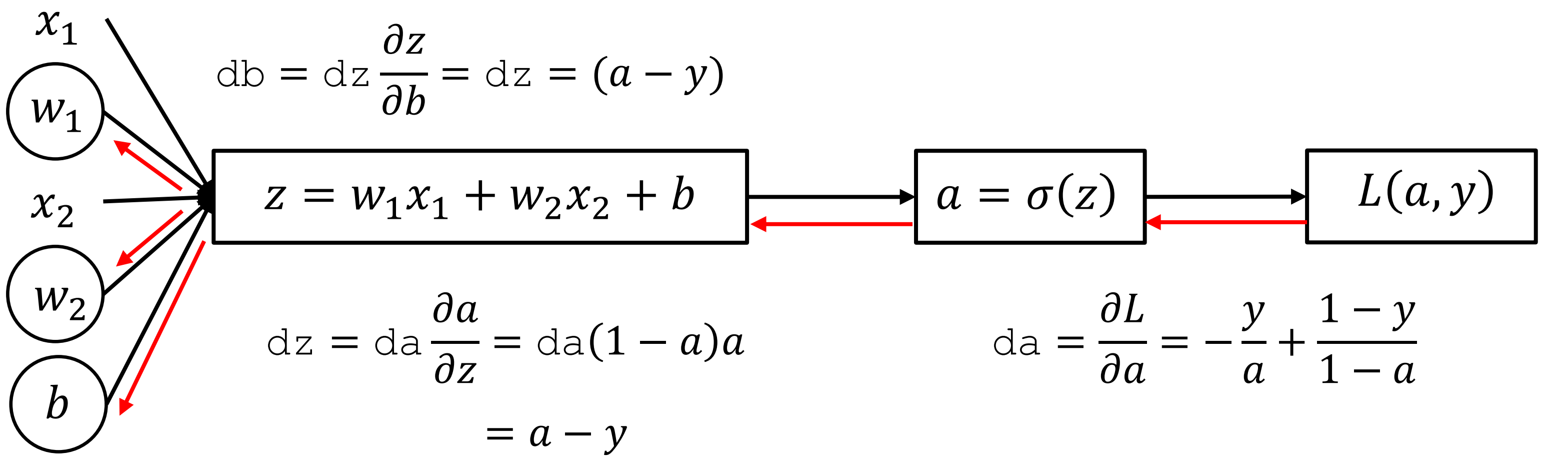
dot-product for matrices/vectors

The derivative of one node can be inferred from that of the adjacent node, and the ***function*** that connects them

Backpropagation

Recap of logistic regression: Loss Function: $L(a, y) = -(y\log(a) + (1-y)\log(1-a)); a = \hat{y}$

$$\left.\begin{aligned} \mathrm{dw}_1 &= \mathrm{dz}\frac{\partial z}{\partial w_1} = \mathrm{dz}x_1 = (a-y)x_1 \\ \mathrm{dw}_2 &= \mathrm{dz}\frac{\partial z}{\partial w_2} = \mathrm{dz}x_2 = (a-y)x_2 \end{aligned}\right\} \quad \boldsymbol{dw} = (dw_1, dw_2) = dz \cdot (x_1, x_2) = dz \cdot \boldsymbol{x}$$



Backpropagation – 1 sample data point (two-layer neural network)

$$\mathbf{d}\boldsymbol{W}^{[1]} = \boldsymbol{dz}^{[1]T}\,\boldsymbol{X}$$

$$\mathbf{dB}^{[1]} = \mathbf{dz}^{[1]}$$

$$\mathbf{d}\boldsymbol{W}^{[2]} = ? \dashrightarrow \mathbf{d}\boldsymbol{W}^{[2]} = \mathbf{dz}^{[2]T}\mathbf{a}^{[1]}$$

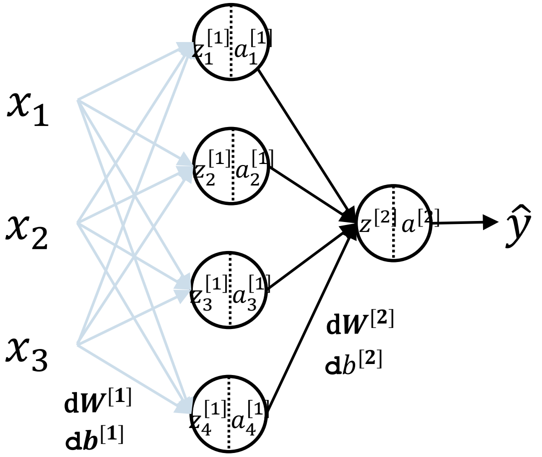
$$\mathbf{db}^{[2]} = \mathbf{dz}^{[2]}$$

$$\boldsymbol{X}, \boldsymbol{W}^{[1]}, \boldsymbol{B}^{[1]} \rightarrow \boxed{\boldsymbol{z}^{[1]} = \boldsymbol{X}\boldsymbol{W}^{[1]T} + \boldsymbol{B}^{[1]}} \rightarrow \boxed{\boldsymbol{a}^{[1]} = g(\boldsymbol{z}^{[1]})} \rightarrow \boxed{z^{[2]} = \boldsymbol{a}^{[1]}\boldsymbol{W}^{[2]T} + b^{[2]}} \rightarrow \boxed{a^{[2]} = \sigma(z^{[2]})} \rightarrow L(a^{[2]}, y)$$

$$\mathbf{dz}^{[1]} = \mathbf{da}^{[1]} * g'\left(\boldsymbol{z}^{[1]}\right) \qquad \mathbf{da}^{[1]} = \mathbf{dz}^{[2]}\, W^{[2]}$$

$$\mathtt{dz}^{[2]} = a^{[2]} - y \qquad \mathtt{da}^{[2]} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

For sigmoid act fn in the output layer



$$\hat{y} = a^{[2]}$$

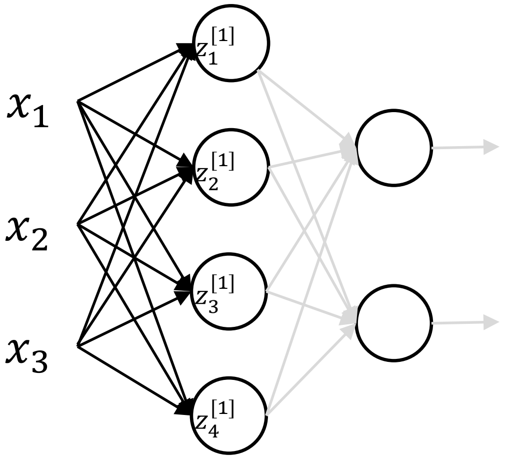
Generalize Back Propagation for Deeper ANNs Using Vectors and Matrices ==➔

Derivatives of the Loss function w.r.t. *W* and *B*

$\boxed{\boldsymbol{z}^{(i)[1]} = \boldsymbol{X}^{(i)} W^{[1]T} + B^{[1]}}$ In order to compute $\mathbf{d}\boldsymbol{W}^{[\mathbf{1}]}$, need to know $\dfrac{\boldsymbol{\partial z}^{(\boldsymbol{i})[\mathbf{1}]}}{\boldsymbol{\partial W}^{[\mathbf{1}]}}$ How?

- The partial derivatives of ***each*** component of $\boldsymbol{z}^{(i)[1]}$ w.r.t. ***each*** element of $W^{[1]}$

$$\boldsymbol{z}^{(i)[1]} = \begin{bmatrix} z_1^{(i)[1]} & z_2^{(i)[1]} & z_3^{(i)[1]} & z_4^{(i)[1]} \end{bmatrix} = \begin{bmatrix} x_1^{(i)[1]} & x_2^{(i)[1]} & x_3^{(i)[1]} \end{bmatrix} \begin{bmatrix} w_{11}^{[1]} & w_{21}^{[1]} & w_{31}^{[1]} & w_{41}^{[1]} \\ w_{12}^{[1]} & w_{22}^{[1]} & w_{32}^{[1]} & w_{42}^{[1]} \\ w_{13}^{[1]} & w_{23}^{[1]} & w_{33}^{[1]} & w_{43}^{[1]} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} & b_2^{[1]} & b_3^{[1]} & b_4^{[1]} \end{bmatrix}$$



$$\frac{\boldsymbol{\partial z}^{(\boldsymbol{i})[\mathbf{1}]}}{\boldsymbol{\partial W}^{[\mathbf{1}]}} = \begin{bmatrix} \dfrac{\partial z_1^{(i)[1]}}{\partial w_{11}^{[1]}} & \dfrac{\partial z_2^{(i)[1]}}{\partial w_{12}^{[1]}} & \dfrac{\partial z_3^{(i)[1]}}{\partial w_{13}^{[1]}} \\ \dfrac{\partial z_1^{(i)[1]}}{\partial w_{21}^{[1]}} & \dfrac{\partial z_2^{(i)[1]}}{\partial w_{22}^{[1]}} & \dfrac{\partial z_3^{(i)[1]}}{\partial w_{23}^{[1]}} \\ \dfrac{\partial z_1^{(i)[1]}}{\partial w_{31}^{[1]}} & \dfrac{\partial z_2^{(i)[1]}}{\partial w_{32}^{[1]}} & \dfrac{\partial z_3^{(i)[1]}}{\partial w_{33}^{[1]}} \\ \dfrac{\partial z_1^{(i)[1]}}{\partial w_{41}^{[1]}} & \dfrac{\partial z_2^{(i)[1]}}{\partial w_{42}^{[1]}} & \dfrac{\partial z_3^{(i)[1]}}{\partial w_{43}^{[1]}} \end{bmatrix} = \begin{bmatrix} x_1^{(i)} & 0 & 0 \\ 0 & x_2^{(i)} & 0 \\ 0 & 0 & x_3^{(i)} \\ 0 & 0 & 0 \end{bmatrix}$$

Collapse into 1x3

- Derivative chain rule:
- $\mathbf{d}\boldsymbol{W}^{(\boldsymbol{i})[\mathbf{1}]} = \mathbf{dz}^{(\boldsymbol{i})[\mathbf{1}]\boldsymbol{T}} \dfrac{\boldsymbol{\partial z}^{(i)[1]}}{\boldsymbol{\partial W}^{[1]}} = \mathbf{dz}^{(\boldsymbol{i})[\mathbf{1}]\boldsymbol{T}} \boldsymbol{x}^{(\boldsymbol{i})}$
- *4 x 3* *4 x1* *1 X 3*
- In general, at each layer *l*,
- $\mathbf{d}\boldsymbol{W}^{(\boldsymbol{i})[\boldsymbol{l}]} = \mathbf{dz}^{(\boldsymbol{i})[\boldsymbol{l}]\boldsymbol{T}} \dfrac{\boldsymbol{\partial z}^{(i)[l]}}{\boldsymbol{\partial W}^{[l]}} = \mathbf{dz}^{(\boldsymbol{i})[\boldsymbol{l}]\boldsymbol{T}} \boldsymbol{a}^{(\boldsymbol{i})[\boldsymbol{l}-\mathbf{1}]}$
- $n^{[l]} \ x \ n^{[l-1]}$ $n^{[l]} \ x \ 1$ $1 \ x \ n^{[l-1]}$

- How about $d\boldsymbol{B}^{(i)[l]}$?
- $d\boldsymbol{B}^{(i)[l]} = \mathbf{dz}^{(i)[l]} \dfrac{\boldsymbol{\partial z}^{(i)[l]}}{\boldsymbol{\partial B}^{[l]}}$
- $d\boldsymbol{B}^{(i)[l]} = \mathbf{dz}^{(\boldsymbol{i})[\boldsymbol{l}]}$
- $1 \ x \ n^{[l]}$

Derivatives of the Loss Function w.r.t. $\boldsymbol{a}$ and $\boldsymbol{z}$

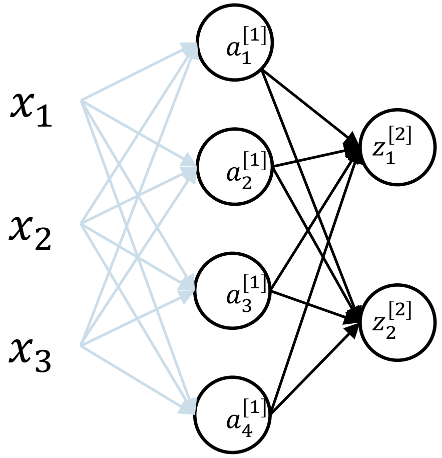
Hidden to output

$$\boxed{\boldsymbol{z}^{(i)[2]} = \boldsymbol{a}^{(i)[1]}\boldsymbol{W}^{[2]T} + \boldsymbol{B}^{[2]}}$$

Want to know $\boldsymbol{da}^{[1]}$

Need to compute $\dfrac{\partial \boldsymbol{z}(\boldsymbol{i})^{[2]}}{\partial \boldsymbol{a}^{[1]}}$

$$\boldsymbol{z}^{(i)[2]} = \begin{bmatrix} z_1^{(i)[2]} & z_2^{(i)[2]} \end{bmatrix} = \begin{bmatrix} a_1^{(i)[1]} & a_2^{(i)[1]} & a_3^{(i)[1]} & a_4^{(i)[1]} \end{bmatrix} \begin{bmatrix} w_{11}^{[2]} & w_{21}^{[2]} \\ w_{12}^{[2]} & w_{22}^{[2]} \\ w_{13}^{[2]} & w_{23}^{[2]} \\ w_{14}^{[2]} & w_{24}^{[2]} \end{bmatrix} + \begin{bmatrix} b_1^{[2]} & b_2^{[2]} \end{bmatrix}$$



$$\frac{\partial \boldsymbol{z}^{(\boldsymbol{i})[\boldsymbol{2}]}}{\partial \boldsymbol{a}^{(\boldsymbol{i})[\boldsymbol{1}]}} = \begin{bmatrix} \frac{\partial z_1^{(i)[2]}}{\partial a_1^{(i)[1]}} & \frac{\partial z_1^{(i)[2]}}{\partial a_2^{(i)[1]}} & \frac{\partial z_1^{(i)[2]}}{\partial a_3^{(i)[1]}} & \frac{\partial z_1^{(i)[2]}}{\partial a_4^{(i)[1]}} \\ \frac{\partial z_2^{(i)[1]}}{\partial a_2^{(i)[1]}} & \frac{\partial z_2^{(i)[1]}}{\partial a_2^{(i)[1]}} & \frac{\partial z_2^{(i)[1]}}{\partial a_3^{(i)[1]}} & \frac{\partial z_2^{(i)[1]}}{\partial a_4^{(i)[1]}} \end{bmatrix} = \begin{bmatrix} w_{11}^{[2]} & w_{12}^{[2]} & w_{13}^{[2]} & w_{14}^{[2]} \\ w_{21}^{[2]} & w_{22}^{[2]} & w_{23}^{[2]} & w_{24}^{[2]} \end{bmatrix} = \boldsymbol{W}^{[2]}$$

- $\boldsymbol{da}^{(\boldsymbol{i})[\boldsymbol{1}]} = \boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{2}]}\boldsymbol{W}^{[2]}$
- In general: $\boldsymbol{da}^{(\boldsymbol{i})[\boldsymbol{l-1}]} = \boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l}]}\boldsymbol{W}^{[l]}$
- How about $\boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{1}]}$, or in general, $\boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l-1}]}$?
- $\mathbf{a}^{(\boldsymbol{i})[\boldsymbol{l-1}]} = g\left(\boldsymbol{z}^{(\boldsymbol{i})[l-1]}\right)$
- $\boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l-1}]} = \mathbf{da}^{(\boldsymbol{i})[\boldsymbol{l-1}]} * g'\left(\boldsymbol{z}^{(\boldsymbol{i})[l-1]}\right)$
- $\boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l-1}]} = \boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l}]}\boldsymbol{W}^{[l]} * g'\left(\boldsymbol{z}^{(\boldsymbol{i})[l-1]}\right)$
- For a sigmoid activation function in the output layer:
  - $\boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{L}]} = \mathbf{a}^{(\boldsymbol{i})[\boldsymbol{L}]} - \boldsymbol{y}^{(i)}$

Vectorizing backprop across *m* training examples

$$d\boldsymbol{W}^{[l]} = \frac{\partial}{\partial \boldsymbol{W}^{[l]}} J = \frac{\partial}{\partial \boldsymbol{W}^{[l]}} \left(\frac{1}{m}\sum_i L^{(i)}\right) = \frac{1}{m}\sum_{i=1}^{m} d\boldsymbol{W}^{(i)[l]} \quad d\boldsymbol{B}^{[l]} = \frac{1}{m}\sum_{i=1}^{m} d\boldsymbol{B}^{(i)[l]} \quad d\boldsymbol{z}^{[l]} = \frac{1}{m}\sum_{i=1}^{m} d\boldsymbol{z}^{(i)[l]} \quad d\boldsymbol{a}^{[l]} = \frac{1}{m}\sum_{i=1}^{m} d\boldsymbol{a}^{(i)[l]}$$

Single example

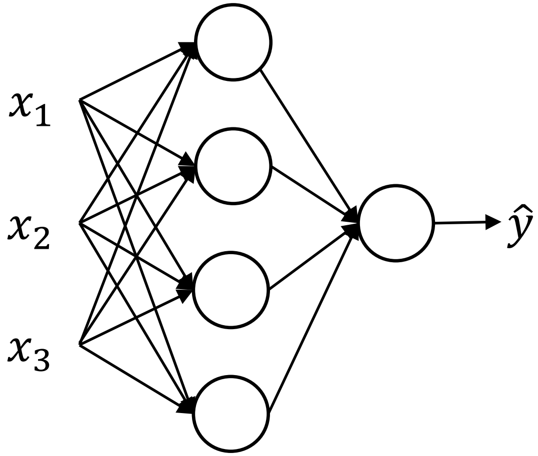
$$\boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l}-\boldsymbol{1}]} = \boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l}]}\boldsymbol{W}^{[l]} * g'\left(\boldsymbol{z}^{(\boldsymbol{i})[l-1]}\right)$$

$$\boldsymbol{da}^{(\boldsymbol{i})[\boldsymbol{l}-\boldsymbol{1}]} = \boldsymbol{dz}^{(\boldsymbol{i})[\boldsymbol{l}]}\boldsymbol{W}^{[l]}$$

$$\mathbf{d}\boldsymbol{W}^{(\boldsymbol{i})[\boldsymbol{l}]} = \mathbf{dz}^{(\boldsymbol{i})[\boldsymbol{l}]\boldsymbol{T}}\boldsymbol{a}^{(\boldsymbol{i})[\boldsymbol{l}-\boldsymbol{1}]}$$

$$d\boldsymbol{B}^{(i)[l]} = \mathbf{dz}^{(\boldsymbol{i})[\boldsymbol{l}]}$$

$$d\boldsymbol{z}^{(\boldsymbol{i})[\boldsymbol{L}]} = \mathbf{a}^{(i)[L]} - \boldsymbol{y}^{(i)}$$ for sigmoid activation function at the output layer



$$d\boldsymbol{z}^{[l-1]} = d\boldsymbol{z}^{[l]}\boldsymbol{W}^{[l]} * g'(\boldsymbol{z}^{[l-1]})$$

$$\boldsymbol{da}^{[\boldsymbol{l}-\boldsymbol{1}]} = \boldsymbol{dz}^{[\boldsymbol{l}]}\boldsymbol{W}^{[l]}$$

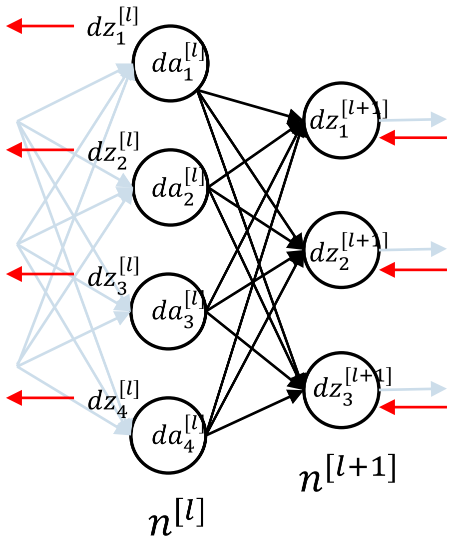
m examples

$$d\boldsymbol{W}^{[l]} = \frac{1}{m} d\boldsymbol{z}^{[l]T}\boldsymbol{a}^{[\boldsymbol{l}-\boldsymbol{1}]}$$

$$d\boldsymbol{B}^{[l]} = \frac{1}{m} rowsum(d\boldsymbol{z}^{[l]})$$

$$d\boldsymbol{z}^{[\boldsymbol{L}]} = \frac{1}{m}\mathrm{rowsum}(\mathbf{a}^{[L]} - \boldsymbol{y})$$ for sigmoid activation function at the output layer

Back Propagation Process



$\boldsymbol{W}^{[l+1]}, \boldsymbol{B}^{[l+1]}$

$d\boldsymbol{z}^{[\boldsymbol{L}]} = \frac{1}{m}\text{rowsum}(\mathbf{a}^{[L]} - \boldsymbol{y})$ for sigmoid activation function at the output layer

$$\boldsymbol{z}^{[l+1]} = \boldsymbol{a}^{[l]}\boldsymbol{W}^{[l+1]T} + \boldsymbol{B}^{[l+1]}$$

$$\boldsymbol{a}^{[l]} = g'(\boldsymbol{z}^{[l]})$$

$\boldsymbol{z}^{[l+1]}: 1\ x\ n^{[l+1]}, \boldsymbol{a}^{[l]}: 1\ x\ n^{[l]}, \boldsymbol{W}^{[l+1]}: n^{[l+1]} \times n^{[l]}, \boldsymbol{B}^{[l+1]}: 1\ x\ n^{[l+1]}$

$$d\boldsymbol{W}^{[l+1]} = \mathbf{dz}^{[\boldsymbol{l}+\mathbf{1}]\boldsymbol{T}}\boldsymbol{a}^{[\boldsymbol{l}]}$$

$n^{[l+1]} \times n^{[l]}$ $\quad n^{[l+1]} \times 1$ $\quad 1 \times n^{[l]}$

$$d\boldsymbol{B}^{[l+1]} = \mathbf{dz}^{[\boldsymbol{l}+\mathbf{1}]}$$

$1 \times n^{[l+1]}$ $\quad 1 \times n^{[l+1]}$

$$\boldsymbol{da}^{[\boldsymbol{l}]} = \boldsymbol{dz}^{[\boldsymbol{l}+\mathbf{1}]}\boldsymbol{W}^{[l+1]}$$

$1\ x\ n^{[l]}$ $\quad 1\ x\ n^{[l+1]}$ $\quad n^{[l+1]}\ x\ n^{[l]}$

Usually combined

- $\boldsymbol{dz}^{[\boldsymbol{l}]} = \mathbf{da}^{[\boldsymbol{l}]} * g'\left(\boldsymbol{z}^{[l]}\right) = \boldsymbol{dz}^{[\boldsymbol{l}+\mathbf{1}]}\boldsymbol{W}^{[l+1]} * g'\left(\boldsymbol{z}^{[l]}\right)$

$1\ x\ n^{[l]}$ $\quad 1\ x\ n^{[l]}$ $\quad 1\ x\ n^{[l+1]}$ $\quad n^{[l+1]} x\ n^{[l]}$

$\boldsymbol{dz}^{[l]}$ is then passed to layer $l-1$ to compute $d\boldsymbol{W}^{[l]}$ and $dB^{[l]}$

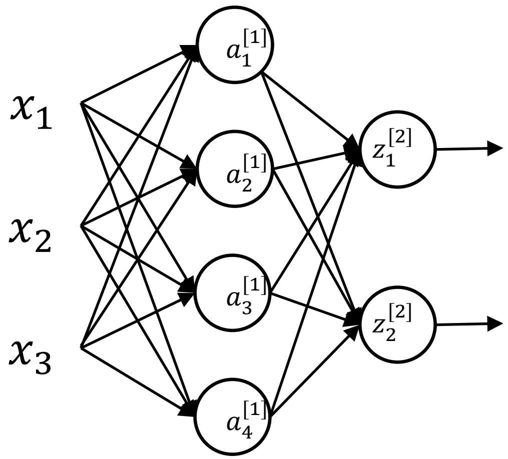
Backpropagation Code Examples

Backprop in PyTorch

Necessary imports

```python
import torch
import torch.nn as nn
import numpy as np
torch.manual_seed(0)
```

```
<torch._C.Generator at 0x111b7ddb0>
```



Define linear and activation layers

```python
linear1 = nn.Linear(3, 4)
act1 = nn.ReLU()
linear2 = nn.Linear(4, 2)
act2 = nn.Sigmoid()
```

Initialize to values easier to read:

```python
linear1.weight.data = torch.tensor(np.arange(1,13).reshape((4,3))).float()
print(linear1.weight)
linear1.bias.data = torch.zeros_like(linear1.bias.data).float()
print(linear1.bias)
```

```
Parameter containing:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.],
        [10., 11., 12.]], requires_grad=True)
Parameter containing:
tensor([0., 0., 0., 0.], requires_grad=True)
```

Backprop in PyTorch (layer 1)

Prepare some toy data, and watch the output from layer 1

```
x = torch.tensor([[1.,2.,3.]], requires_grad=True)
print('x: ', x)
z1 = linear1(x)
a1 = act1(z1)

print(z1)
print(a1)
print(a1.shape)
```

```
x:  tensor([[1., 2., 3.]], requires_grad=True)
tensor([[14., 32., 50., 68.]], grad_fn=<AddmmBackward>)
tensor([[14., 32., 50., 68.]], grad_fn=<ReluBackward0>)
torch.Size([1, 4])
```

Forward:

$$z1 = \underbrace{\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}}_{x} \underbrace{\begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}}_{\text{Weight.T}} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{Bias}}$$

Backprop in PyTorch (layer 1)

Create some pseudo gradients that are in the same shape as a1

```
external_grad = torch.ones_like(a1) * 0.5
a1.backward(gradient=external_grad)
```

$$dz = da = [.5 \quad .5 \quad .5 \quad .5] \quad W = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Gradients computed:

```
print(x.grad)
print(linear1.weight.grad)
print(linear1.bias.grad)
```

```
tensor([[11., 13., 15.]])
tensor([[0.5000, 1.0000, 1.5000],
        [0.5000, 1.0000, 1.5000],
        [0.5000, 1.0000, 1.5000],
        [0.5000, 1.0000, 1.5000]])
tensor([0.5000, 0.5000, 0.5000, 0.5000])
```

$$dx = dzW = [.5 \quad .5 \quad .5 \quad .5] \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = [11 \quad 13 \quad 15]$$

$$dW = dz^T \cdot x = \begin{bmatrix} .5 \\ .5 \\ .5 \\ .5 \end{bmatrix} [1. \quad 2. \quad 3.] =$$

$$dB = dz = [.5 \quad .5 \quad .5 \quad .5]$$

Backprop in PyTorch (layer 2)

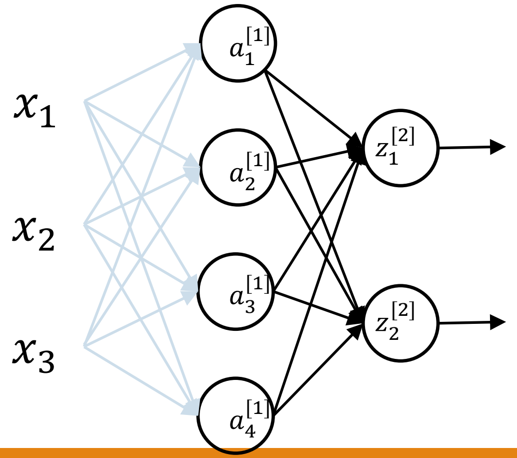
Initialize the weight of linear2 to zeros (0), because sigmoid function saturates very fast

Forward pass:

```
x = torch.tensor([[1.,2.,3.]])
z1 = linear1(x)
a1 = act1(z1)
z2 = linear2(a1)
a2 = act2(z2)
```

```
linear2.weight.data = torch.zeros_like(linear2.weight.data).float()
linear2.bias.data = torch.zeros_like(linear2.bias.data).float()
print(linear2.weight)
print(linear2.bias)
```

```
Parameter containing:
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]], requires_grad=True)
Parameter containing:
tensor([0., 0.], requires_grad=True)
```



Output:

```
z2:  tensor([[0., 0.]], grad_fn=<AddmmBackward>)
a2:  tensor([[0.5000, 0.5000]], grad_fn=<SigmoidBackward>)
```

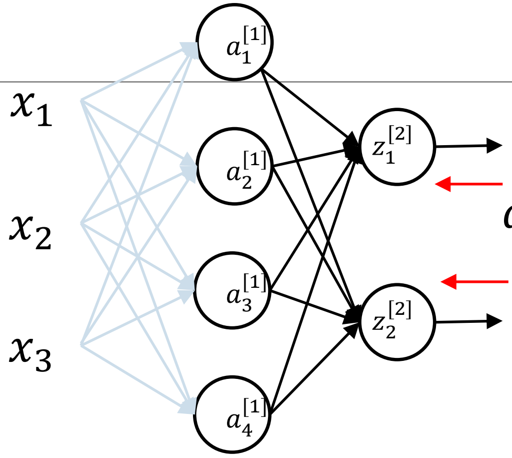
$$\sigma(0) = \frac{1}{1 + e^0} = 0.5$$

Backprop in PyTorch (layer 2)

Backward pass:

Tell PyTorch to save the gradients for intermediate variables

```
z2.retain_grad()
a1.retain_grad()
external_grad = torch.ones_like(a2) * 0.5
a2.backward(gradient=external_grad)
```



$$da^{[2]} = (0.5 \quad 0.5)$$

Manually check dz_2

```
# dz = a*(1 - a) * da
a2_np = a2.data.numpy()
dz2_manual = a2_np * (1 - a2_np) * external_grad.numpy()
dz2 = z2.grad.numpy()
print(np.equal(dz2_manual, dz2))
print('dz2: ', dz2)
```

Output:
```
[[ True  True]]
dz2:  [[0.125 0.125]]
```

$$dz^{[2]} = da^{[2]} * \frac{\partial a^{[2]}}{\partial z^{[2]}}$$

$$= da^2 * g'(z^{[2]})$$

$$= da^2 * \sigma'(z^{[2]})$$

Derivatives of sigmoid function:

$$\sigma'(z^{[2]})$$

$$= \sigma(z^{[2]}) \cdot \left(1 - \sigma(z^{[2]})\right)$$

$$= a^{[2]} \cdot (1 - a^{[2]}))$$

$$(0.5 \quad 0.5) * (0.5 \quad 0.5)(1 - (.5 \quad .5)))$$

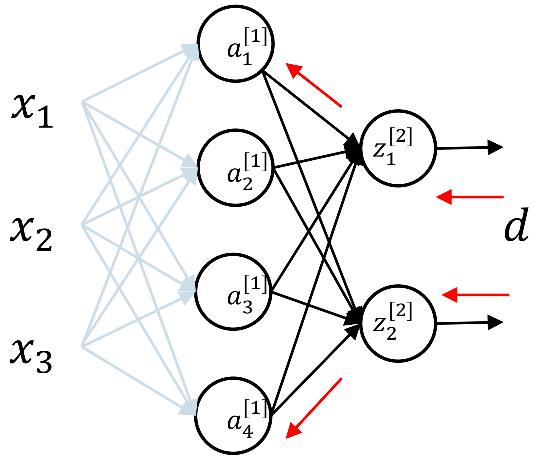
$$= (.25 \quad .25) = (.125 \quad .125)$$

Backprop in PyTorch (layer 2) contd.

$$W^{[2]} = \begin{bmatrix}0000\\0000\end{bmatrix}$$

We know: $dz^{[2]} = (.125 \quad .125)$ so $db^{[2]} = dz^{[2]} = (.125 \quad .125)$

$$da^{[1]} = dz^{[2]}W^{[2]} = (.125 \quad .125)\begin{bmatrix}0000\\0000\end{bmatrix} = [0. \quad 0. \quad 0. \quad 0.]$$



$da^{[2]} = (0.5 \quad 0.5)$

$da^{[1]} = ?$

```
print('dW2: ', linear2.weight.grad)
print('db2: ', linear2.bias.grad)
print('dz2: ', z2.grad)
print('da1: ', a1.grad)
```

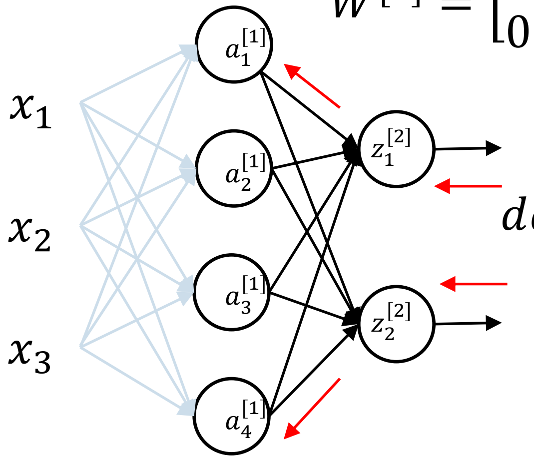
```
dW2:  tensor([[1.7500, 4.0000, 6.2500, 8.5000],
        [1.7500, 4.0000, 6.2500, 8.5000]])
db2:  tensor([0.1250, 0.1250])
dz2:  tensor([[0.1250, 0.1250]])
da1:  tensor([[0., 0., 0., 0.]])
```

Backprop in PyTorch (layer 2) contd.

$W^{[2]} = \begin{bmatrix} 0000 \\ 0000 \end{bmatrix}$ $\quad dW_2 = dz^{[2]T} a^{[1]}$

$$= \begin{pmatrix} 0.125 \\ 0.125 \end{pmatrix} (14\ 32\ 50\ 68)$$

$$= \begin{pmatrix} 1.75 & 4. & 6.25 & 8.5 \\ 1.75 & 4. & 6.25 & 8.5 \end{pmatrix}$$



$da^{[2]} = (.5 \quad .5)$

$a^{[1]} = [14. \quad 32. \quad 50. \quad 68]$ $\qquad dW^{[2]} = ?$

Manually check dW_2

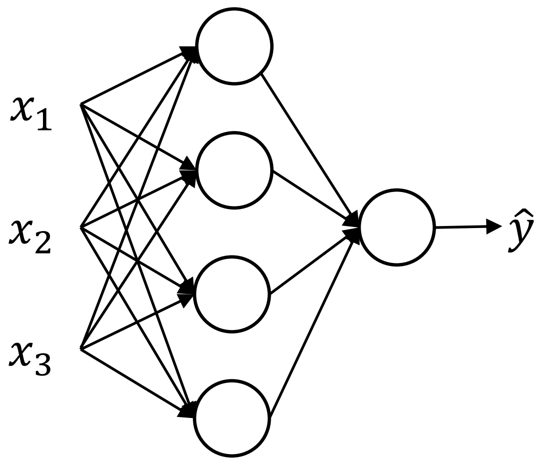
```
a1_np = a1.data.numpy()
print('a1: ', a1_np)
dW2 = np.dot(dz2.T, a1_np)
print('dW2: ', dW2)
```

Output:

```
dW2:  [[1.75 4.   6.25 8.5 ]
 [1.75 4.   6.25 8.5 ]]
```

Initializing W and B?

How to initialize *W* parameters? To zeros?



$$W^{[1]} = \begin{bmatrix} 000 \\ 000 \\ 000 \\ 000 \end{bmatrix} \Rightarrow a_1^{[1]} = a_2^{[1]} = a_3^{[1]} = a_4^{[1]}$$

$$dW^{[2]} = dz^{[2]T} a^{[1]}$$

All same

$$W^{[2]} = [0,0,0,0]$$

All same

$$\Rightarrow da^{[1]} = W^{[2]T} dz^{[2]}$$

$dz^{[1]}$ All same

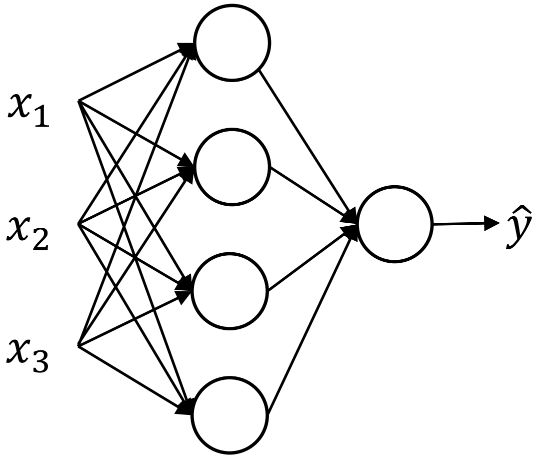
$\Downarrow$

$$dW^{[1]} = dz^{[1]T} x$$

Update to all rows are same

All hidden units computing the same function!
Pointless!

Use Random initialization for *W* Instead



$W^{[1]} =$ `np.random.randn(4, 3) * 0.01`

$B^{[1]} =$ `np.zeros((4, 1))`

$W^{[2]} =$ `np.random.randn(1, 4) * 0.01`

$B^{[2]} =$ `np.zeros((1, 1))`