Summary Notes

- **NN-1: Intro to NN**

  What are the issues with Classic ML Algos?
  - Require us to do manual feature engineering to be able to create complex boundaries
  - They might not always work well with big datasets, and we are living in a big-data regime
  - Works poorly for sparse data and for unstructured data (image/ text/ speech data)

  Clearly, all these algos have some limitations which call for a more powerful ML model that can work in the conditions mentioned above. This brings us to Neural Networks (NN).

  NN models power pretty much every minute of your digital life
  - Online Ads (Google, YouTube)
  - Data compression: Done using Autoencoders
  - Image enhancement: Eg Magic Eraser
  - Gmail: Eg Autocomplete, smart reply

  Where did the inspiration for a neuron come from?

  NN is loosely inspired by the biological neurons found in the human brain.

  In the brain, there exist biological neurons that are connected to each other, forming a network

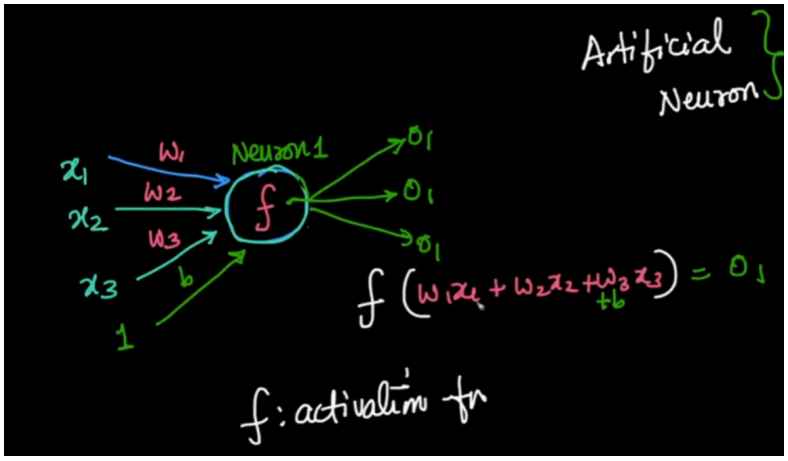
  In simple terms, we understood that
  - Neuron takes input(s)
  - Perform some computations.
  - Ultimately, it fires/passes the output to further neurons, for further processing.

  How does an Artificial neuron do its computations?

  Consider an artificial neuron.
  - It receives input features: x_1, x_2, x_3
  - Every input has a weight associated with it: w_1, w_2, w_3
    - These weights are multiplied by the input values, thereby telling us how important a given input is to the neuron.
  - Inputs are processed by taking the weighted sum.
  - Also, a bias term is added.
    - The net input becomes: $w_1x_1 + w_2x_2 + w_3x_3 + b = z$ (let)
  - There is a function, called activation function f, which is associated with a neuron
    - The neuron applies this function on the net input value: $f(z) = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$
  - The result of this function becomes the output $o_1 = f(z)$
  - This output is then forwarded to other neurons.

  This flow of computations is known as **Forward Propagation.**

  Notice that we are going from left to right during Forward Propagation.



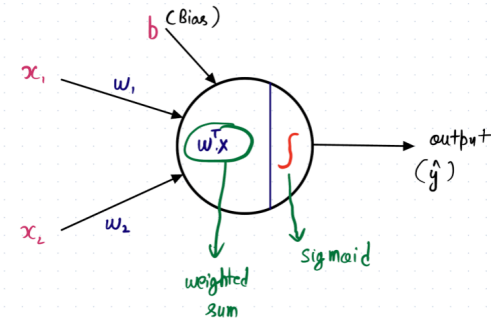
What happens if we put the sigmoid function as the activation function?

We get a Logistic Regression model, as the output of this becomes:

$o_1 = \text{sigmoid}(w_1x_1 + w_2x_2 + w_3x_3 + b)$

This neuron, where the activation is a sigmoid function is called a Logistic Regression Unit (LRU)

We can diagrammatically represent a neuron with 2 inputs as:



What if the model looked the same, but the activation function is different? Would it still be called the same NN?

If we replace the activation function to a hinge loss function, we get a Linear SVM model.

Forward propagation here would look like:-

- $z = w_1x_1 + w_2x_2 + \ldots + w_dx_d + b$
- $\hat{y} = f_{hinge}(z)$

Note:

- A single neuron is able to represent such powerful models, just imagine what will happen if we use multiple neurons.
- Those models would be able to represent really complex relations.

A brief history of Artificial Neural Networks

Perceptron

- Perceptron is the very first NN-based model. It was designed by Rosenblatt in 1957

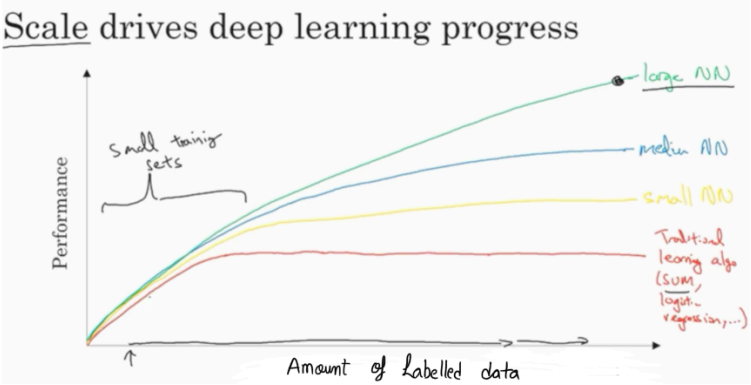
- It is different from LRU, because it used a step function for activation, which is:

$$f_{perceptron}(x_i, w, b) = \begin{cases} 1, & \text{if } w^T x_i + b > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Challenges:

- As we tried to increase the complexity of NN, they performed poorly.
- There was no optimal way of training the NN.
- This put the whole area in deep freeze for a decade or so.
- A breakthrough came in 1986 when **backpropagation** was introduced by Geoff Hinton and his team.
- This also brought up a lot of hype for artificial intelligence, But, all of that hype died down by 1990s
- There were two main bottlenecks:-
    - We didn't have computational power
    - Nor did we have enough data to train
    - Backprop was failing for NNs with more depth.
- This time period is called as **AI winter,** where the funding for AI dried up by 1995
- Meanwhile, Geoff Hinton continued his research and finally came up with a solution to this problem in 2006.
- Also, the discovery of using ReLu and Leaky ReLu as activation functions was another breakthrough.

How do NN fare against classical ML (based on training data)?



An LRU can only help in the case of binary classification (0 or 1).

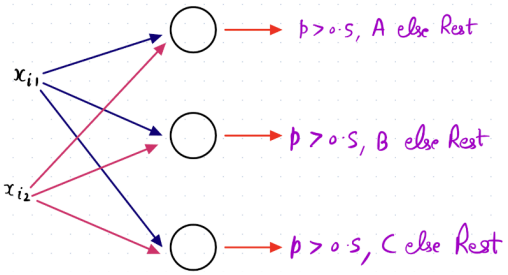
How can we adapt a NN to do multi-class classification?

Suppose we wish to do multi-class classification for 3 classes: A, B, and C.

Recall multi-class classification:

- We calculated the probability that a given data point belongs to class A, B, or C respectively.
- Then, we returned the class with the highest probability as the answer.

- This gives us the intuition that perhaps, our output layer should have 3 outputs. One for each class.



We cannot perform multi-class classification using a sigmoid because we might get $p > 0.5$ for more than one class.

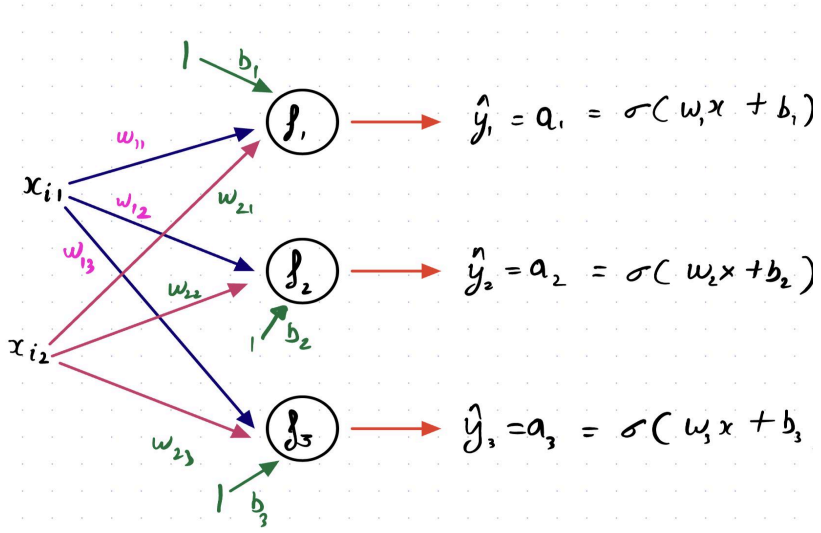
- Model will predict the presence of multiple classes in the output - [1, 1, 0], [1, 1, 1], [1, 0, 1]
- Conclusion: We want these probabilities values to sum to 1, as we had in Logistic Regression (p and 1-p).

It should be $p_A + p_B + p_C = 1$

In order to do this, we use the **softmax function** as activation in the neurons of the **output layer**.

$$p_i = \frac{e^{z_i}}{\sum_{i=0}^{k} e^{z_i}}$$

The model now looks like:-



Note:

- There will be 2*3 = 6 weights, and it'll be stored as a 2x3 matrix: W_{2x3}
- There will be 3 biases, one for each neuron, and it'll be stored as a 1x3 matrix: b_{1x3}

How will we calculate the loss for this multi-class classification NN Model?
We use Categorical Cross Entropy.

Cross Entropy (CE_i) for i^{th} datapoint will be:

$$CE_i = -\sum_{j=1}^{k} y_{ij} log(P_{ij})$$

where,
k -> number of classes
y_{ij} -> one hot encoded label. Ex: [1,0,0], [0,1,0], or [0,0,1]
P_{ij} -> Calculated Probability of datapoint belonging to class j
This can be seen as log loss extended to the multiclass setting. For k=2, we will get log loss formulation.

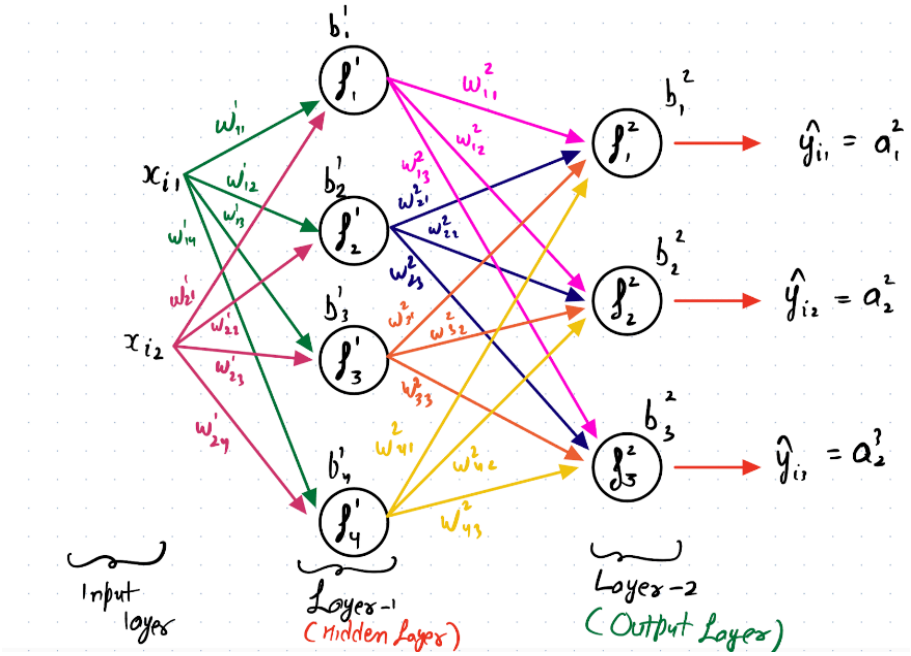
How to train NN?
Let, m -> no of training examples
d -> no of features
n -> no of classes/neurons in the output layer

Process to train a NN is:-

- Randomly Initialise parameters: W and b matrices
- Do forward propagation
  - $Z \ = \ X_{m\,x\,d} . W_{\ d\,x\,n} \ + \ b_{\ 1\,x\,n}$
  - $A \ = \ activation(Z\ )$
- Calculate the Loss
- Repeat until Loss converges
  - update $w_i = w_i - lr\ *\ (\partial Loss/\partial w_i)$
  - calculate the output using hypothesis and updated params
  - calculate the Loss

- **NN-3: Backpropagation and Activation functions**
  What are MLPs?
  If we wish to get a more complex decision boundary, we need to add another layer of neurons in the model. This is known as the **hidden layer**.
  - These models are known as MLPs (Multi Layer Perceptrons). Or N-layered NN
  - They are based on the idea of function composition.
  - We can have as many hidden layers as we want, however, the greater the number, the higher the risk of overfitting.
  - The activation of hidden layers also needs to always be **non-linear,** otherwise, we will not be able to get complex features.

  A 2-layer model looks like:-



Since there are multiple layers, we modify the notation to cater to it

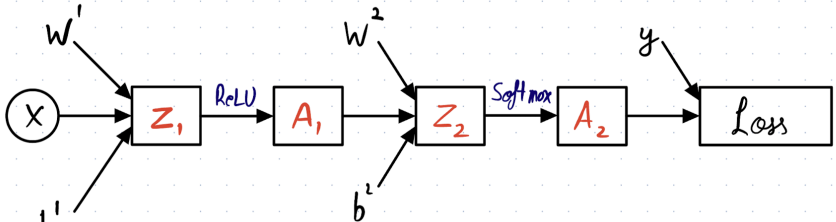
- Weights: $\mathbf{W}^{L}_{ij}$
- Bias: $\mathbf{b}^{L}_{i}$

Note:-

- Weights in layer-1 will be stored as a 2x4 matrix: $W^{1}_{2\,x\,4}$
- Biases in layer-1 will be stored as a 1x4 matrix: $b^{1}_{1\,x\,4}$
- Weights in layer-2 will be stored as a 4x3 matrix: $W^{2}_{4\,x\,3}$
- Biases in layer-2 will be stored as a 1x3 matrix: $b^{2}_{1\,x\,3}$

Why do we need to add a hidden layer to increase complexity?
The idea is that Stacking a non-linearity over a linear function, and repeating the process helps create complex features

What does forward propagation look like for this NN?
Let h -> no of neurons in the hidden layer.
m -> no of training examples
d -> no of features
n -> no of classes/neurons in the output layer



Forward propagation:-

$$Z^1_{m\,x\,h} = X_{m\,x\,d} . W^1_{d\,x\,h} + b^1_{1\,x\,h}$$

$$A^1_{m\,x\,h} = f^1(Z^1_{m\,x\,h})$$

$$Z^2_{m\,x\,n} = A^1_{m\,x\,h} . W^2_{h\,x\,n} + b^2_{1\,x\,n}$$

$$A^2_{m\,x\,n} = f^2(Z^2_{m\,x\,n})$$

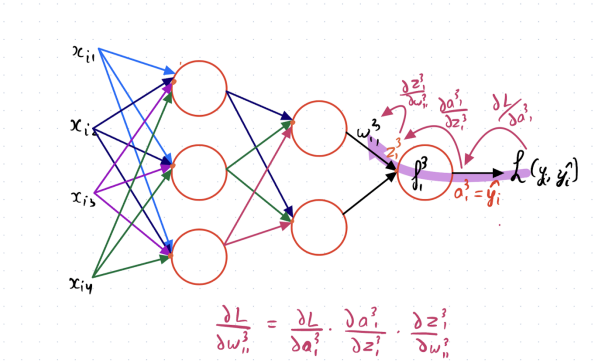
How do we train this complex NN? What is Backpropagation?

- In order to update the parameters, we need to find their gradients. For this, we use the Backpropagation algorithm, where we traverse from right to left in the NN.
- It is based on the concept of chain rule of differentiation.

Gradient of w^3_{11}

We'll encounter a^3_1 while going from loss towards w^3_{11}

Therefore gradient: $\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial a^3_1} . \frac{\partial a^3_1}{\partial z^3_1} . \frac{\partial z^3_1}{\partial w^3_{11}}$

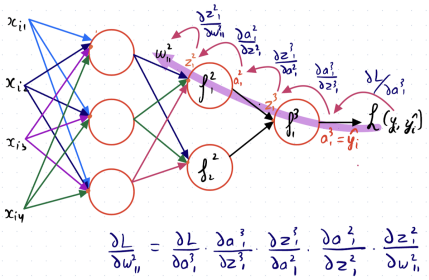


Gradient of w^2_{11}

We'll encounter a^3_1 and a^2_1 while going from loss towards w^2_{11}

Therefore gradient:

$$\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial a^3_1} . \frac{\partial a^3_1}{\partial z^3_1} . \frac{\partial z^3_1}{\partial a^2_1} . \frac{\partial a^2_1}{\partial z^2_1} . \frac{\partial z^2_1}{\partial w^2_{11}}$$



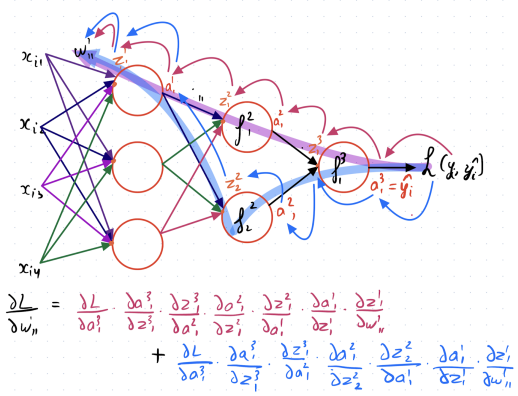
Gradient of w^1_{11}

There are 2 possible paths to reach **w^1_{11}**

Path-1 : L -> a^3_1 -> a^2_1 -> a^1_1 -> w^1_{11}

Path-2 : L -> a^3_1 -> a^2_2 -> a^1_1 -> w^1_{11}

We need to combine the derivatives from path 1 and 2 by adding them up.



Therefore gradient:

$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial a^3_1}.\frac{\partial a^3_1}{\partial z^3_1}.\frac{\partial z^3_1}{\partial a^2_1}\frac{\partial a^2_1}{\partial z^2_1}.\frac{\partial z^2_1}{\partial a^1_1}.\frac{\partial a^1_1}{\partial z^1_1}.\frac{\partial z^1_1}{\partial w^1_{11}} + \frac{\partial L}{\partial a^3_1}.\frac{\partial a^3_1}{\partial z^3_2}.\frac{\partial z^3_1}{\partial a^2_1}.\frac{\partial a^2_2}{\partial z^2_2}.\frac{\partial z^2_2}{\partial a^1_1}.\frac{\partial a^1_1}{\partial z^1_1}.\frac{\partial z^1_1}{\partial w^1_{11}}$$

What are the different activation functions?

- Sigmoid function
- Hyperbolic tan function: $\frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLu: $ReLu(z) = max(z, 0)$
- Leaky ReLu: $Leaky\ ReLu(z) = max(z, \alpha z)$; α is a small gradient that we add

What is the vanishing gradient problem?

- Downside of both sigmoid and tanh is that their gradient is ~0, for most of the values of z
- This hampers the gradient descent process, as the calculated gradients become very small.
- For eg Suppose we wish to update weight w^1_{11}. its gradient is calculated as:

$$\frac{\partial L}{\partial w^1_{11}} = \frac{\partial L}{\partial a^3_1}\left[\frac{\partial a^3_1}{\partial a^2_{11}}.\frac{\partial a^2_{11}}{\partial a^1_{11}}.\frac{\partial a^1_{11}}{\partial w^1_{11}} + \frac{\partial a^3_1}{\partial a^2_{21}}.\frac{\partial a^2_{21}}{\partial a^1_{12}}.\frac{\partial a^1_{12}}{\partial w^1_{11}}\right]$$

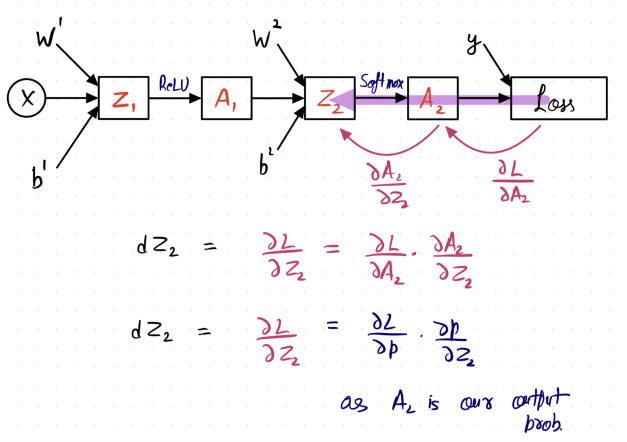
- So, the product of these terms inside the bracket will become very small.
- In fact, as the number of layers in the NN increase, this product will become smaller and smaller.

Backprop for MLP

- **Calculating dZ²**

$dZ^2 = \frac{\partial L}{\partial Z^2} = \frac{\partial L}{\partial A^2}.\frac{\partial A^2}{\partial Z^2}$

$dZ^2 = \frac{\partial L}{\partial p}.\frac{\partial p}{\partial Z^2} = p_i - I(i == k)$

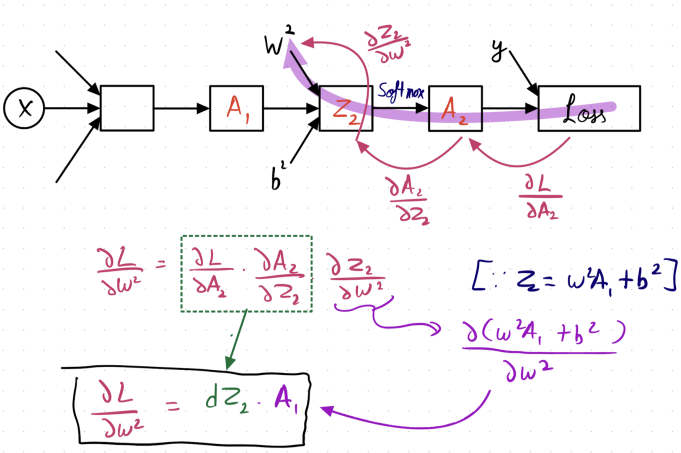


- **Calculating dW^2**

$$dW^2 = \frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial W^2}$$

$$dW^2 = dZ^2 \cdot \frac{\partial Z^2}{\partial W^2}$$

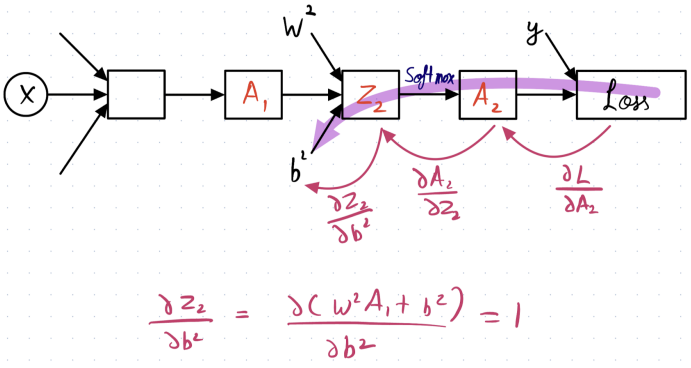
$$dW^2 = dZ^2 \cdot A^1$$



- **Calculating db^2**

$$db^2 = \frac{\partial L}{\partial b^2} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial b^2}$$

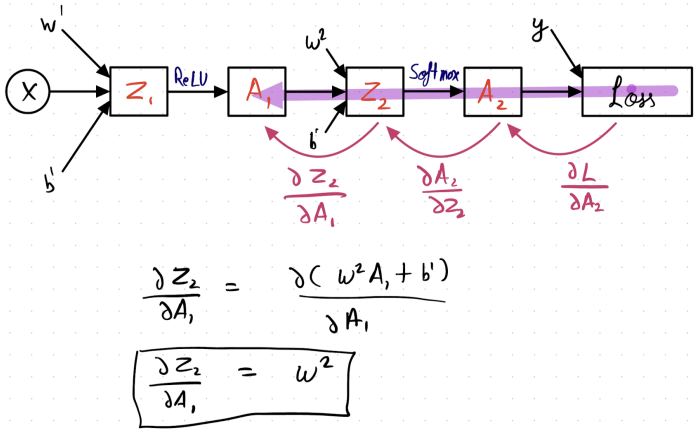
$$db^2 = dZ^2 \cdot \frac{\partial Z^2}{\partial b^2} = dZ^2$$



- **Calculating dA^1**

$$dA^1 = \frac{\partial L}{\partial A^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1}$$

$$dA^1 = dZ^2 \cdot \frac{\partial Z^2}{\partial A^1} = dZ^2 \cdot W^2$$

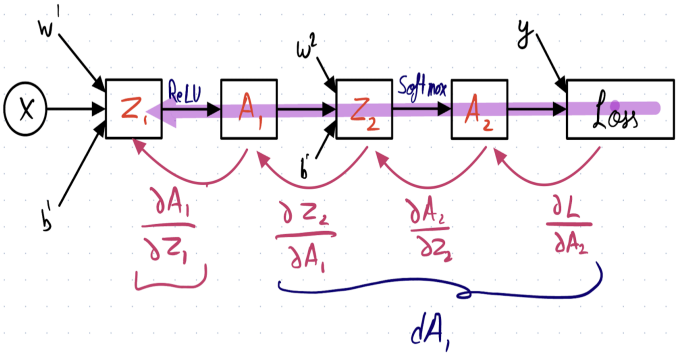


- **Calculating dZ^1**

$$dZ^1 = \frac{\partial L}{\partial Z^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1}$$

$$dZ^1 = dA^1 . \frac{\partial A^1}{\partial Z^1}$$

$$dZ^1 = dA^1 . \frac{\partial A^1}{\partial Z^1}$$



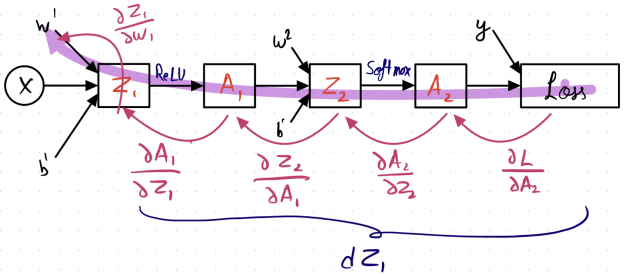
$$\frac{\partial L}{\partial Z_1} = \partial A_1 \cdot \frac{\partial A_1}{\partial Z_1}$$ → Can be 0 or 1

$$= \begin{cases} \partial A_1 \times 0 & \text{if } Z_1 \leq 0 \\ \partial A_1 \times 1 & \text{if } Z_1 > 0 \end{cases}$$

- **Calculating dW^1**

$$dW^1 = \frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial Z^1}{\partial W^1}$$

$$dW^1 = dZ^1 \frac{\partial Z^1}{\partial W^1} = dZ^1 . X$$

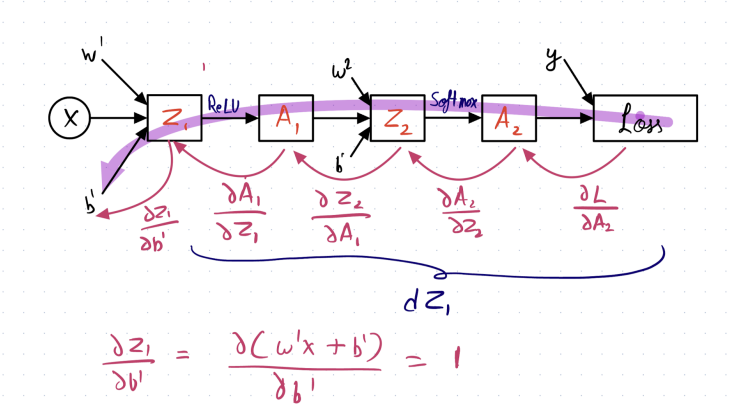


$$\frac{\partial Z_1}{\partial w'} = \frac{\partial (w'x + b')}{\partial w'} = x$$

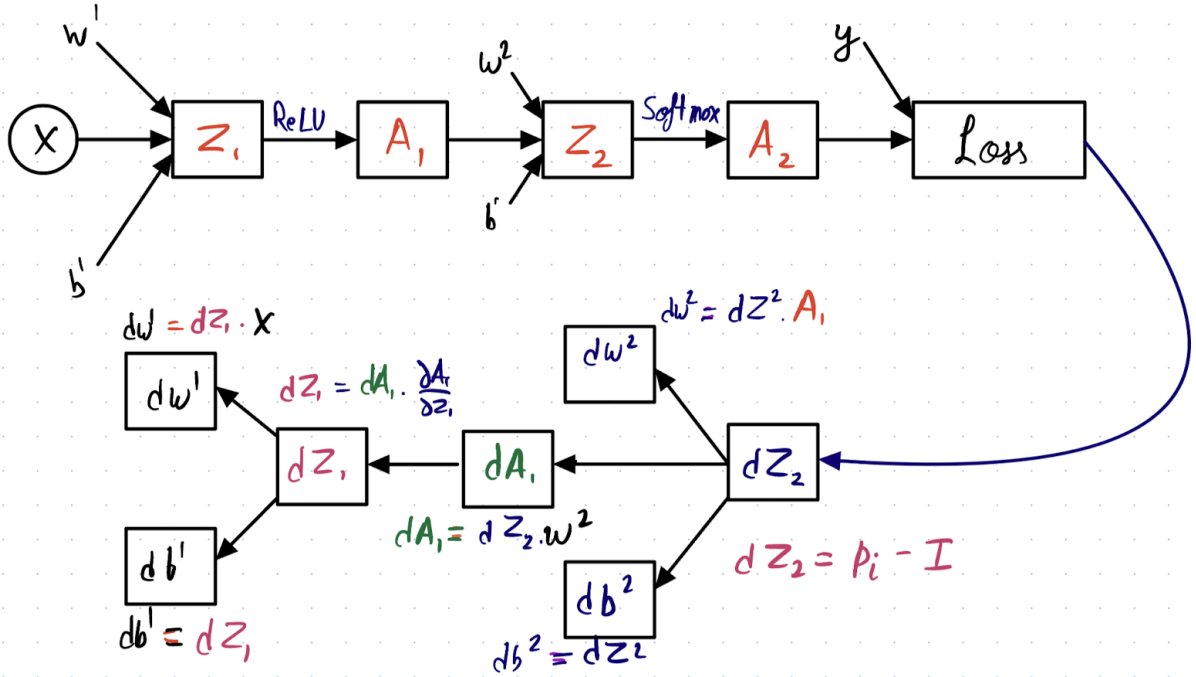
- **Calculating db^1**

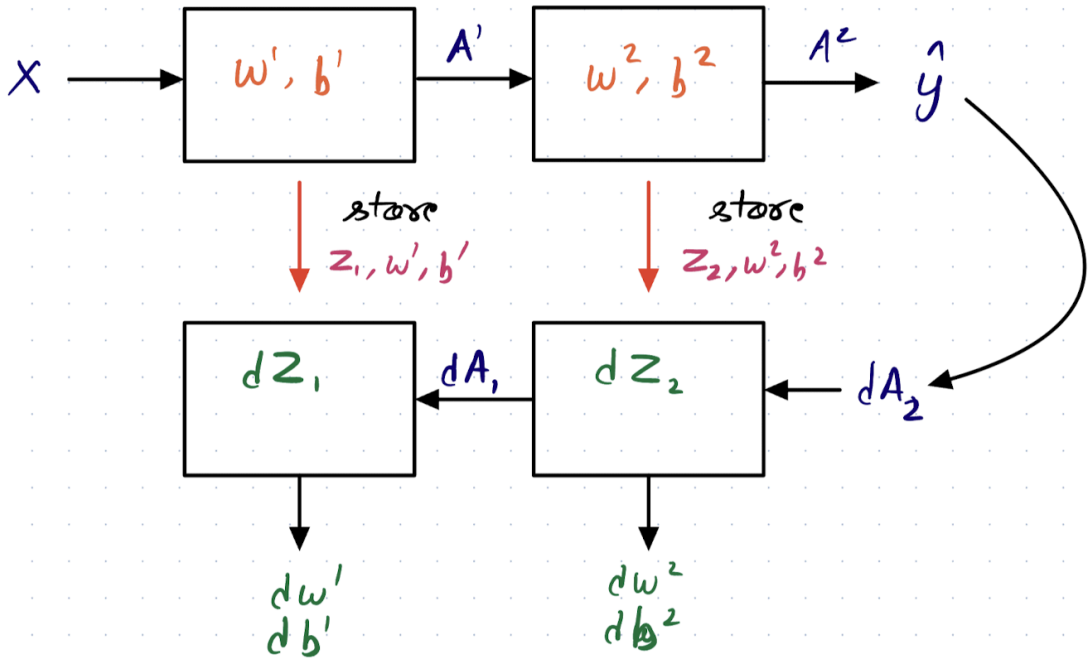
$$db^1 = \frac{\partial L}{\partial b^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial Z^2} \frac{\partial Z^2}{\partial A^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial Z^1}{\partial b^1}$$

$$db^1 = dZ^1 \cdot \frac{\partial Z^1}{\partial b^1} = dZ^1 \cdot 1 = dZ^1$$



Summarizing forward and backward prop for MLP





While performing forward prop,

- we store/cache the value of Zj, Wj, bj in order to use them during back prop

Can we use Neural Networks for the Regression task?

- Yes, if the activation function for the output layer is a linear function, then NN will do regression.
- The activations for intermediate layers still need to be non-linear, otherwise, NN will not be able to map complex relationships.
-