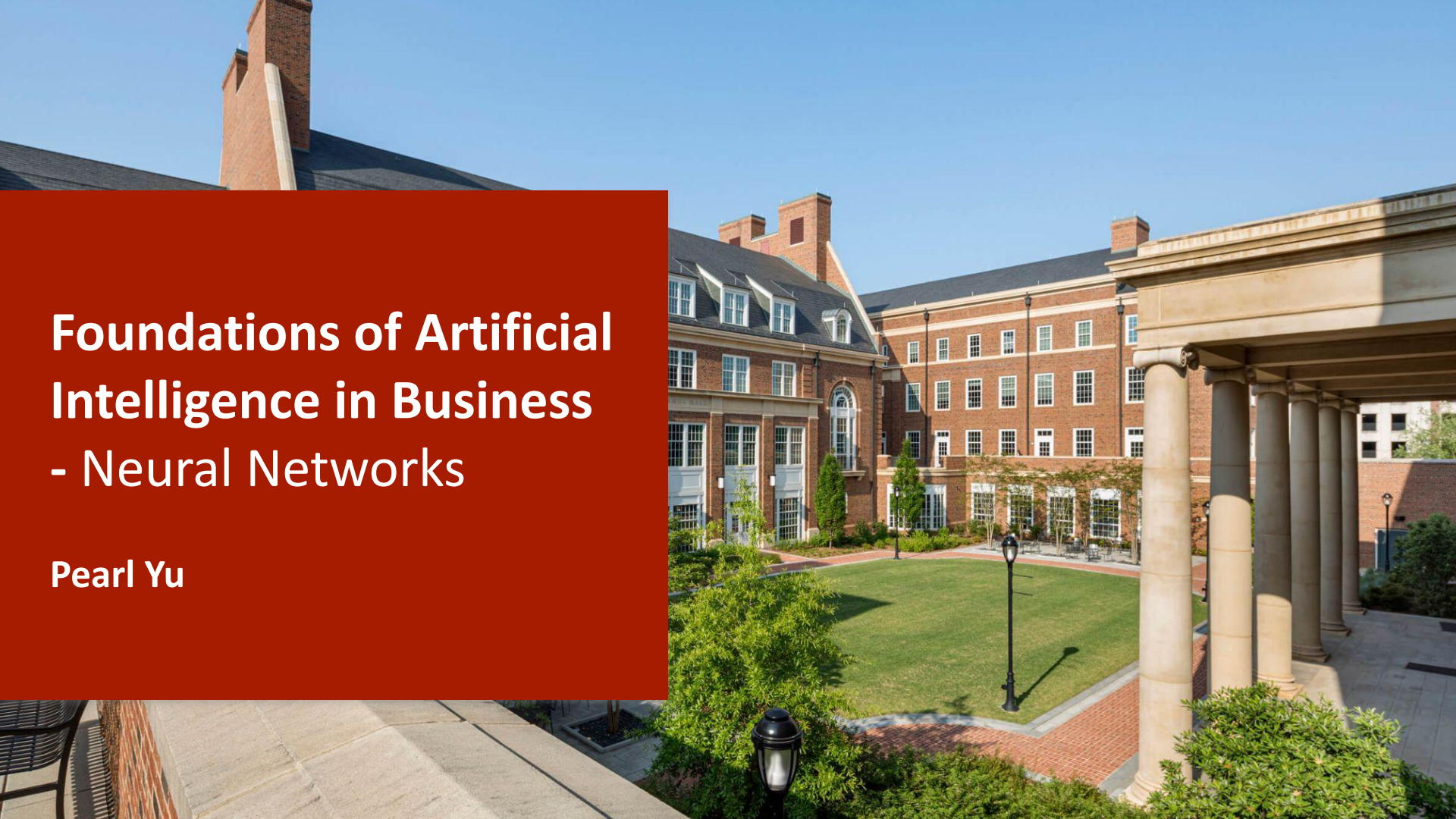
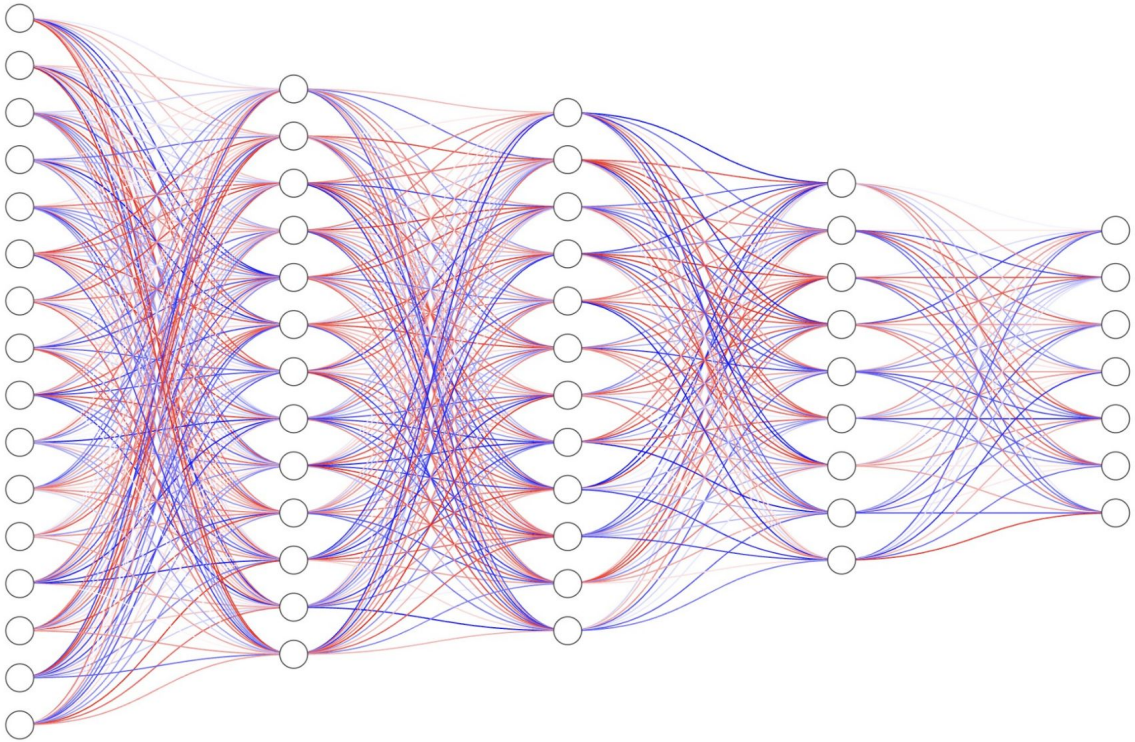


Foundations of Artificial Intelligence in Business - Neural Networks

Pearl Yu

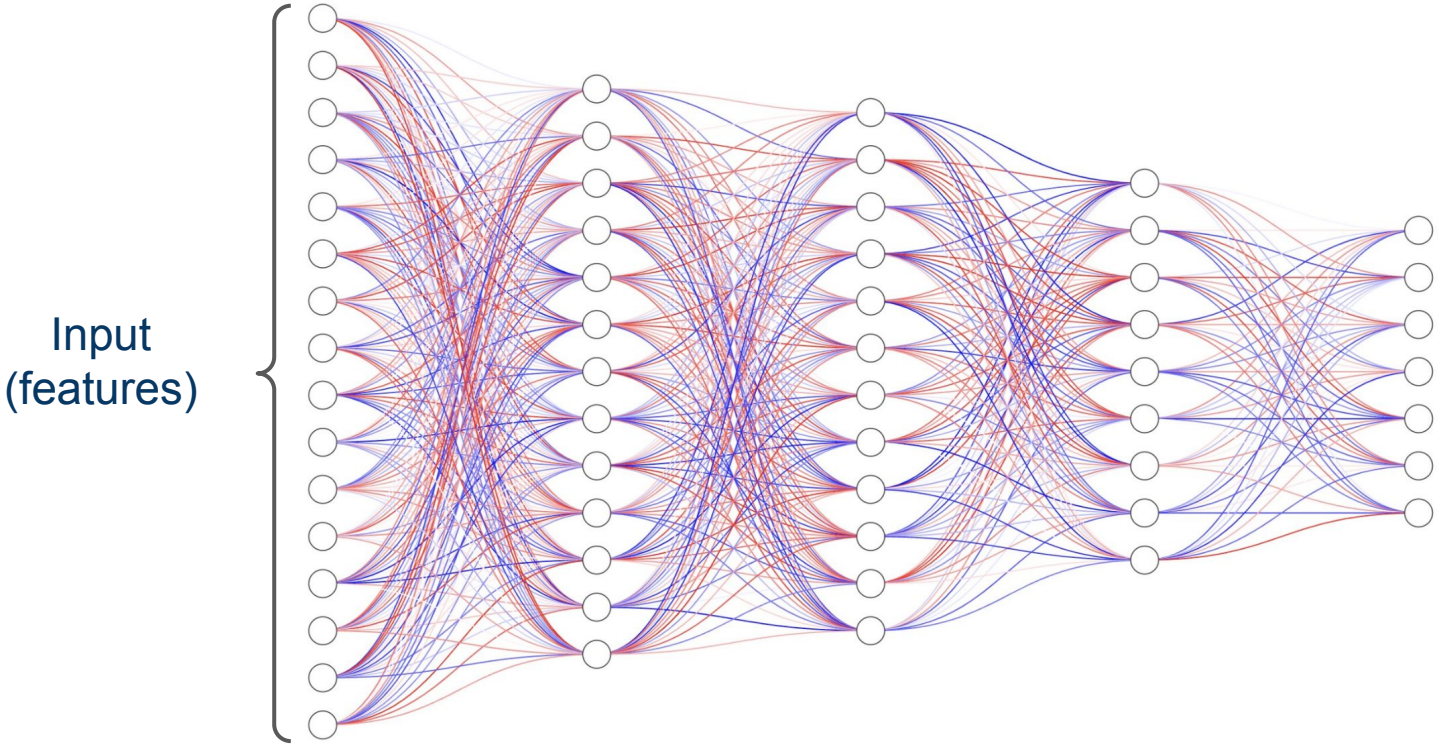
Deep Neural Network

- Neural networks are composed of **nodes and connections**.



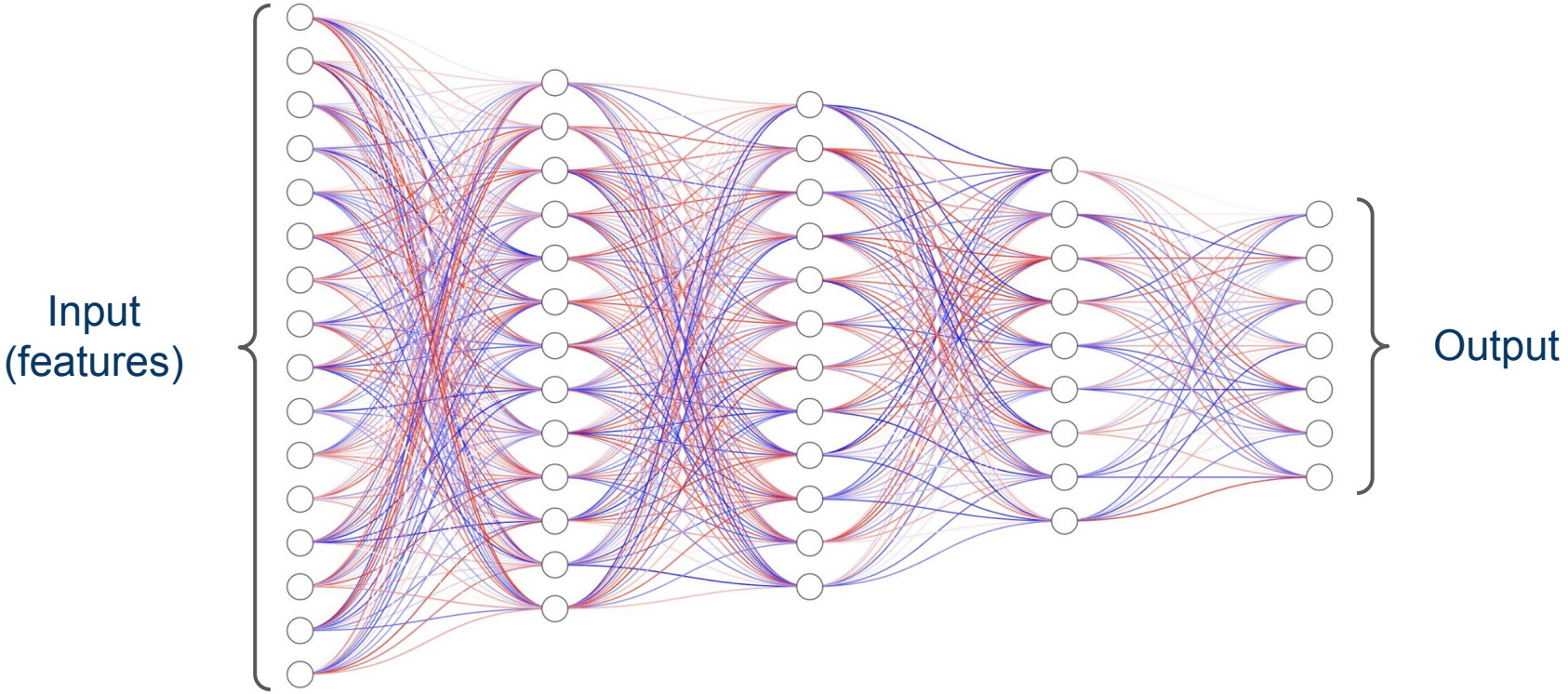
Deep Neural Network

- Neural networks are composed of **nodes and connections**.



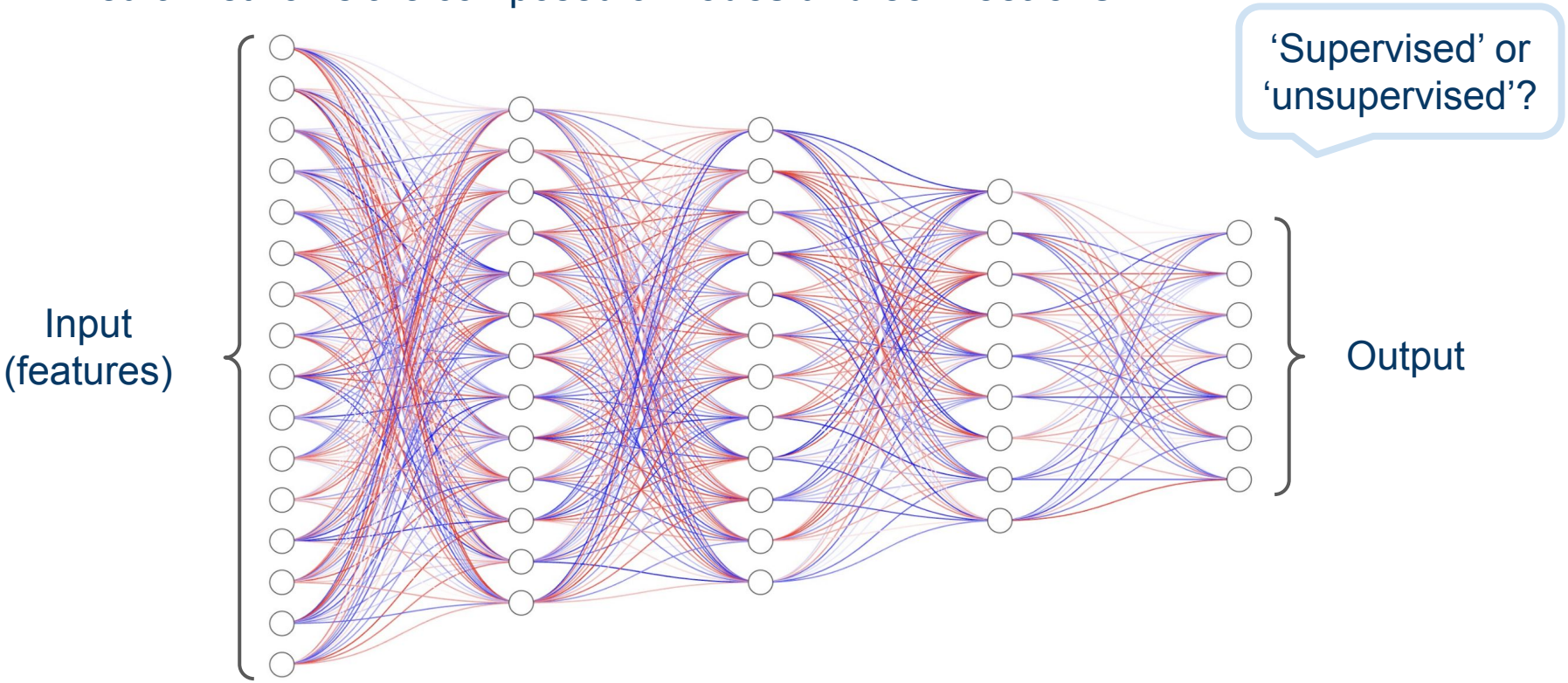
Deep Neural Network

- Neural networks are composed of **nodes and connections**.



Deep Neural Network

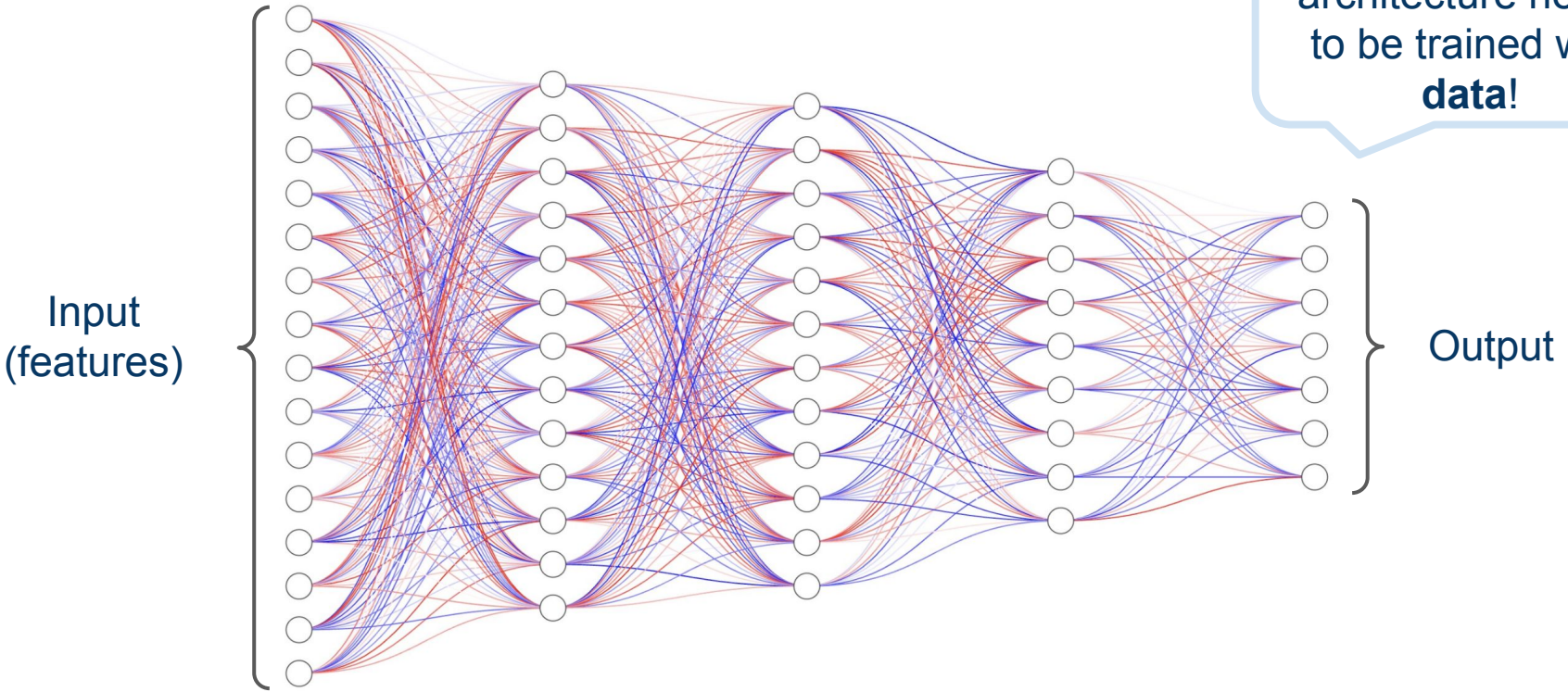
- Neural networks are composed of **nodes and connections**.



Deep Neural Network

- Neural networks are composed of **nodes and connections**.

The whole architecture needs to be trained with **data**!

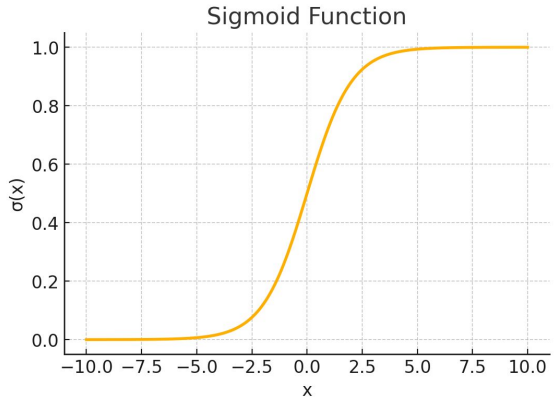


Logistic Regression

$$\hat{y}(\mathbf{x}) = sigmoid(w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p)$$

$$sigmoid: \frac{1}{1+e^{-z}}$$

linear combination of inputs

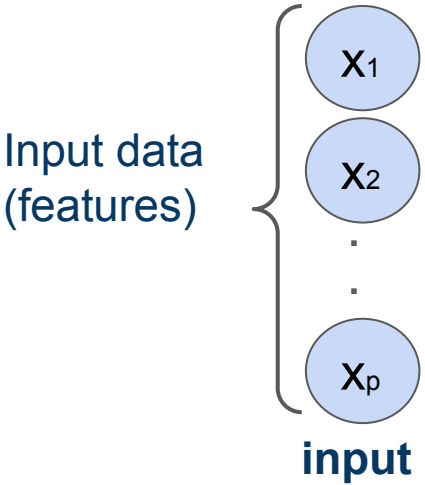


The sigmoid function maps the output of linear combinations to a probability in [0,1]

Basic Neural Architecture

- Logistic Regression, drawn in **a neuro-inspired way**

$$\hat{y}(\mathbf{x}) = sigmoid(\underbrace{w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p}_{\text{linear combination of inputs}})$$

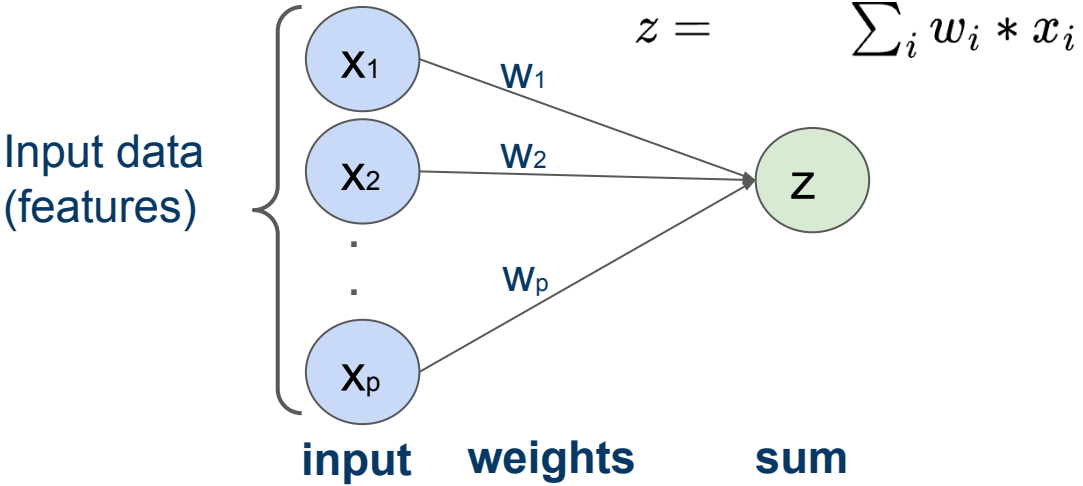


Basic Neural Architecture

- Logistic Regression, drawn in **a neuro-inspired way**

$$\hat{y}(\mathbf{x}) = sigmoid(\underbrace{w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p}_{\text{linear combination of inputs}})$$

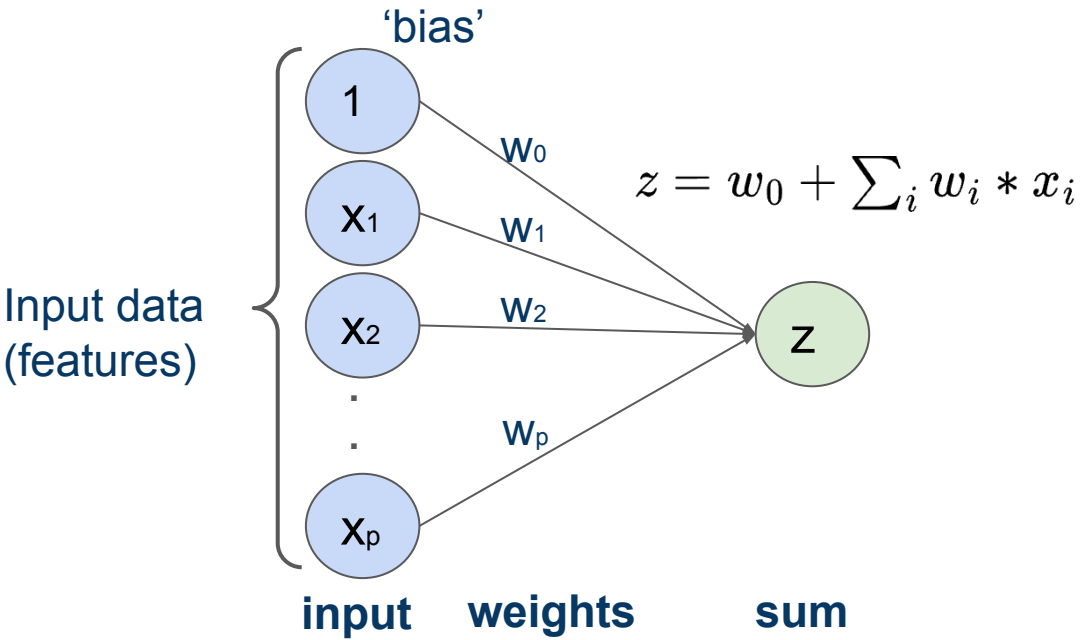
What do these weights do?



Basic Neural Architecture

- Logistic Regression, drawn in **a neuro-inspired way**

$$\hat{y}(\mathbf{x}) = sigmoid(\underbrace{w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p}_{\text{linear combination of inputs}})$$



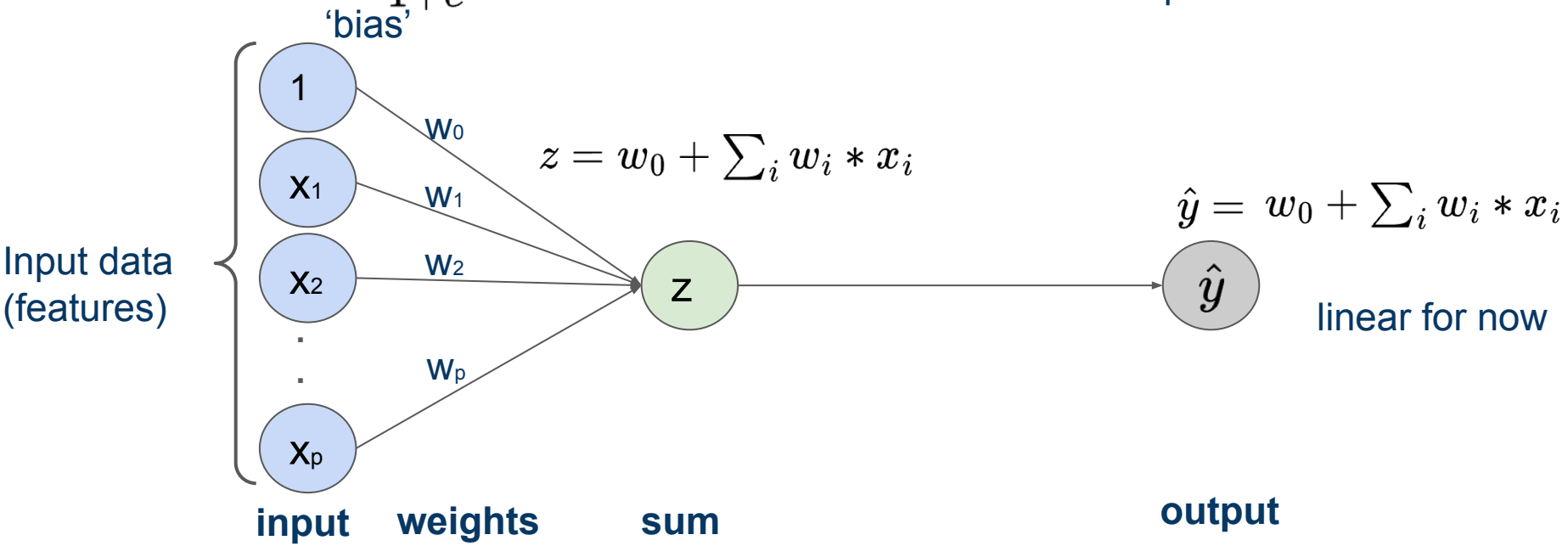
Basic Neural Architecture

- Logistic Regression, drawn in **a neuro-inspired way**

$$\hat{y}(\mathbf{x}) = sigmoid(w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p)$$

$$sigmoid: \frac{1}{1+e^{-z}}$$

linear combination of inputs



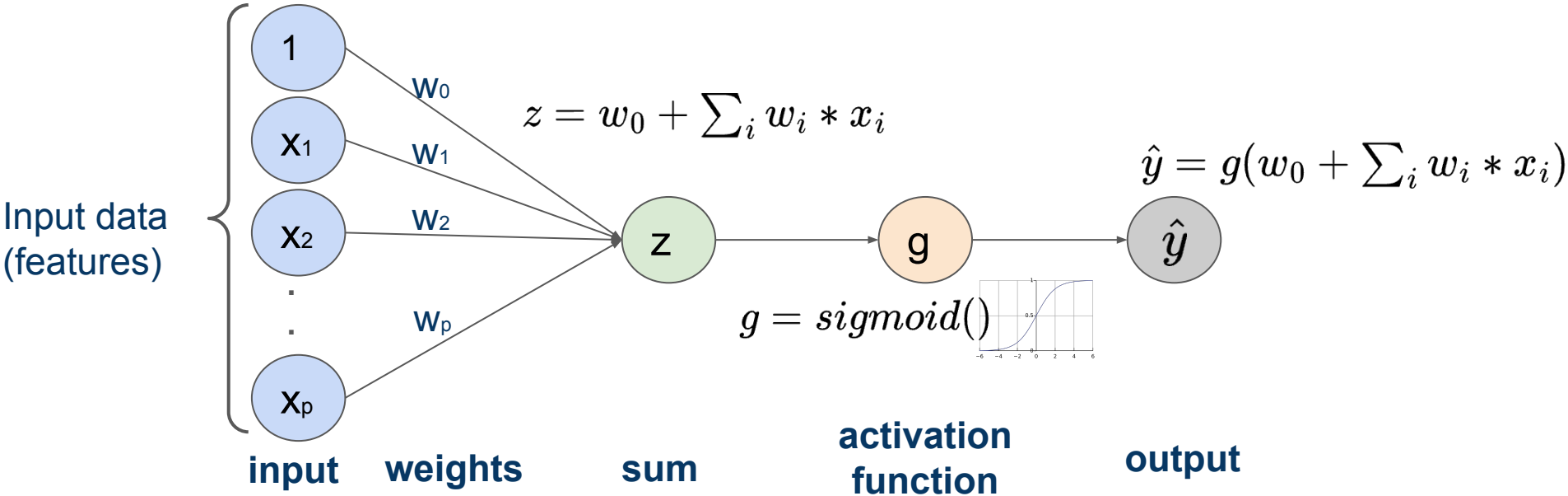
Basic Neural Architecture

- Logistic Regression, drawn in **a neuro-inspired way**

$$\hat{y}(\mathbf{x}) = sigmoid(w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p)$$

$$\frac{1}{1+e^{-z}}$$

linear combination of inputs



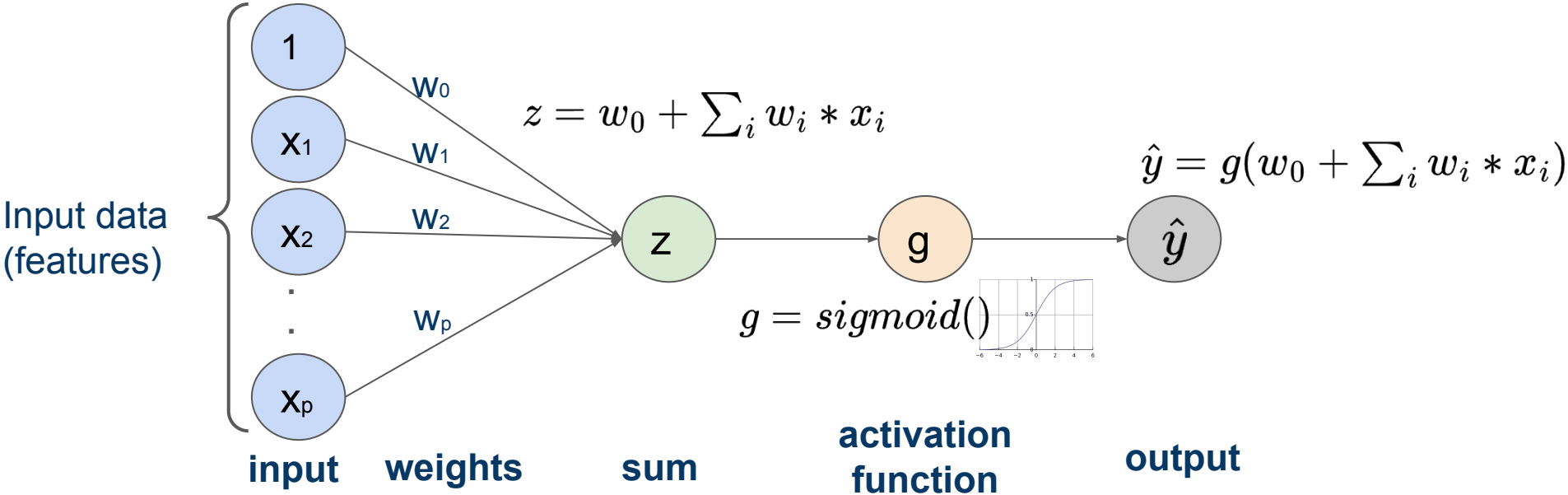
Basic Neural Architecture

- Logistic Regression, drawn in **a neuro-inspired way**

$$\hat{y}(\mathbf{x}) = sigmoid(w_0 + w_1 * x_1 + w_2 * x_2 + \ldots + w_p * x_p)$$

$$sigmoid: \frac{1}{1+e^{-z}}$$

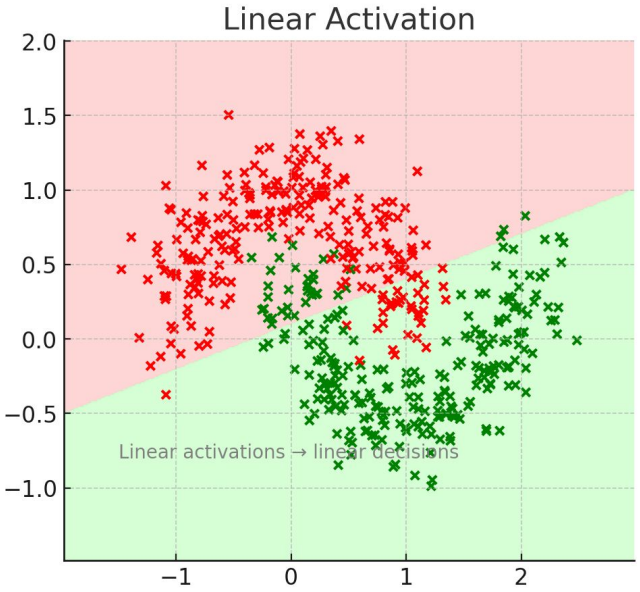
linear combination of inputs

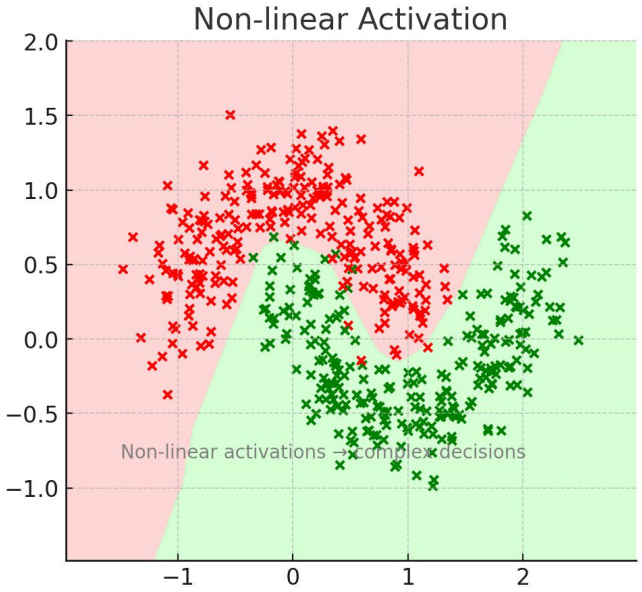


Activation Functions

$$z = w_0 + \sum_i w_i * x_i \longrightarrow g = g(z)$$

g is a non-linear activation function

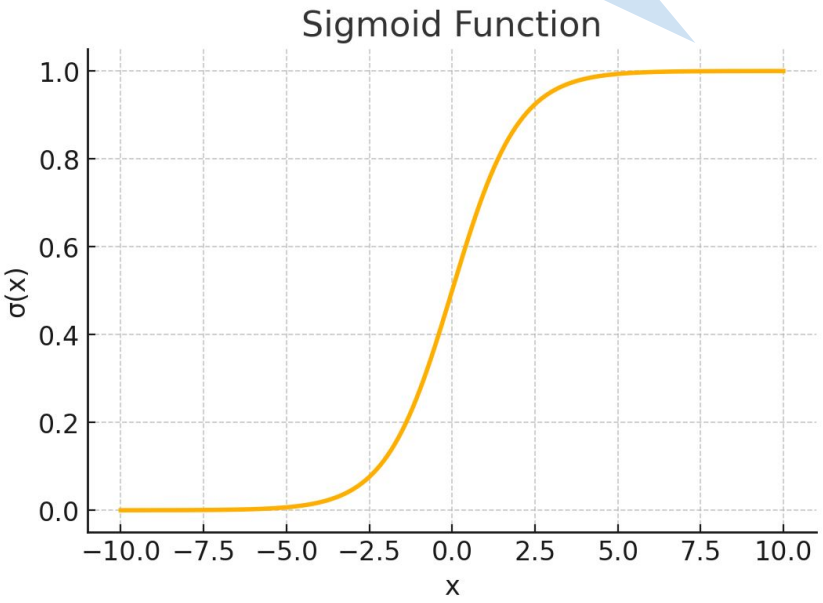




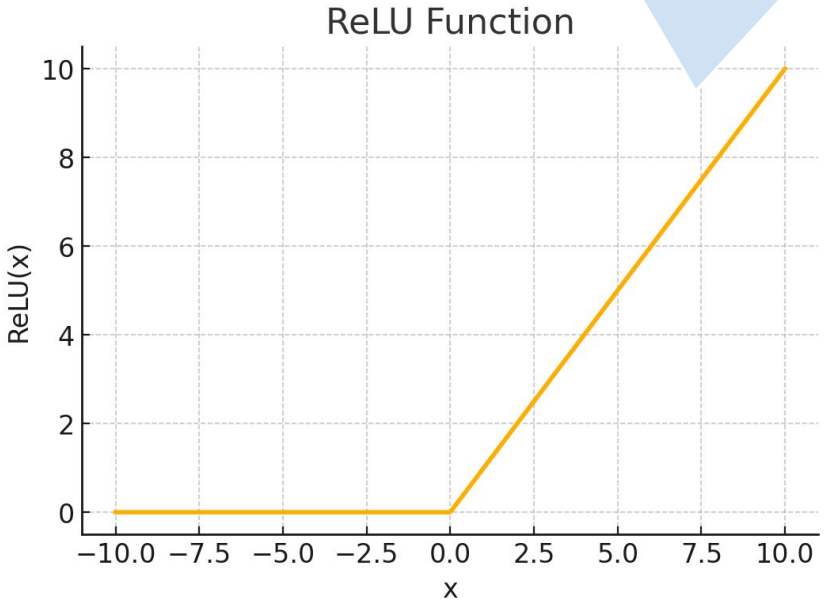
- **Non-linear** activation functions can help **approximate complex functions**.

Classic Activation Functions

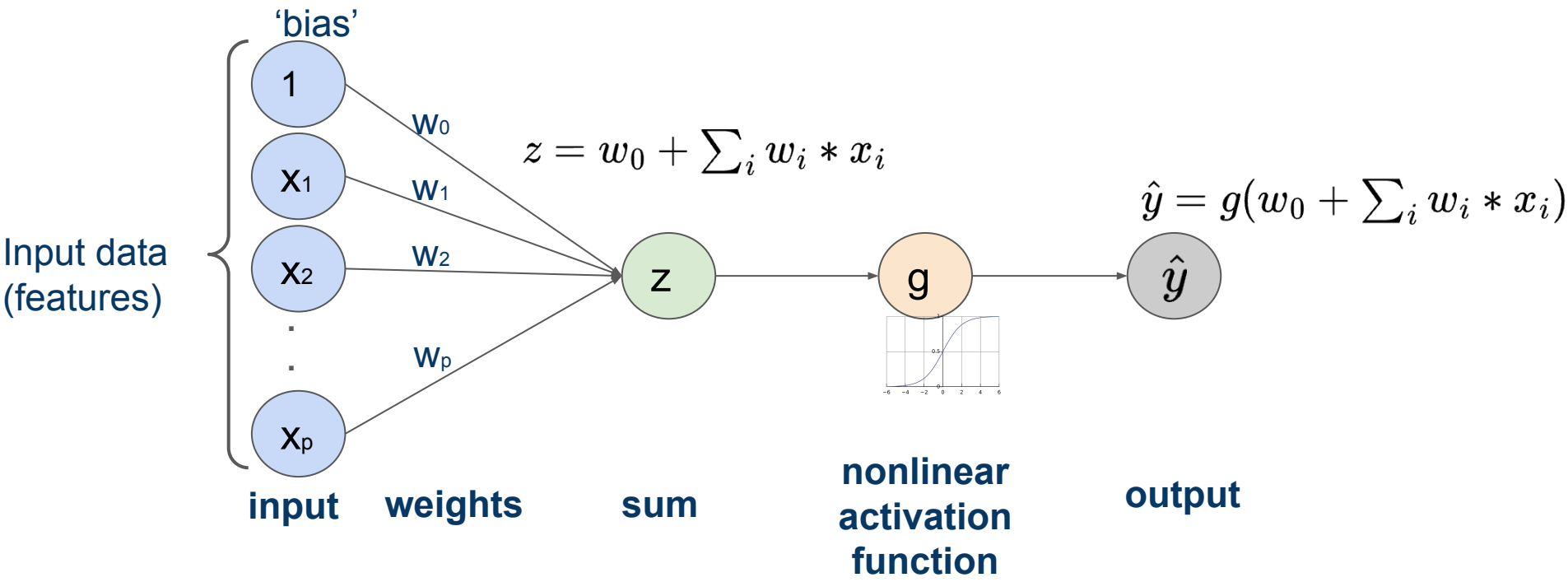
‘How important is this node?’



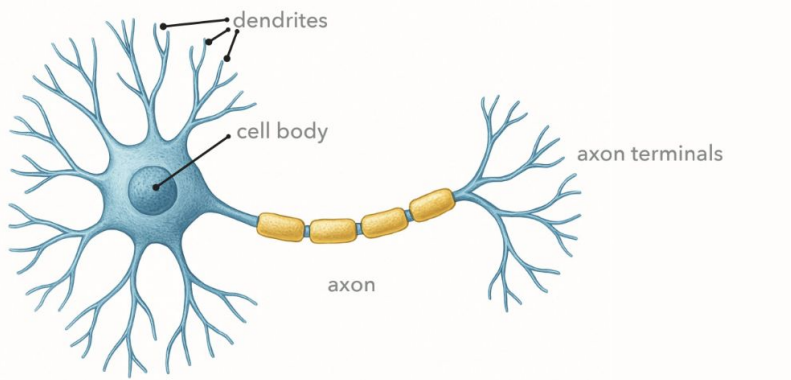
‘Is this node important? If so, how much?’

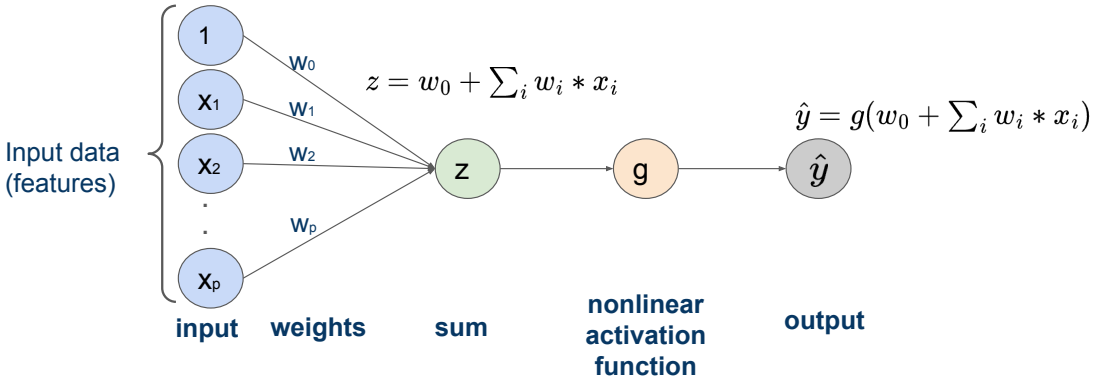


The full perceptron



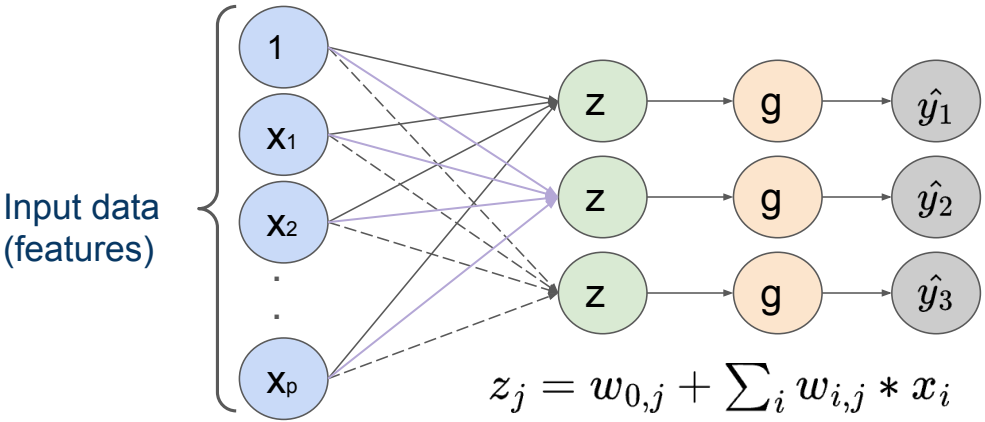
Artificial Neuron v.s. Brain Neuron





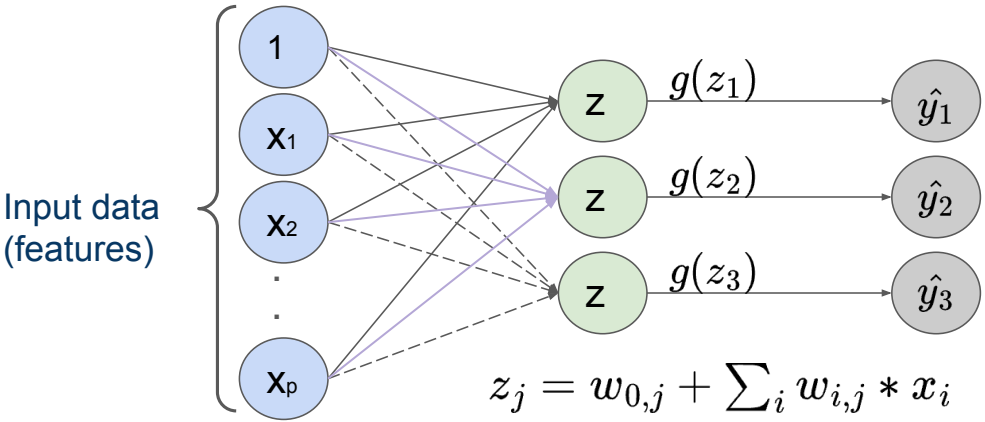
Towards a Neural Network

- All inputs are densely connected to the inputs. We call this neural network structure a '**dense layer**'.



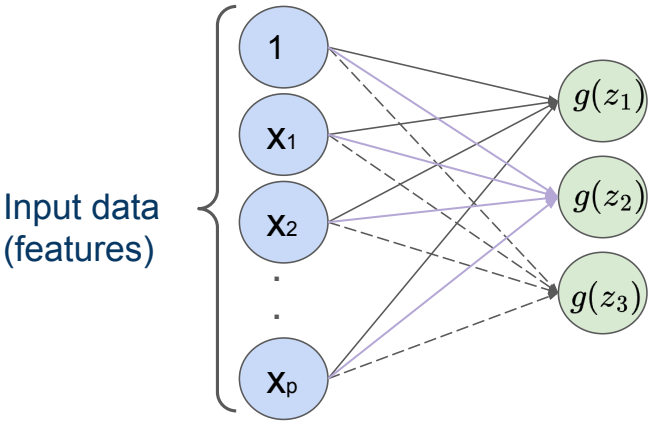
Towards a Neural Network

- All inputs are densely connected to the inputs. We call this neural network structure a '**dense layer**'.



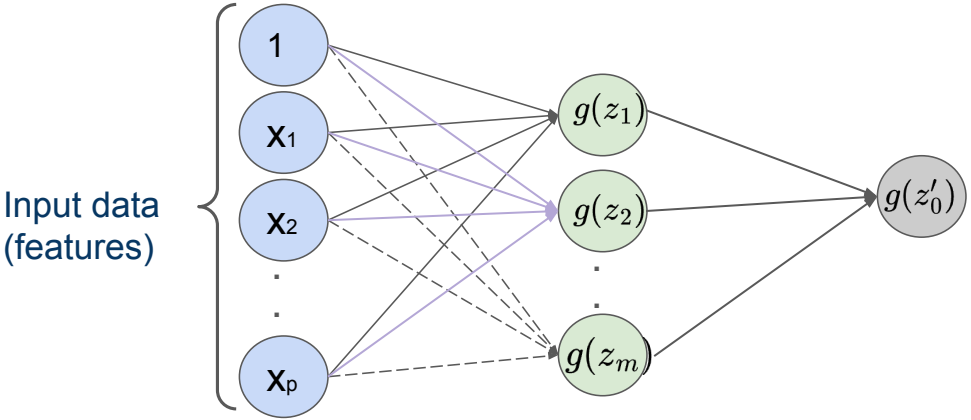
Towards a Neural Network

- All inputs are densely connected to the inputs. We call this neural network structure a '**dense layer**'.



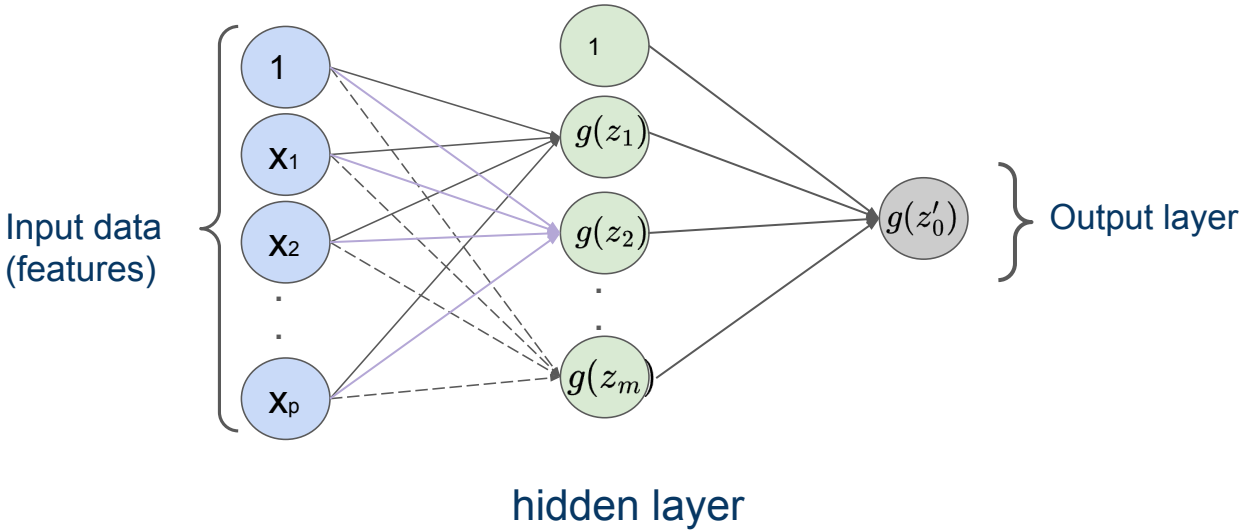
Towards a Neural Network

- Single-layer neural network



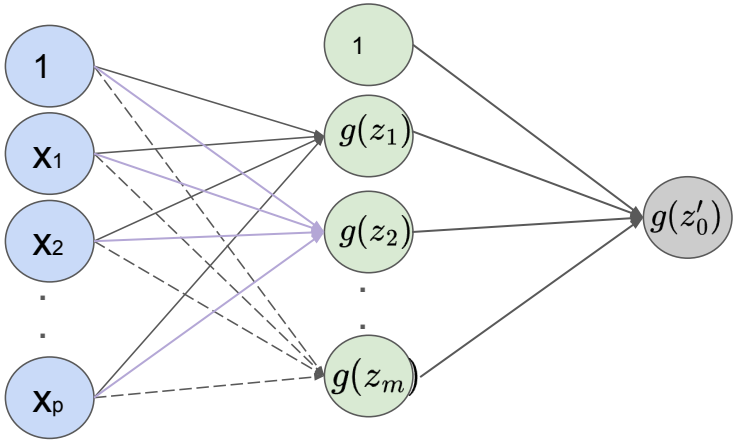
Towards a Neural Network

- Single-layer neural network



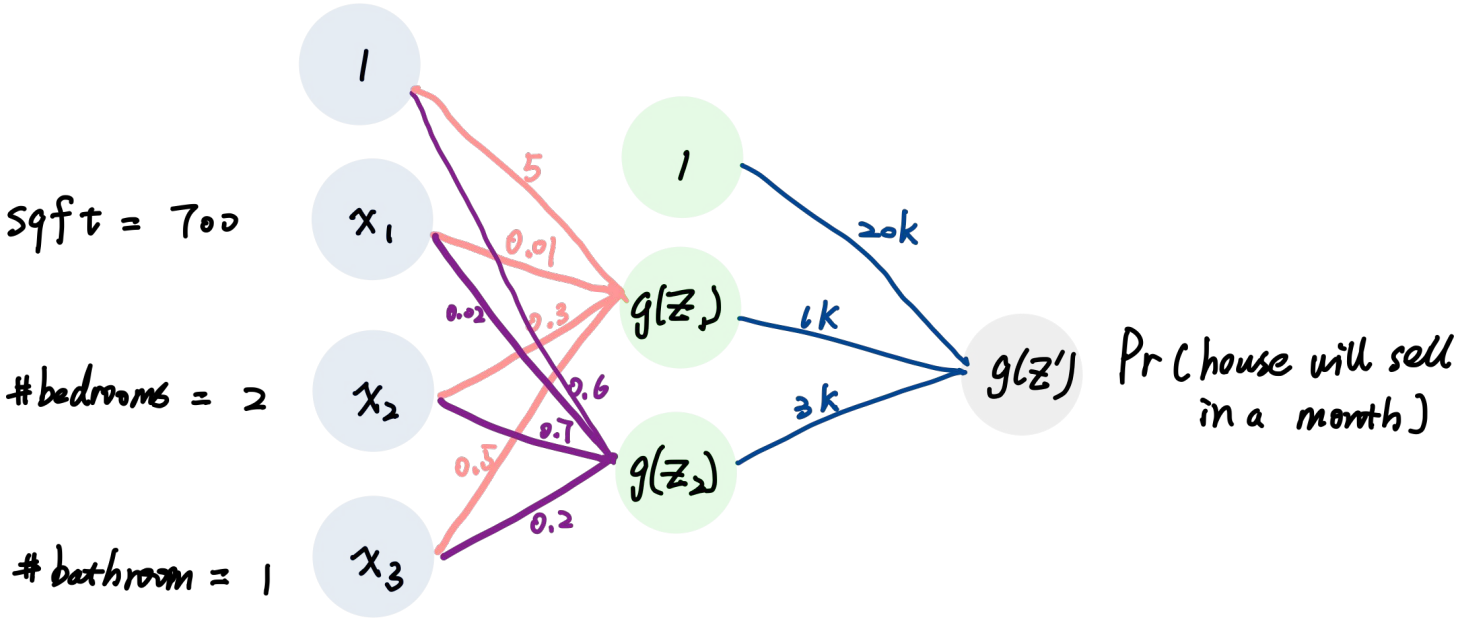
How many parameters?

- Single-layer neural network



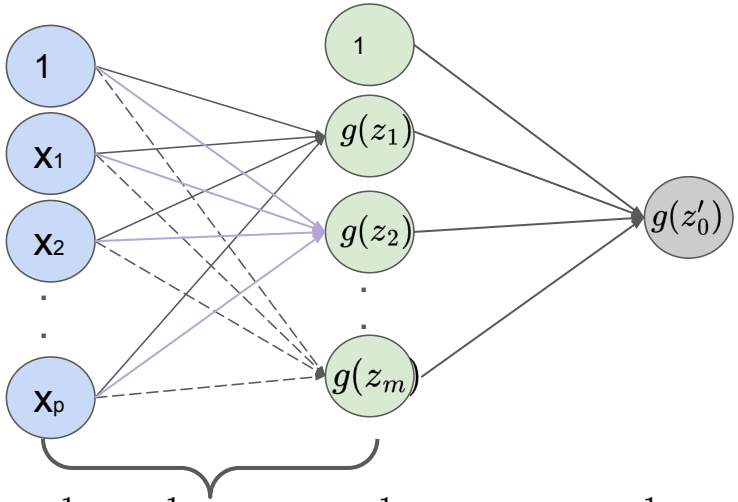
How many parameters?

- Single-layer neural network



How many parameters?

- Single-layer neural network



$$g(z_1) = w_0^1 + w_1^1 * x_1 1 + w_2^1 * x_2 + \ldots + w_p^1 * x_p$$

$$g(z_2) = w_0^2 + w_1^2 * x_1 1 + w_2^2 * x_2 + \ldots + w_p^2 * x_p$$

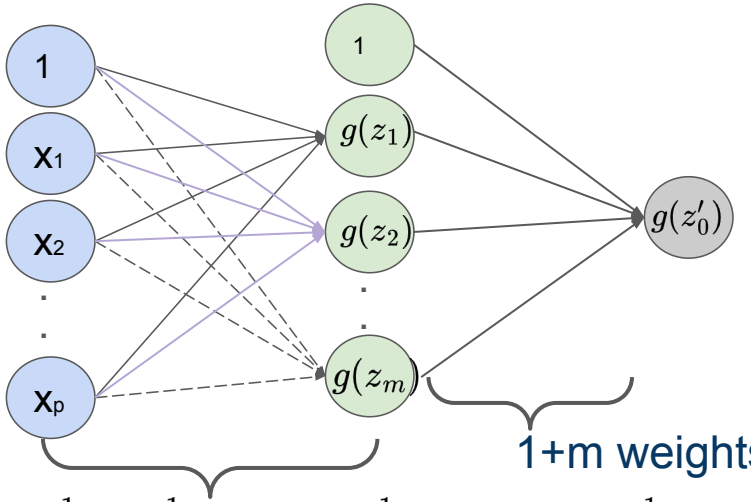
…

$$g(z_m) = w_0^m + w_1^m * x_1 1 + w_2^m * x_2 + \ldots + w_p^m * x_p$$

(1+p)*m weights

How many parameters?

- Single-layer neural network



$$g(z_1) = w_0^1 + w_1^1 * x_1 1 + w_2^1 * x_2 + \ldots + w_p^1 * x_p$$

$$g(z_2) = w_0^2 + w_1^2 * x_1 1 + w_2^2 * x_2 + \ldots + w_p^2 * x_p$$

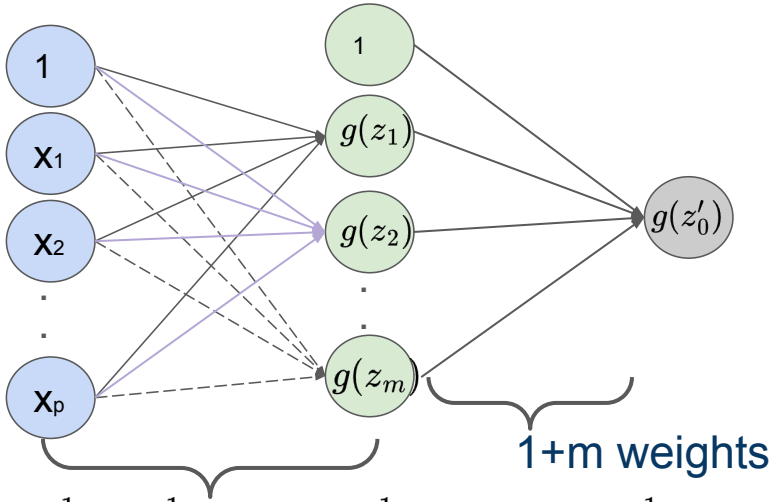
…

$$g(z_m) = w_0^m + w_1^m * x_1 1 + w_2^m * x_2 + \ldots + w_p^m * x_p$$

(1+p)*m weights

How many parameters?

- Single-layer neural network



$$g(z_1) = w_0^1 + w_1^1 * x_1 1 + w_2^1 * x_2 + \ldots + w_p^1 * x_p$$

$$g(z_2) = w_0^2 + w_1^2 * x_1 1 + w_2^2 * x_2 + \ldots + w_p^2 * x_p$$

…

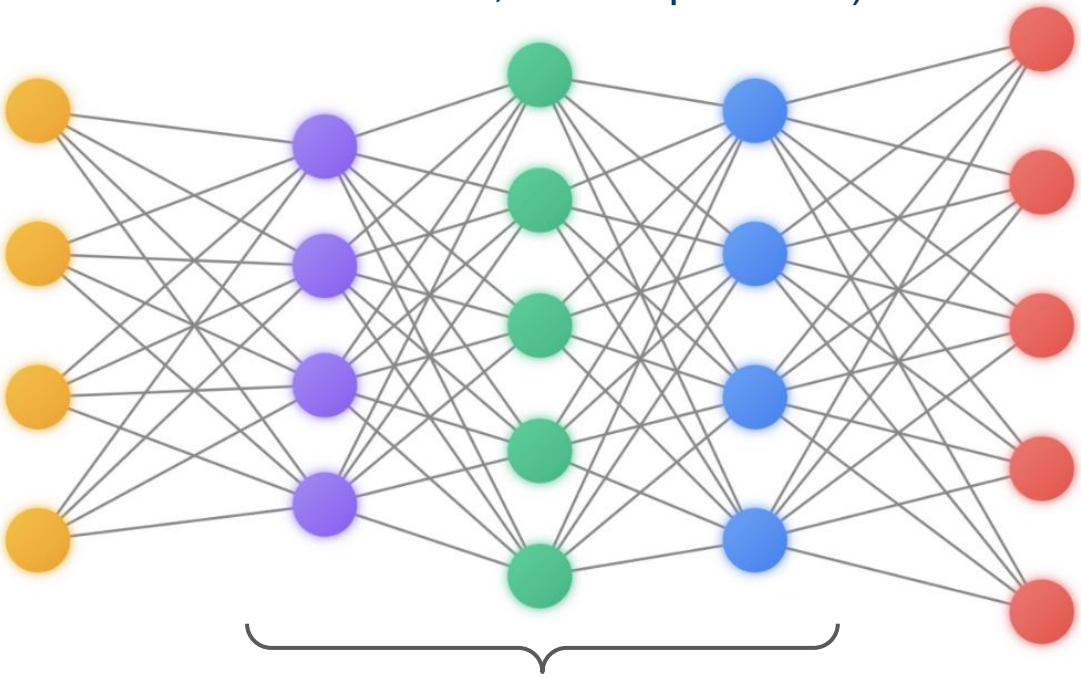
$$g(z_m) = w_0^m + w_1^m * x_1 1 + w_2^m * x_2 + \ldots + w_p^m * x_p$$

(1+p)*m weights

Total number of params:
(1+p)*m + (1+m)

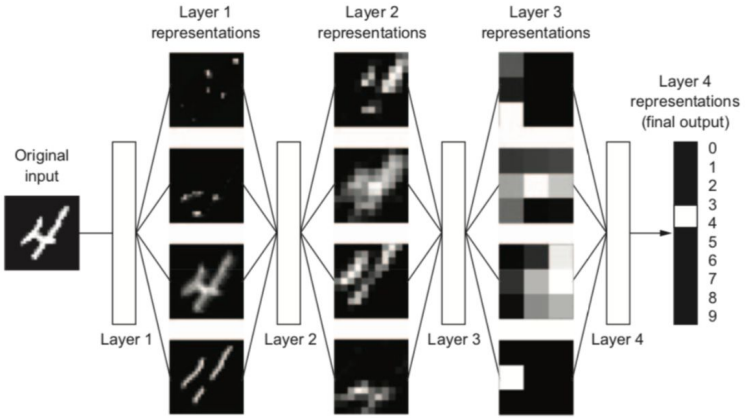
Towards a Neural Network

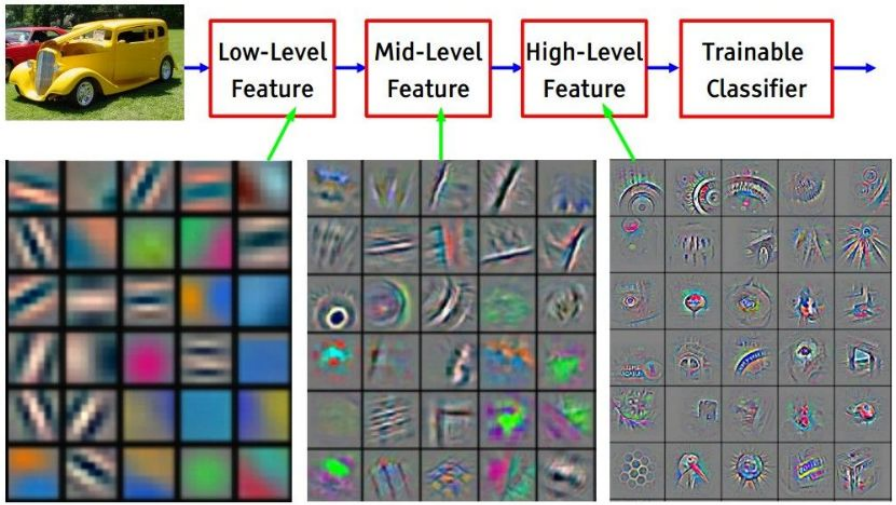
- More layers, more nodes per layer, to make the model more complex
- The output layer can have multiple nodes
  - (e.g. multi-class classification, 1 node per class)



Why the layers?

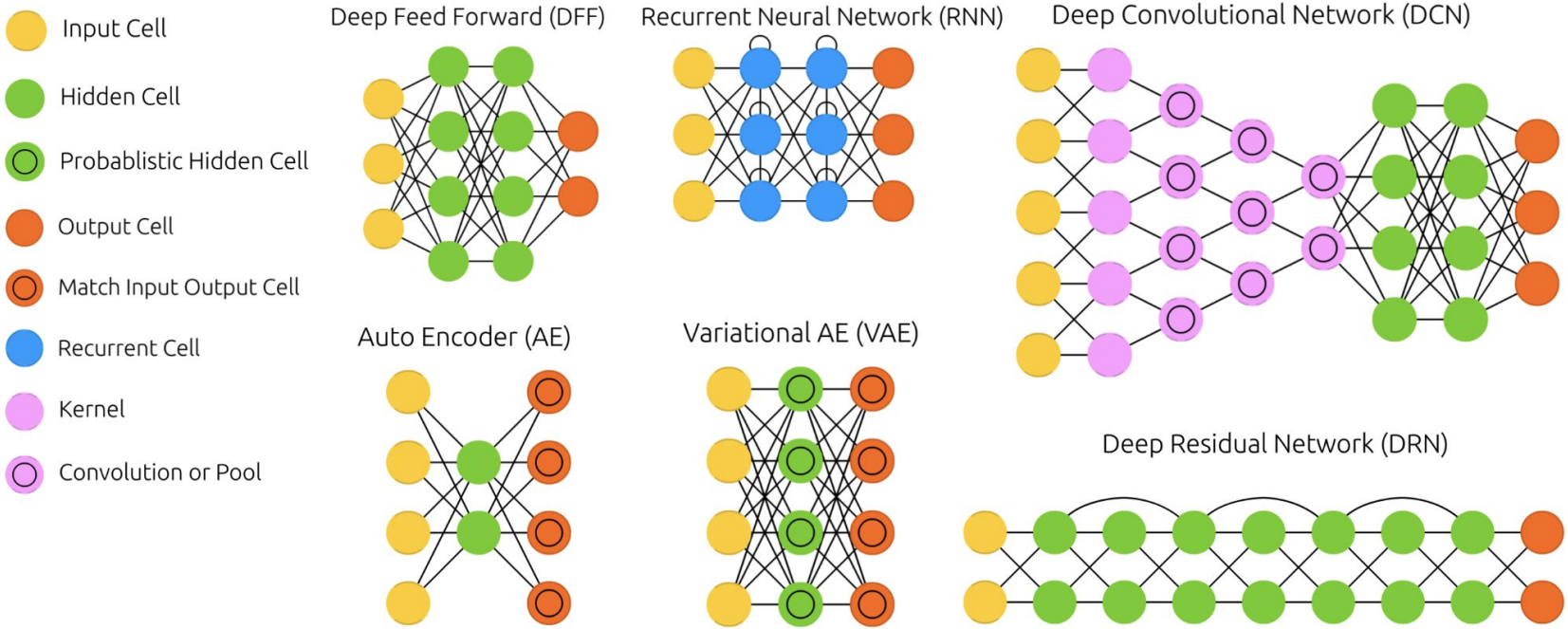
- Each layer acts like a ‘filter’ and learns something new about the data
  - Subsequent layers can learn refinements.
  - Easier than learning a complex relationship in one go.
- Example: For image input, each layer yields new (filtered) images
  - From low-level patterns (edges, endpoints,..) to combinations





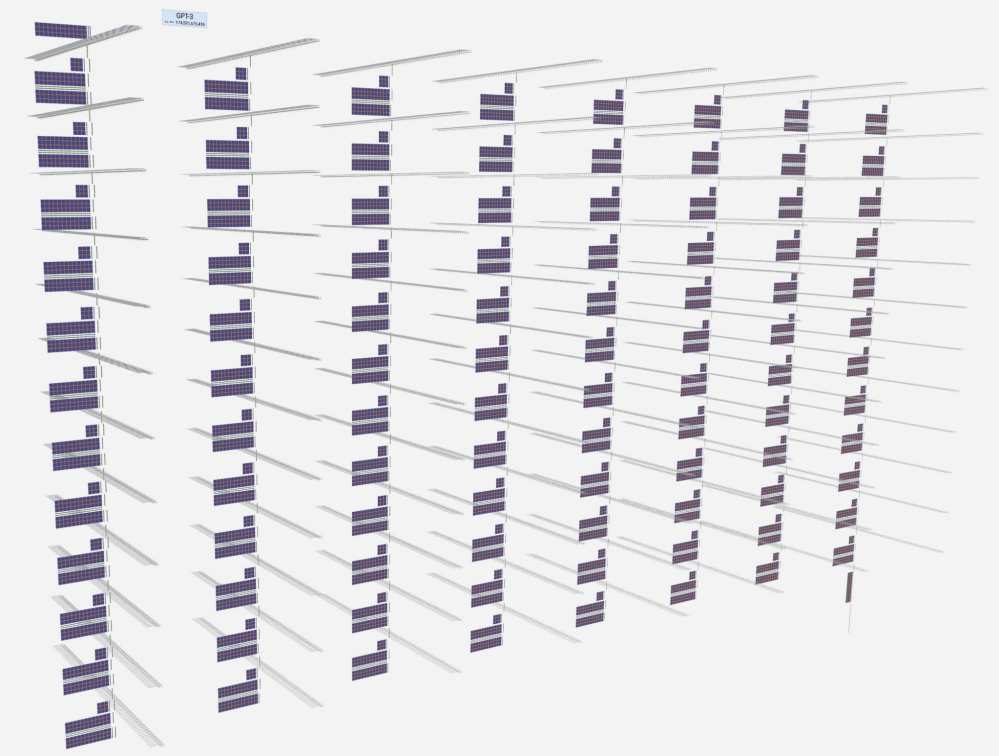
Other Architectures

- There’re many types of neural networks for different tasks.



Other Architectures

- ChatGPT is an enormous neural network called a transformer.



https://bbycroft.net/llm

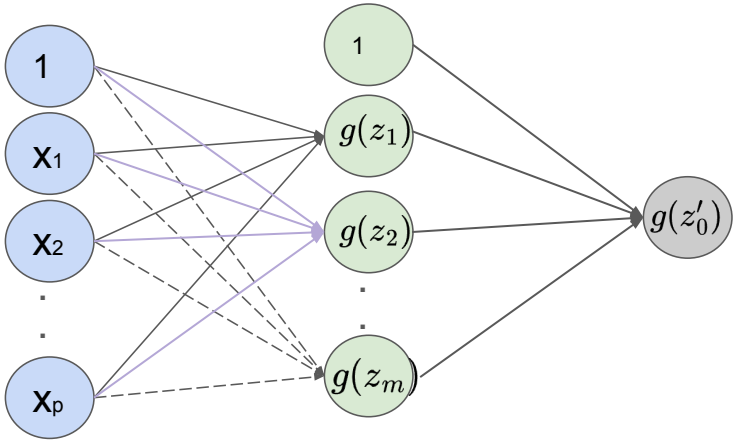
The Define a Model Step

- The input and output:
  - **Features** - Each input node is one feature variable.
    (e.g one for sqft, one for #bedrooms,..)
  - '**Target variable**' - One node for the output variable
    (e.g. housing price/if can be sold),
    Or multiple nodes
    (e.g. One node for low price, one for mid-range, one for high.)

- Design the architecture
  - Choose **the number of layers**, **the number of nodes** at each layer
  - Choose the **activation functions** (e.g. sigmoid)

How to train neural nets?

- Choose a **loss function**
  - The loss measures the error from incorrect predictions.



$$Loss(\hat{y}_i, y_i)$$

$$J(W) = \frac{1}{N} \sum_i^N Loss(\hat{y}_i, y_i)$$

Example:
Predicted: 0.4
Actual: 1

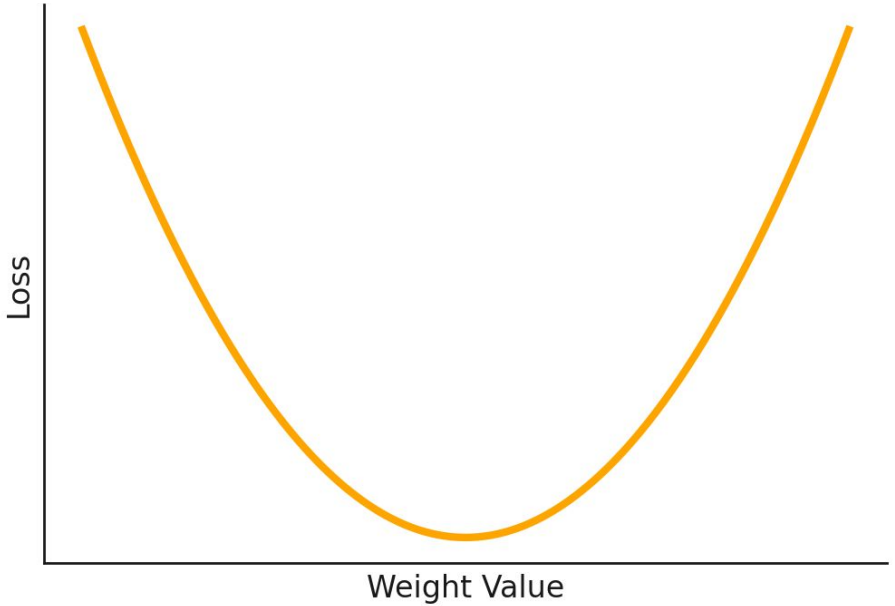
Also called objective function / cost function/ empirical risk

How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**

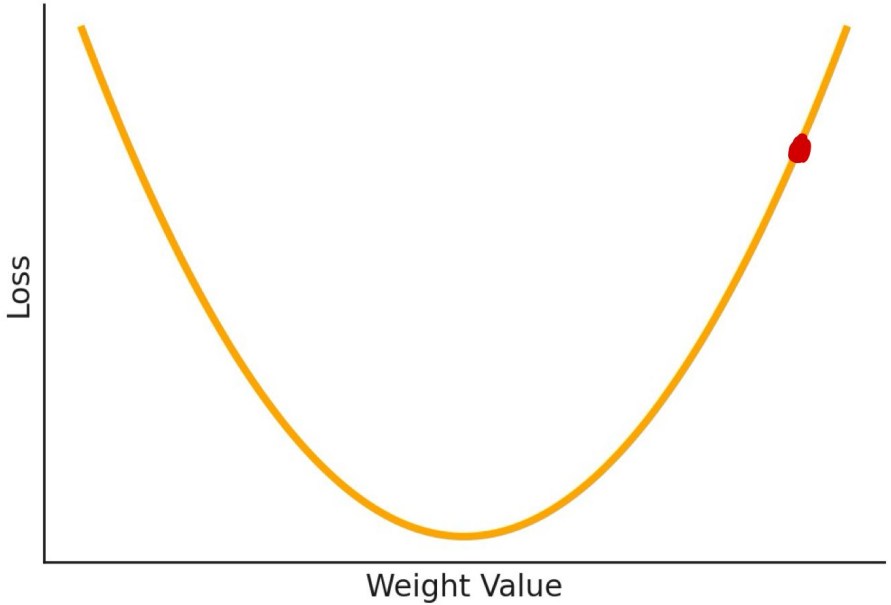
How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**



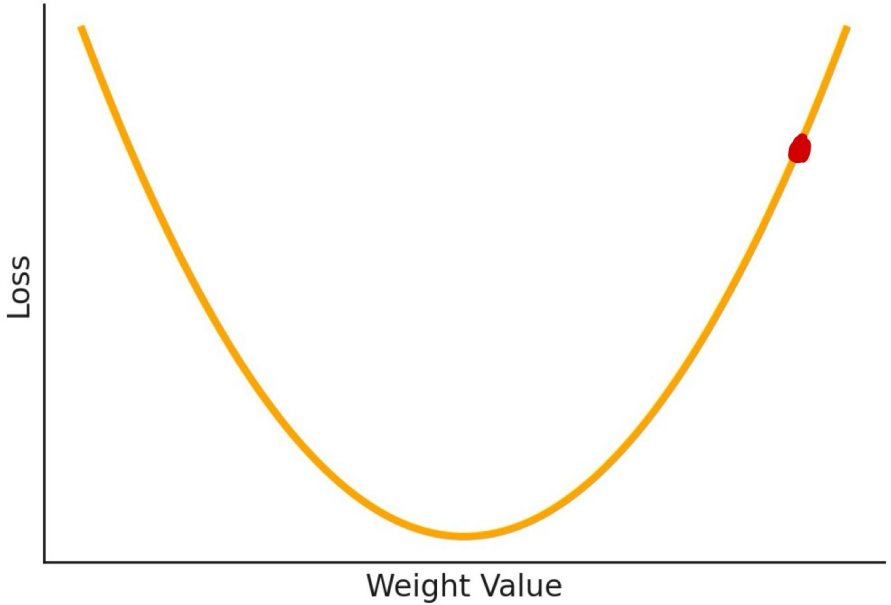
How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**
    - Start at initialization



How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**



- Start at initialization
- Which direction to move?

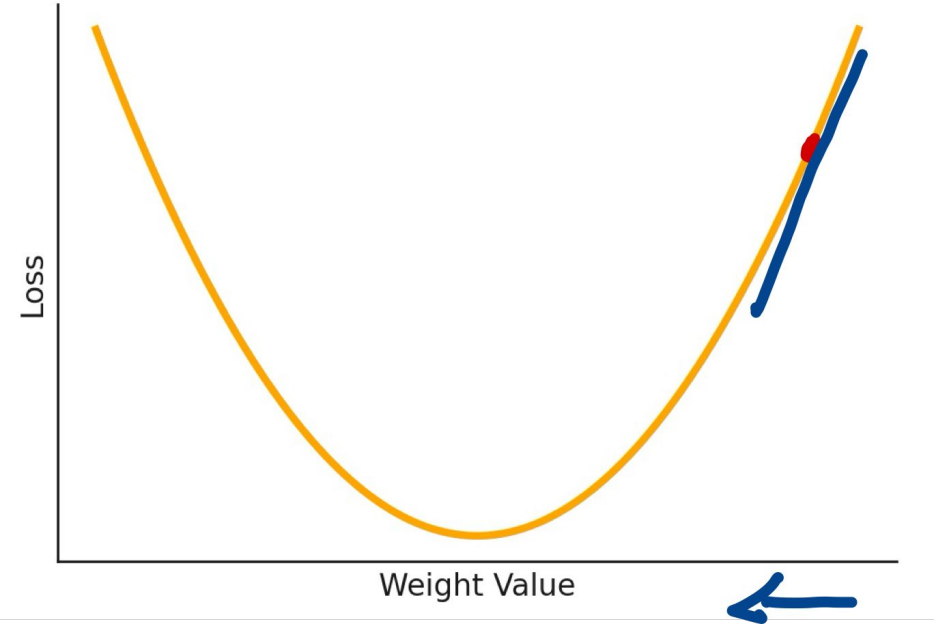
Chatgpt generated.

How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**

gradient: $\frac{\partial \text{Loss}}{\partial w}$

slope



- Start at initialization
- Which direction to move?

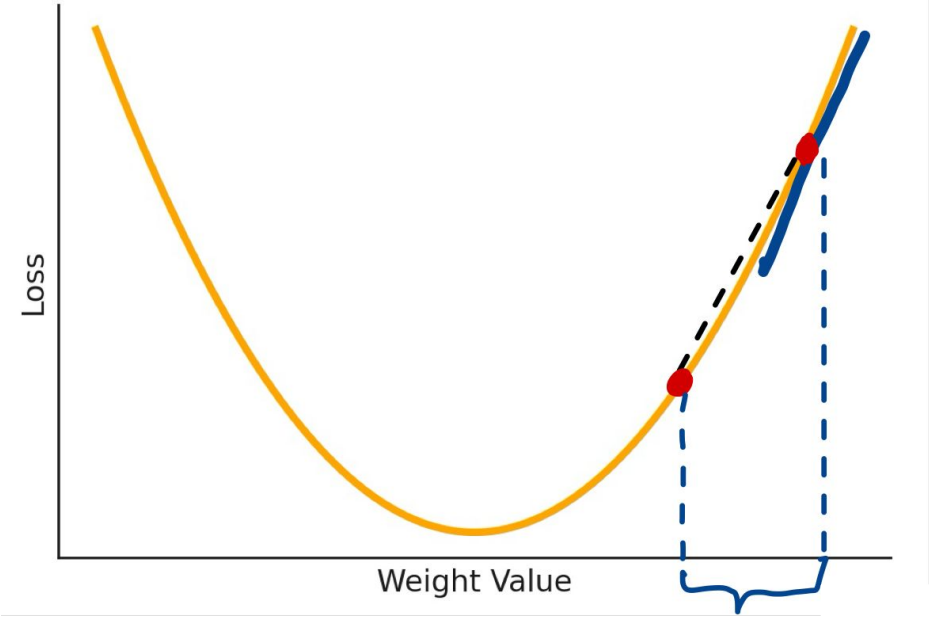
  Negative gradient, move to left

- How much to move?

How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**

gradient: $\frac{\partial \text{Loss}}{\partial w}$ (slope)



- Start at initialization
- Which direction to move?

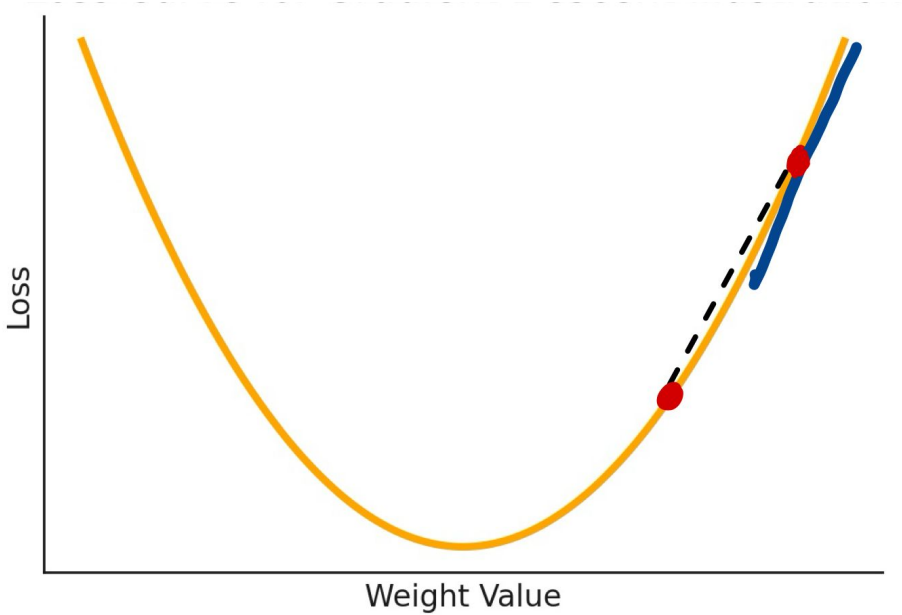
  Negative gradient, move to left

- How much to move (step size)?

$w_{t+1} = w_t + \eta \cdot \text{gradient}$

How to train neural nets?

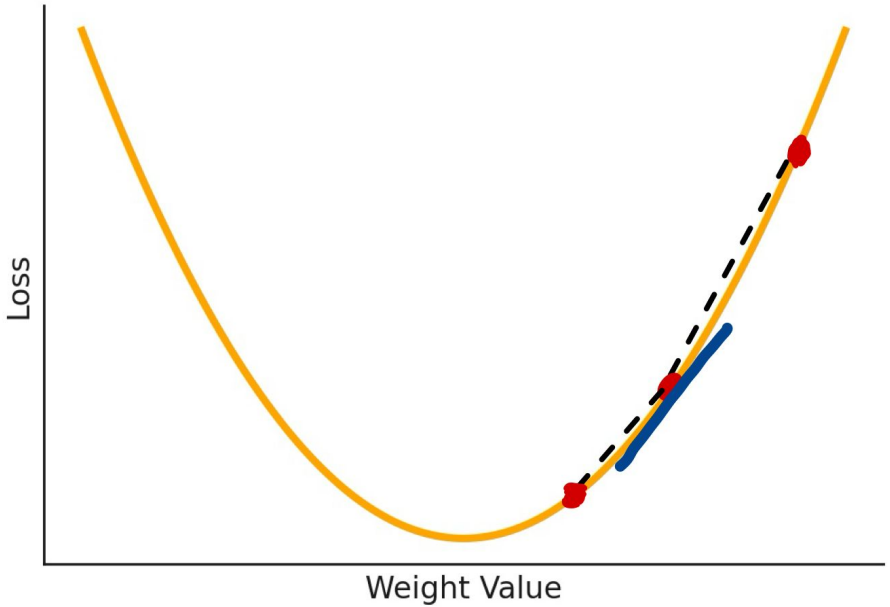
- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**



  - Iteratively adjust the weight value

How to train neural nets?

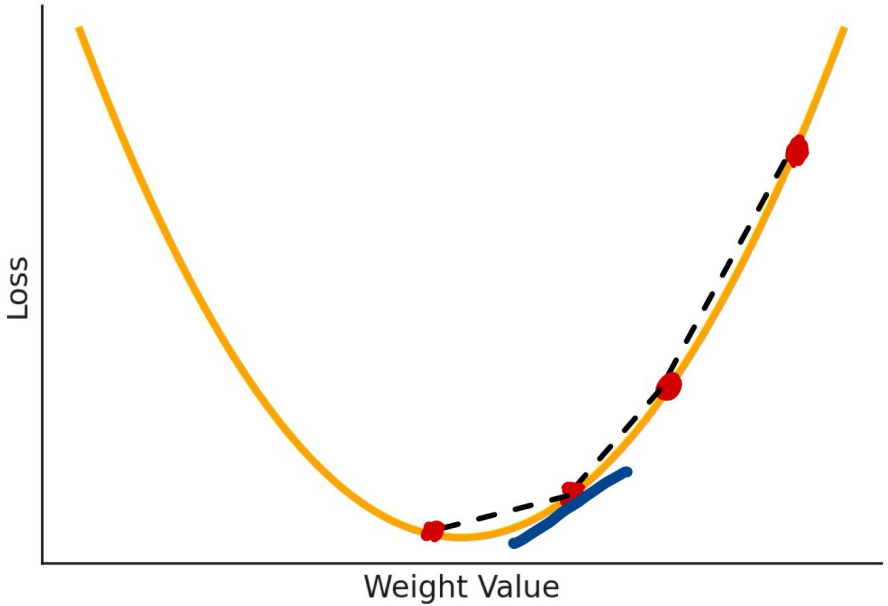
- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**



  - Iteratively adjust the weight value

How to train neural nets?

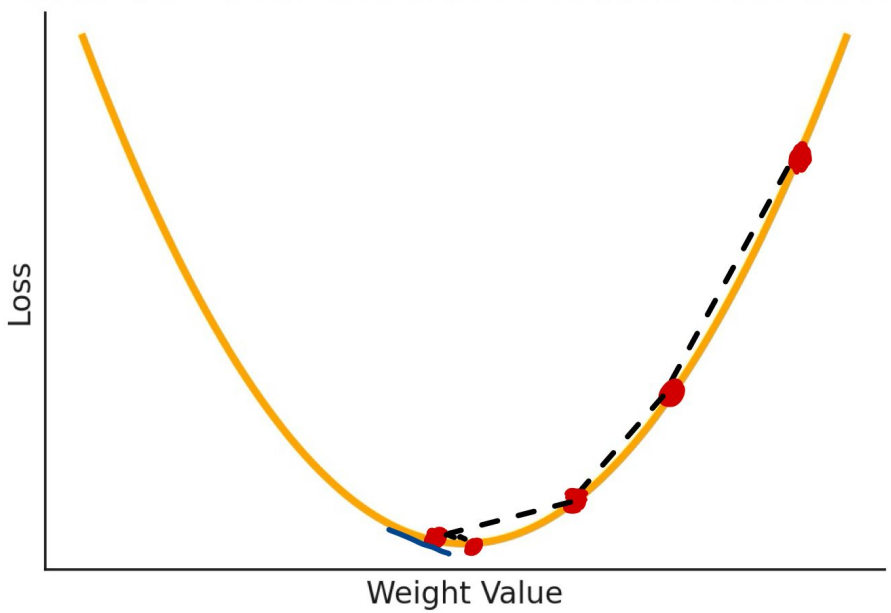
- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**



  - Iteratively adjust the weight value

How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent**



  - Iteratively adjust the weight value till convergence

Converge: The loss doesn't decrease much when moving around parameters.

How to train neural nets?

- **Find the weights** that **minimize the loss function**
  - Optimizer: **Gradient descent -** Choosing learning rate [interactive]
  - Advanced: & adaptive learning rate
    E.g. Adam optimizer is extremely popular!

Learning rate too large?

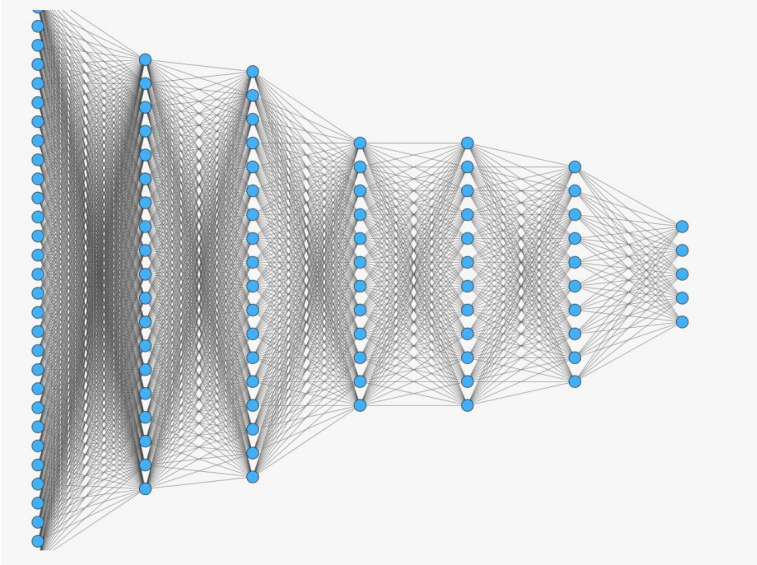
Zigzag. Never converges?

Learning rate too small?

Painfully slow training

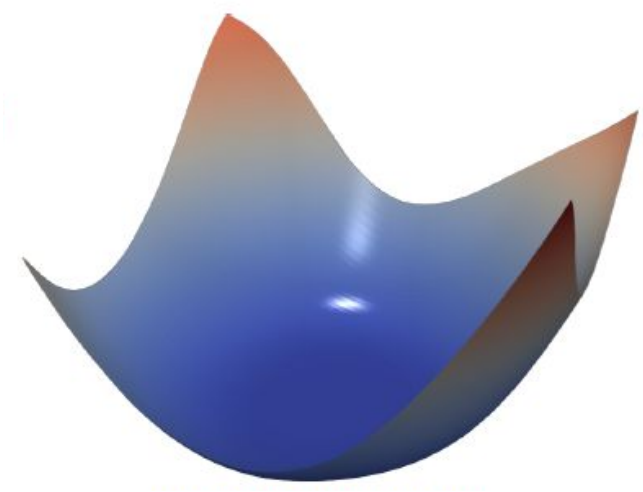
How to train neural nets?

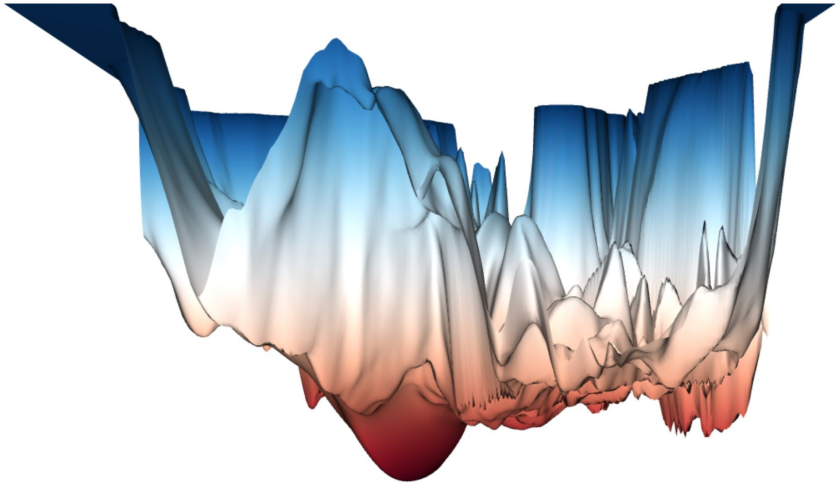
- **Finding the weights** that **minimize the loss function** seems challenging.
  - A huge number of weights!



How to train neural nets?

- **Finding the weights** that **minimize the loss function** seems challenging.
  - A huge number of weights!
  - The loss landscape is not smooth at all!



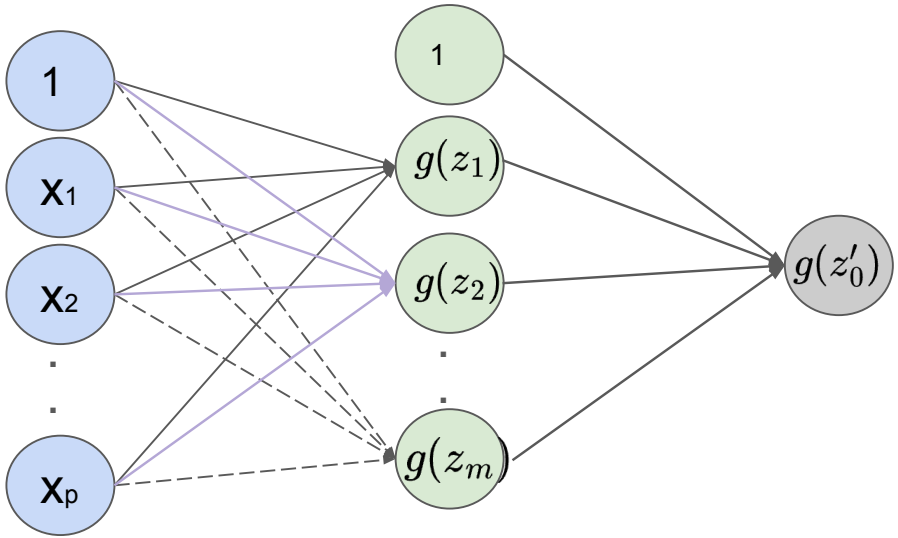


https://www.telesens.co/loss-landscape-viz/viewer.html

How to train a neural network

- Optimize the values of the weights to minimize the loss
  - Initialize the weights

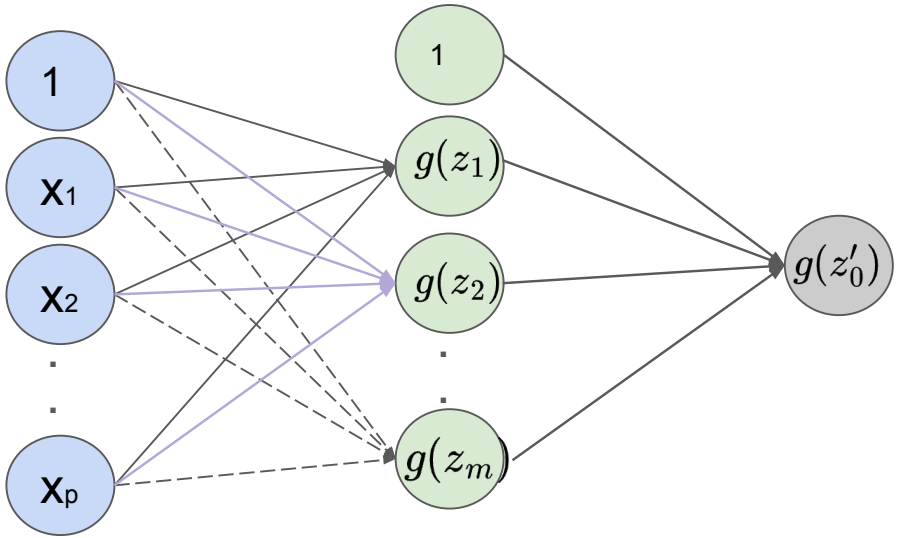
Don't start with all 0s, but random numbers.



$$Loss(\hat{y}_i, y_i)$$
$$J(W) = \frac{1}{N} \sum_i^N Loss(\hat{y}_i, y_i)$$

How to train a neural network

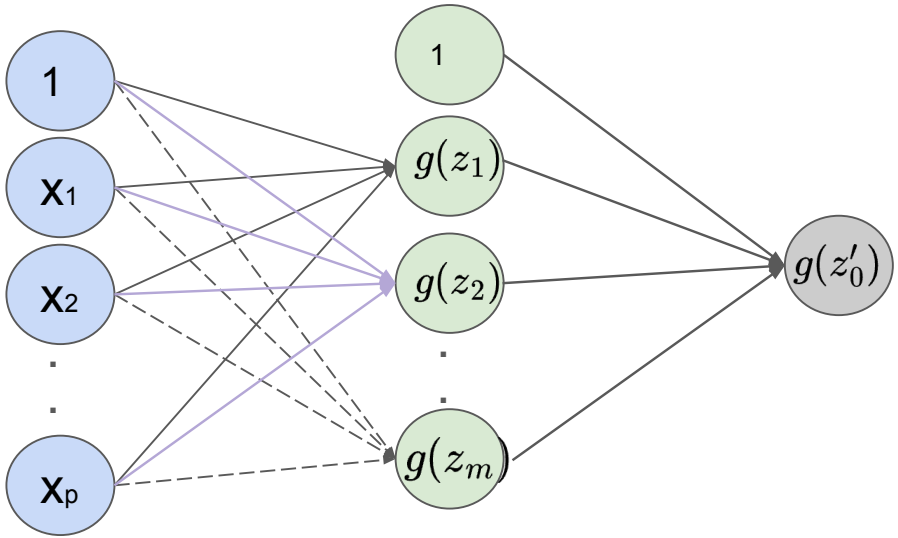
- Optimize the values of the weights to minimize the loss
  - Initialize the weights
  - Back propagation: loop until convergence



$$Loss(\hat{y}_i, y_i)$$
$$J(W) = \frac{1}{N} \sum_i^N Loss(\hat{y}_i, y_i)$$

How to train a neural network

- Back Propagation

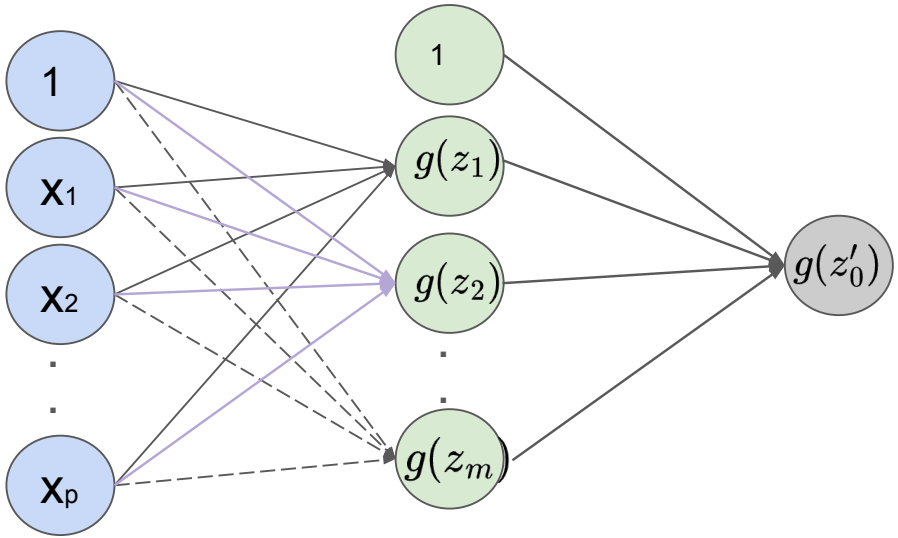


$$Loss(\hat{y}_i, y_i)$$
$$J(W) = \frac{1}{N} \sum_i^N Loss(\hat{y}_i, y_i)$$

How to train a neural network

- Back Propagation
  - Step 1: Forward pass

    Just let data go through the NN, calculate the predictions and the loss.

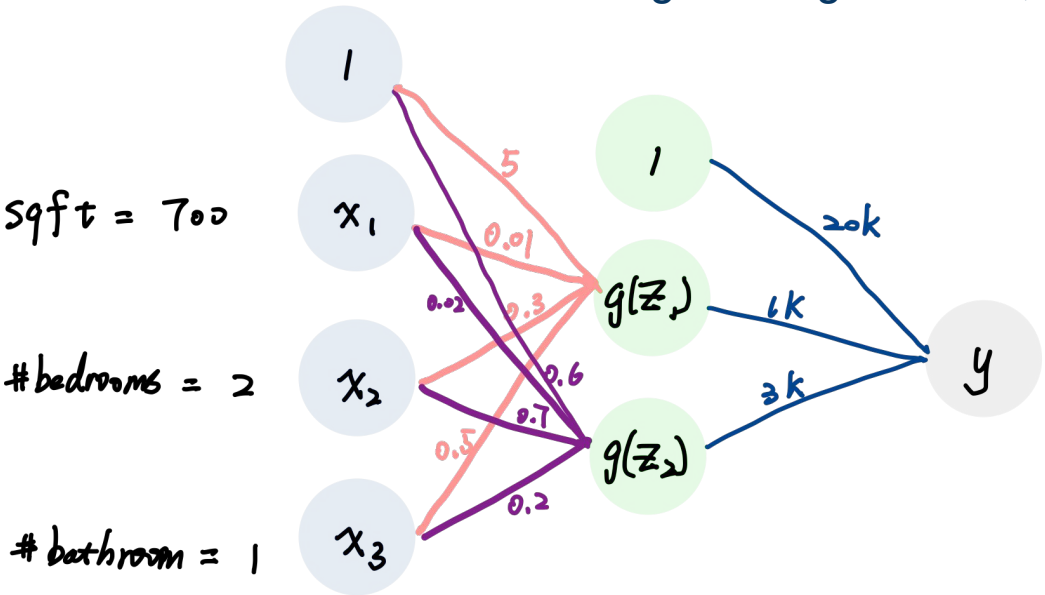


$$Loss(\hat{y}_i, y_i)$$
$$J(W) = \frac{1}{N} \sum_i^N Loss(\hat{y}_i, y_i)$$

How to train a neural network

- Back Propagation
  - Step 1: Forward pass

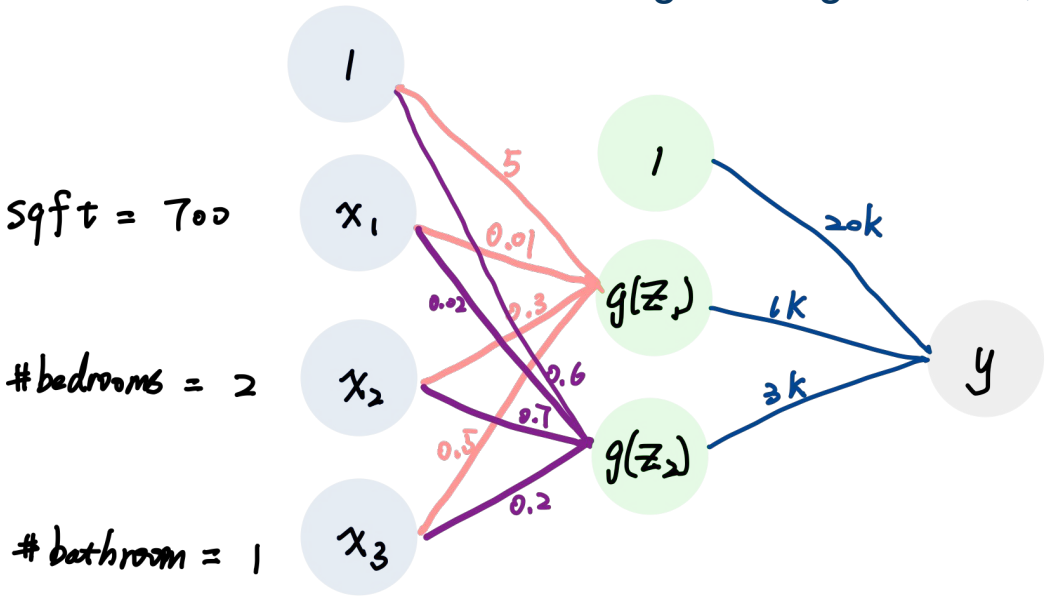
    Just let data go through the NN, calculate the predictions and the loss.



How to train a neural network

- Back Propagation
  - Step 1: Forward pass

    Just let data go through the NN, calculate the predictions and the loss.



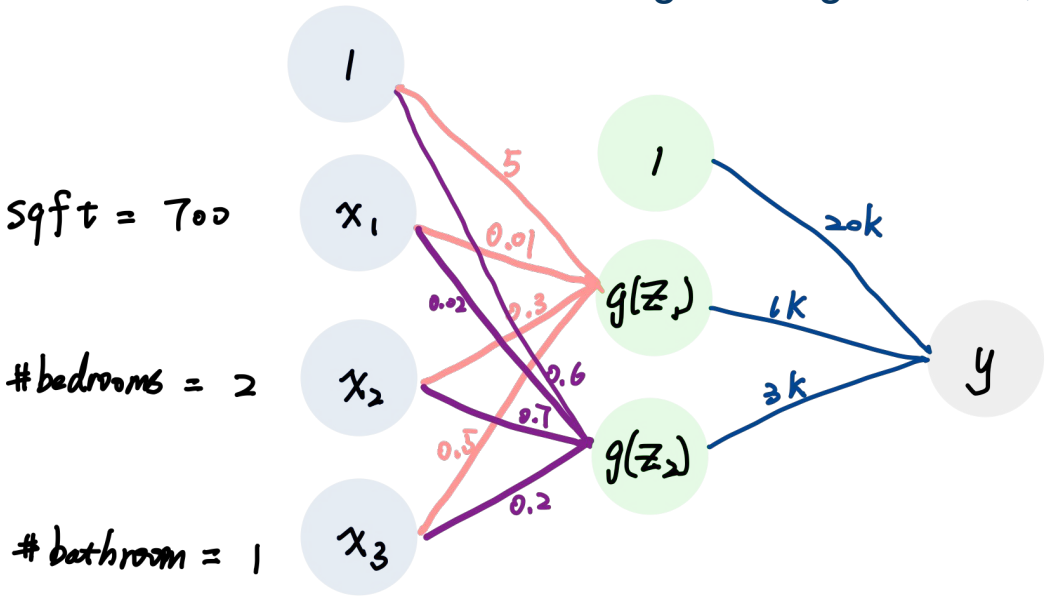
$\hat{z}_1 = 5 \times 1 + 0.01 \times 700 + 0.3 \times 2 + 0.5 \times 1$

$\hat{z}_2 = 0.6 \times 1 + 0.02 \times 700 + 0.7 \times 2 + 0.2 \times 1$

How to train a neural network

- Back Propagation
  - Step 1: Forward pass

    Just let data go through the NN, calculate the predictions and the loss.



$\hat{z}_1 = 5 \times 1 + 0.01 \times 700 + 0.3 \times 2 + 0.5 \times 1$

$\hat{z}_2 = 0.6 \times 1 + 0.02 \times 700 + 0.7 \times 2 + 0.2 \times 1$

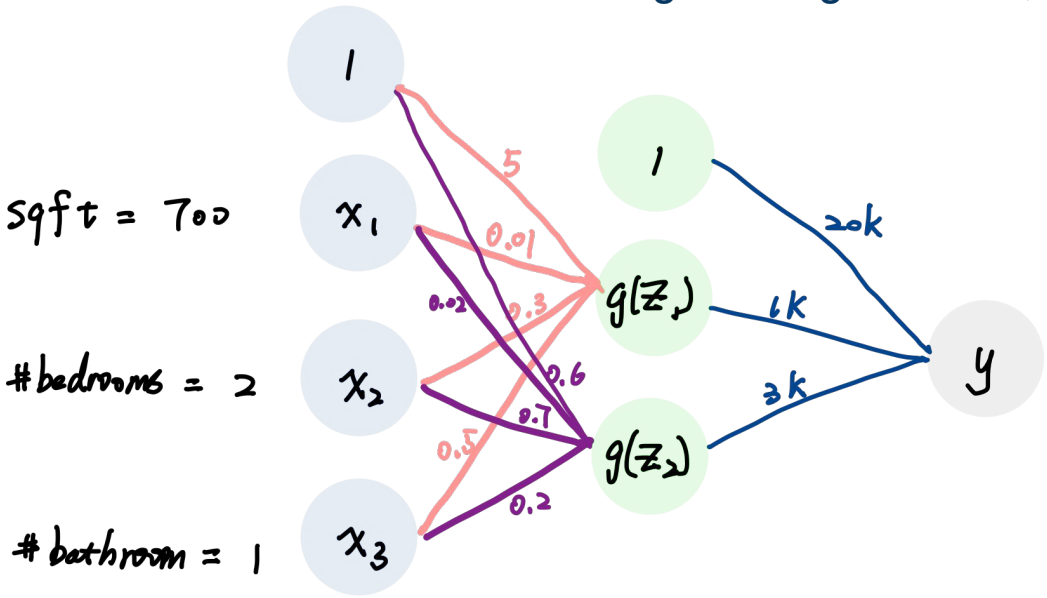
$\hat{g}(z_1) = \text{sigmoid}(z_1)$

$\hat{g}(z_2) = \text{sigmoid}(z_2)$

How to train a neural network

- Back Propagation
  - Step 1: Forward pass

    Just let data go through the NN, calculate the predictions and the loss.



$$\hat{z}_1 = 5\times 1 + 0.01\times 700 + 0.3\times 2 + 0.5\times 1$$

$$\hat{z}_2 = 0.6\times 1 + 0.02\times 700 + 0.7\times 2 + 0.2\times 1$$

$$\hat{g}(z_1) = \text{sigmoid}(z_1)$$

$$\hat{g}(z_2) = \text{sigmoid}(z_2)$$

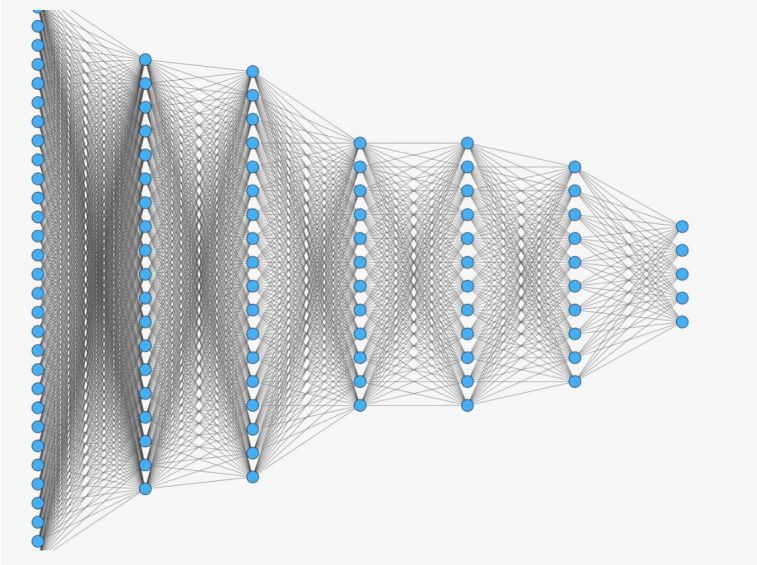
$$\hat{y} = \text{sigmoid}(20k\times 1 + 1k\times \hat{g}(\hat{z}_1) + 3k\times \hat{g}(z_2))$$

Recap

- Define the model
  - Basic unit: Perceptron
    - The weights: how important the node(variable) is.
    - The non-linear activation function: add non-linearity
  - Architecture of neural networks
    - Stacking layers of perceptron
- Training:
  - Minimize the loss function via gradient descent.

How to train neural nets?

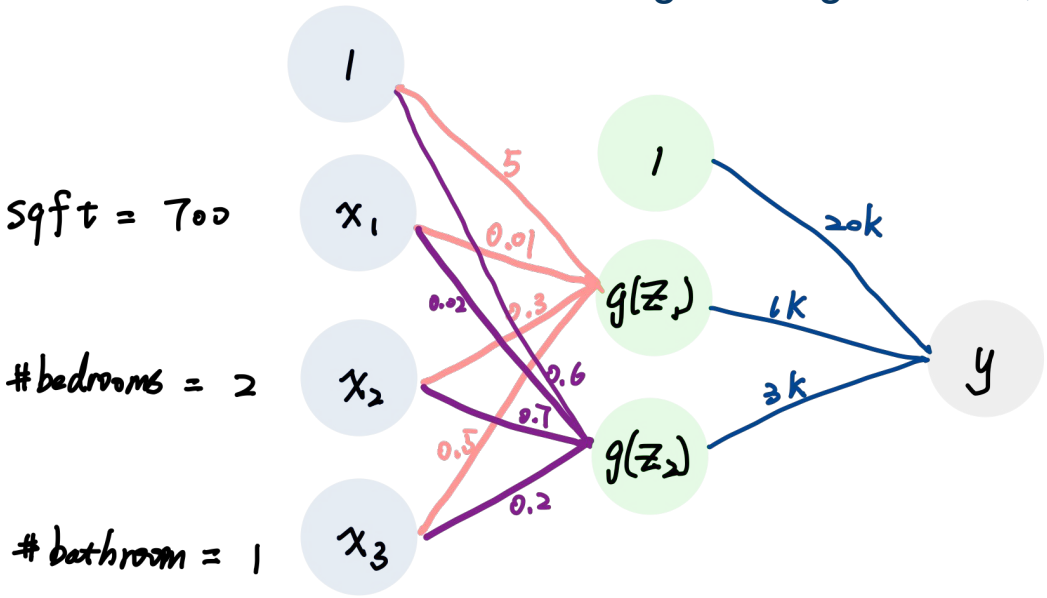
- **Finding the weights** that **minimize the loss function** seems challenging.
  - A huge number of weights!



How to train a neural network

- Back Propagation
  - Step 1: Forward pass

    Just let data go through the NN, calculate the predictions and the loss.



$$\hat{z}_1 = 5 \times 1 + 0.01 \times 700 + 0.3 \times 2 + 0.5 \times 1$$

$$\hat{z}_2 = 0.6 \times 1 + 0.02 \times 700 + 0.7 \times 2 + 0.2 \times 1$$

$$\hat{g}(z_1) = \text{sigmoid}(z_1)$$

$$\hat{g}(z_2) = \text{sigmoid}(z_2)$$

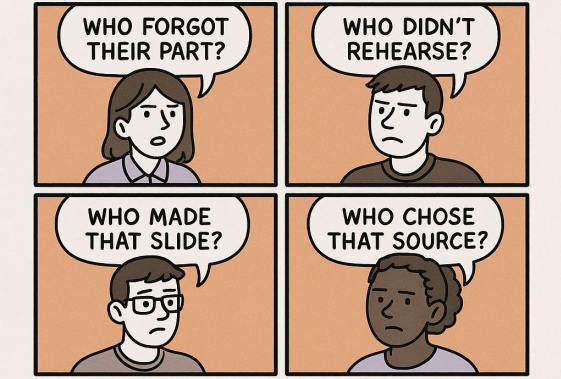
$$\hat{y} = \text{sigmoid}(20k \times 1 + 1k \times \hat{g}(\hat{z}_1) + 3k \times \hat{g}(z_2))$$

$$\text{loss} = \frac{1}{N} \sum_i^N (\hat{y} - y)^2$$

How to train a neural network

- Back Propagation
  - Step 1: Forward pass
  - Step 2: Backward pass

How much does each weight contribute to the error? Then adjust the weights accordingly.

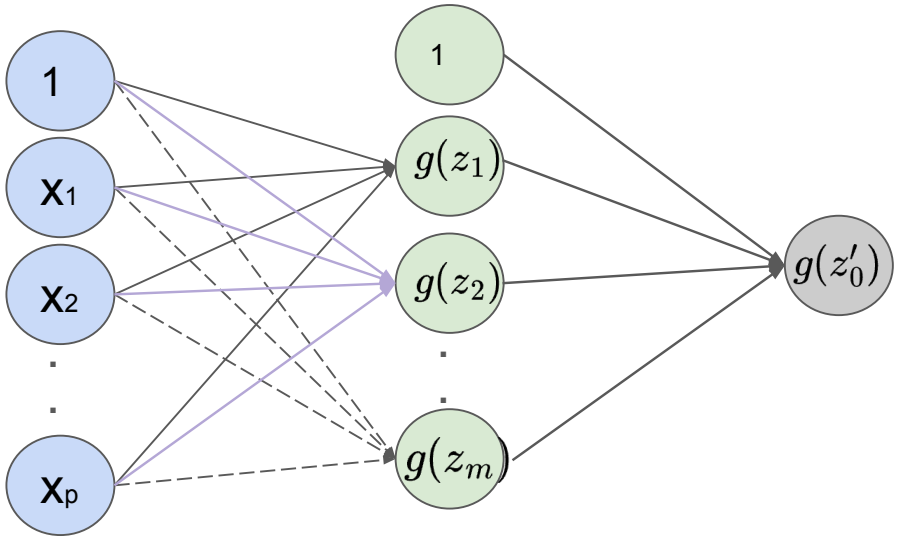


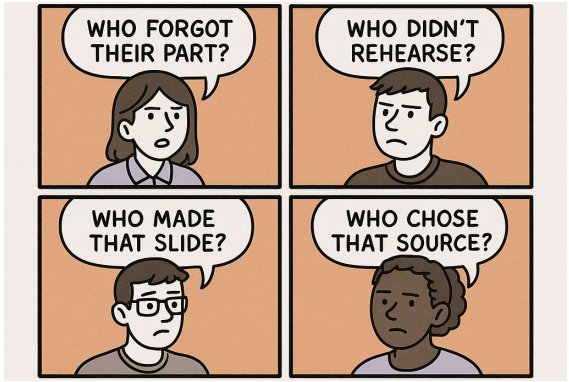
Chatgpt generated.

How to train a neural network

- Back Propagation
  - Step 1: Forward pass
  - Step 2: Backward pass
    - Compute gradient using the **chain rule**

How much does each weight contribute to the error? Then adjust the weights accordingly.

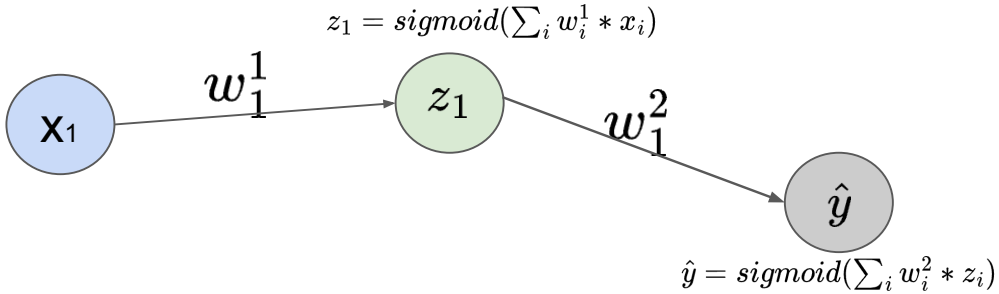




Chatgpt generated.

How to train a neural network

- Back Propagation
  - Step 1: Forward pass
  - Step 2: Backward pass
    - Compute gradient using the **chain rule**



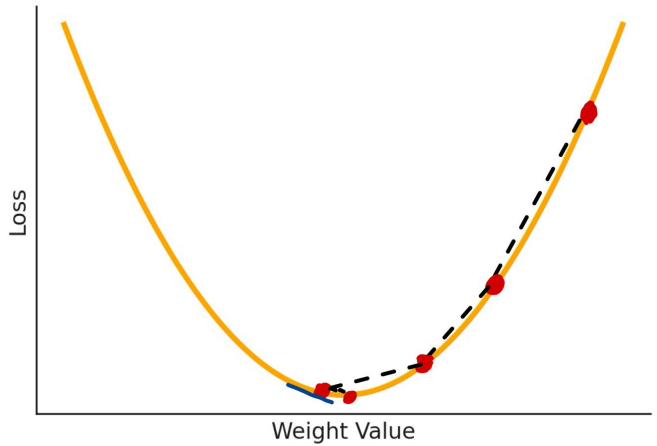
$$Loss(\hat{y}_i, y_i)$$
$$J(W) = \frac{1}{N} \sum_i^N Loss(\hat{y}_i, y_i)$$

How does a little change in w_1^1 affect the loss function?

$$\frac{\partial Loss}{\partial w_1^1} = \frac{\partial Loss}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1^1}$$

How to train a neural network

- Back Propagation
  - Step 1: Forward pass
  - Step 2: Backward pass
    - Compute gradient using the **chain rule**
    - Update weights using **gradient descent**



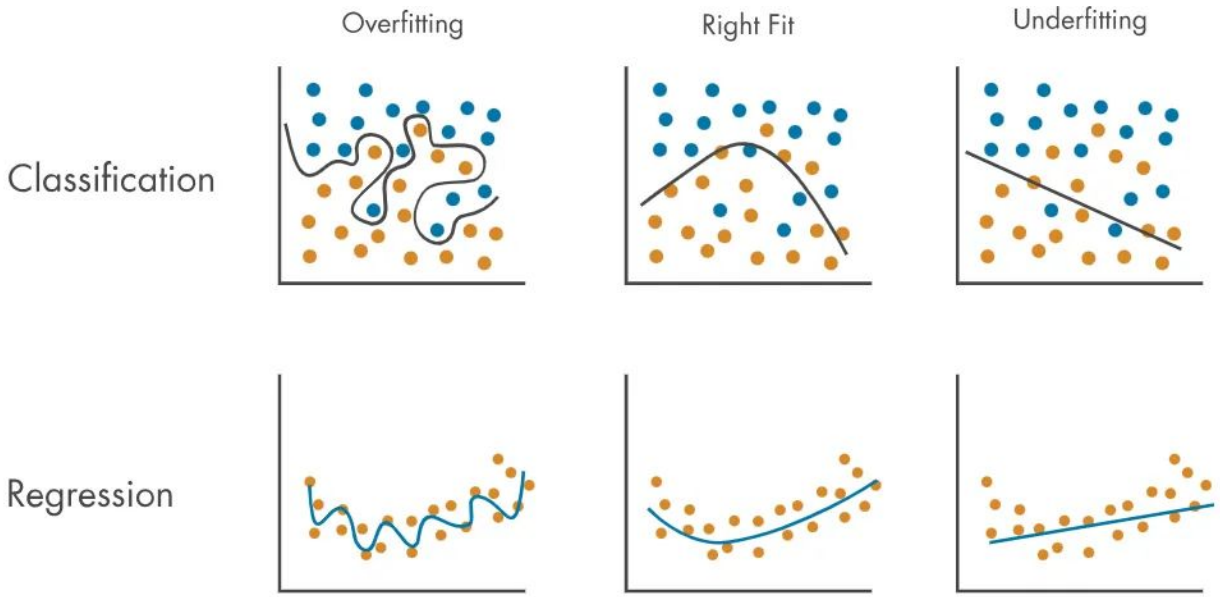
New weight = old weight – learning rate × gradient.

The Full Training

Training is iterative. A loop is also called an epoch, an interaction. All the same thing.

- Initialize the weights (at random)
- Back preparation - loop until convergence
  - Step 1: Forward pass: send data through, calculate prediction & loss
  - Step 2: Backward pass
    - Use the chain rule to calculate the gradients for nodes
    - Update weights using gradient descent
- Return the weights.

Overfitting



Neural networks: Too complex n powerful. It might memorizes the training data.

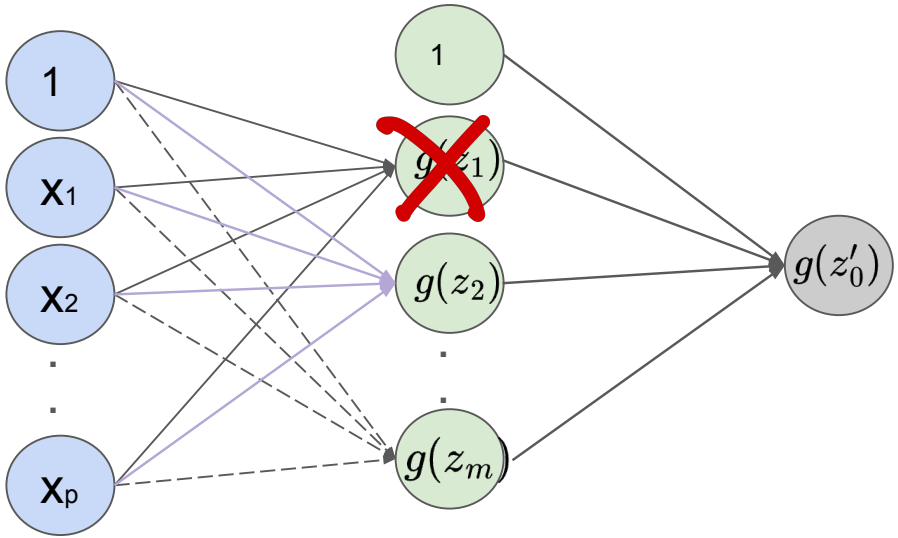
- We need **regularization techniques** to avoid over-complexity, so the model could be generalized to new situations.

Overfitting

- Too complex neural networks:
  - Too many hidden layers.
  - Too many nodes each layer

Regularization idea 1: Drop out

Randomly dropping some nodes **during training** (i.e., setting to 0) to force the network not to rely on just a few nodes.

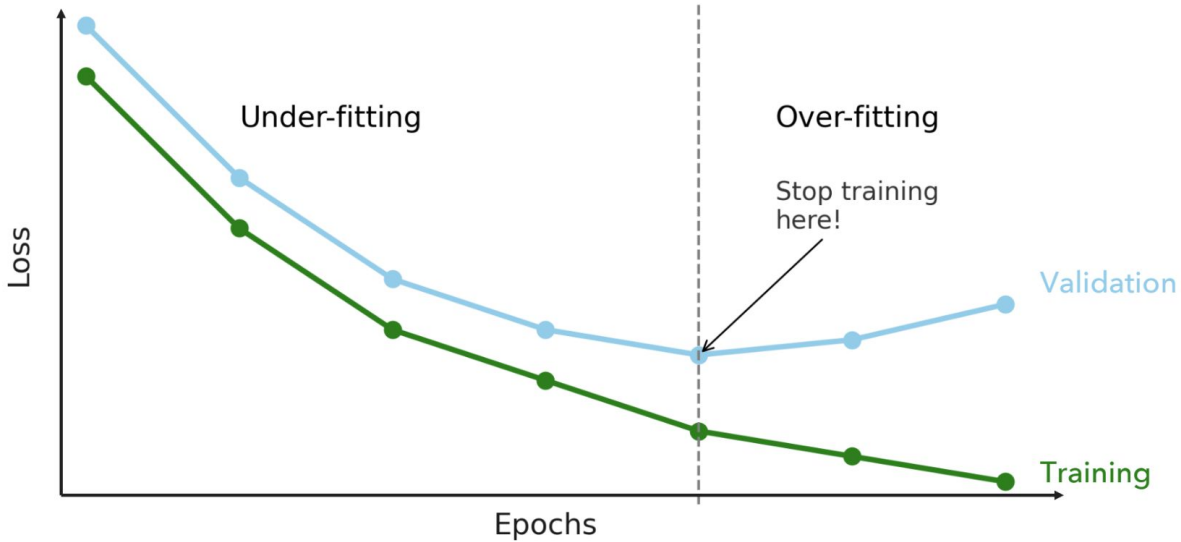


Overfitting

- The neural network is ‘too well trained’.

Regularization idea 2: Early Stopping

Stop the training early.



Summary

- Define the model
  - Basic unit: Perceptron
    - The weights: how important the node(variable) is.
    - The non-linear activation function.
  - Architecture of neural networks
    - Stacking layers of perceptron
- Training:
  - Weight initialization
  - Back propagation - loops till convergence
    - Forward pass
    - Backward pass
      - Apply the chain rule to calculate the gradients
      - Update weights using gradient descent
- Evaluation
  - Regularization techniques to avoid overfitting
    - Dropout/ Early stopping