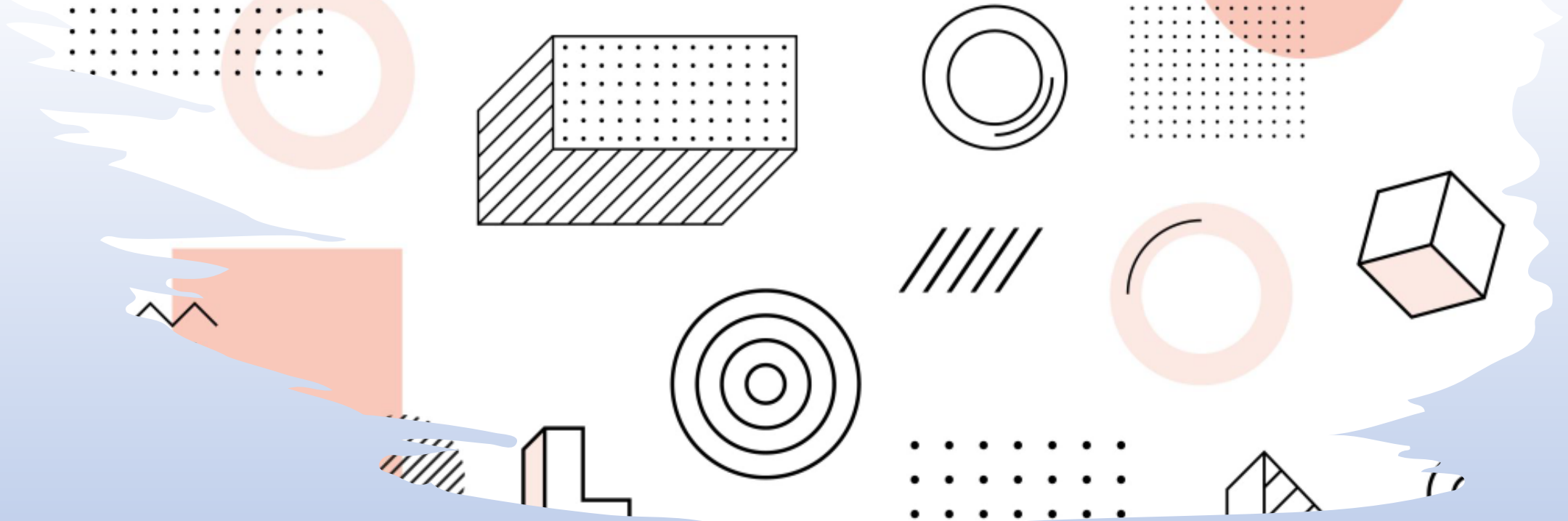


Introduction to Deep Learning

UNIT1

CONTENT

- Basics of Artificial Neural Networks (ANN)
- Activation functions (ReLU, Sigmoid, Tanh)
- Forward propagation
- Backpropagation
- Gradient descent Optimization techniques
  - SGD
  - Adam
  - RMSprop
- Introduction to deep learning frameworks (TensorFlow, PyTorch).

Course Objectives

Understand	the theoretical foundations and principles of deep learning algorithms and their applications.
Explore	different architectures of neural networks such as CNNs, RNNs, and GANs.
Apply	deep learning techniques to solve real-world problems such as image recognition, natural language processing, and speech processing.
Gain	hands-on experience in implementing deep learning models using popular frameworks like TensorFlow and PyTorch.
Analyse and optimize	deep learning models for better performance and efficiency.

Deep Learning | What is Deep Learning? | Deep Learning Tutorial For Beginners | 2023 | Simplilearn

1.3M views • 5 years ago

Simplilearn

This video on What is Deep Learningprovides a fun and simple introduction to its concepts. We learn about where **Deep Le...**

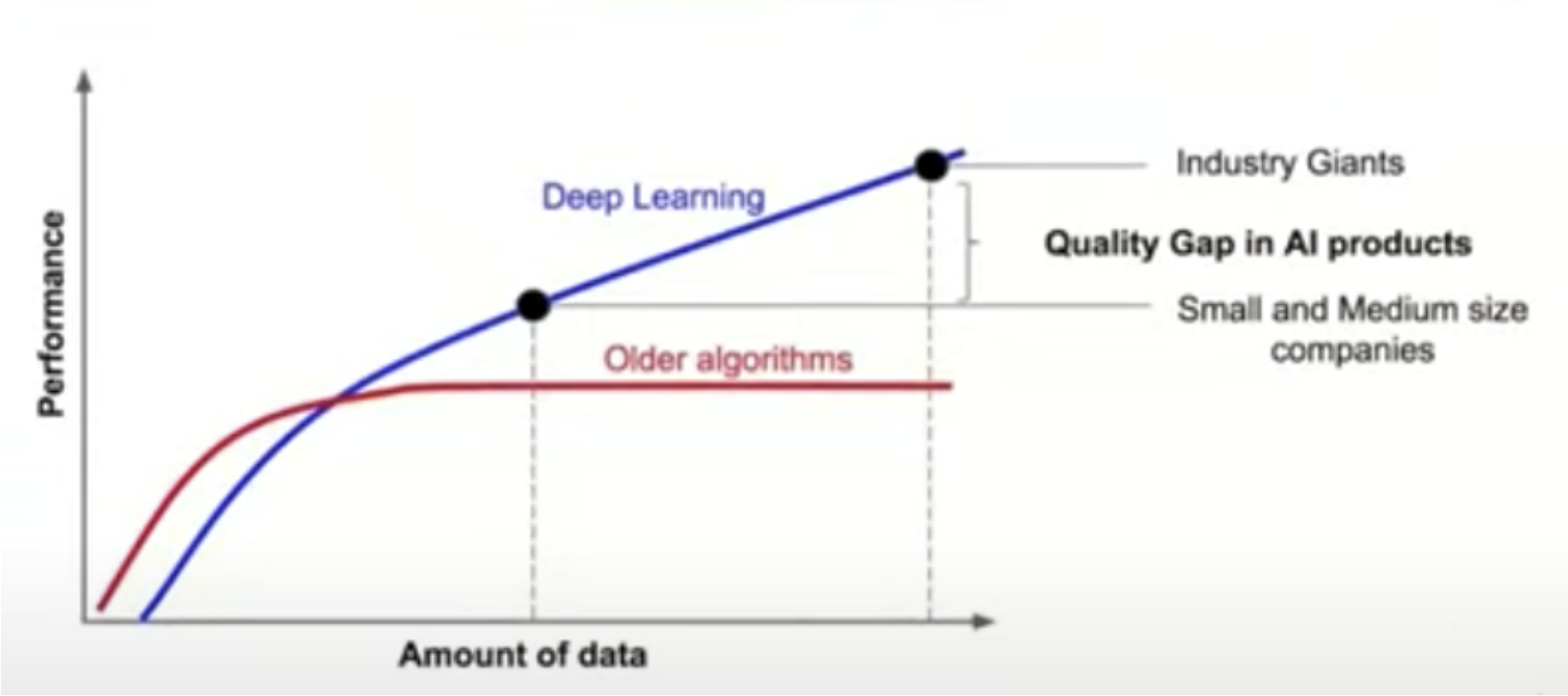
5 chapters Intro | What is Deep Learning | Working of Neural Networks | Where is Deep Learning...

Deep Learning

- https://youtu.be/6M5VXKLf4D4?si=LkBPIbB2mAyms_3u

Why do we need Deep Learning?

- Deep learning is subset of ML algorithm
- It work well if you provide more and more data.
- It prevents overfitting, unlike ML which after a threshold value becomes stagnate.

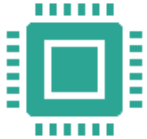


What is deep learning?

Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions

Deep learning technology drives many AI applications used in everyday products,

- Digital assistants
- Voice-activated television remotes
- Fraud detection
- Automatic facial recognition



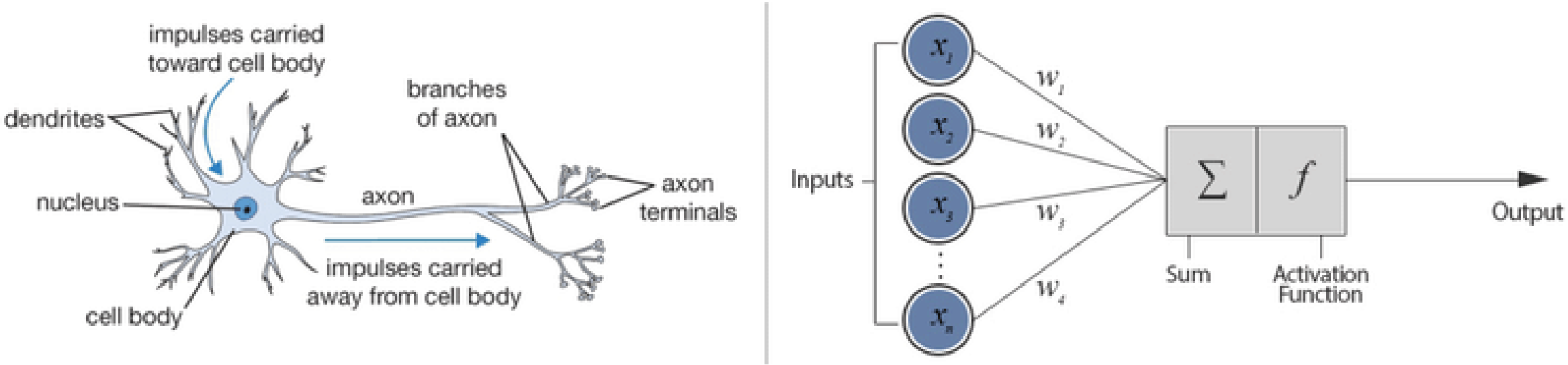
Deep learning models are computer files that data scientists have trained to perform tasks using an algorithm or a predefined set of steps.

Businesses use deep learning models to analyze data and make predictions in various applications.

What is deep learning?

- Deep learning is a subset of machine learning
  - that uses multilayered neural networks, called deep neural networks, to simulate the complex decision-making power of the human brain.
- Deep learning is a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain.

Biological Neuron versus Artificial Neural Network



ANN

Neurons:

The basic units of neural networks that process inputs to produce outputs.

Layers:

ANNs consist of an input layer, hidden layers, and an output layer. "Deep" refers to networks with many hidden layers.

Activation Functions:

Functions like ReLU (Rectified Linear Unit), sigmoid, and tanh introduce non-linearity to help the network learn complex patterns.

Key Architectures

Feedforward Neural Networks (FNN):	The simplest type of ANN where the information moves in only one direction, from input to output.
Convolutional Neural Networks (CNNs):	Primarily used for image recognition tasks, CNNs detect patterns and hierarchies in data using convolutional layers.
Recurrent Neural Networks (RNNs):	Used for sequence data like time series or text, RNNs keep track of past information through their loops, enabling learning from sequences.
Long Short-Term Memory Networks (LSTMs):	A special type of RNN that solves the vanishing gradient problem, allowing learning over long sequences of data.
Autoencoders:	Used for unsupervised learning, autoencoders compress data and then reconstruct it, useful for tasks like image denoising or anomaly detection.
Generative Adversarial Networks (GANs):	Consist of two neural networks (generator and discriminator) that compete, allowing for the generation of realistic synthetic data, such as images or text.

How deep learning works: *Forward Propagation*

Forward propagation refers to the movement of data through the layers of the network from input to output.

Each neuron in one layer is connected to neurons in the next layer, where each connection has a weight.

The input data is multiplied by these weights and passed through an activation function to produce an output, which becomes the input for the next layer.

The process continues until the output layer provides the final prediction.

How deep learning works: ***Backpropagation***

Backpropagation is the learning process where the neural network adjusts its weights and biases to minimize prediction errors:

The network compares its output to the actual result using a **loss function** (which measures error).

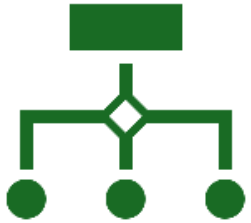
It calculates how much each neuron in each layer contributed to the error.

Using the **gradient descent algorithm**, it adjusts the weights and biases by moving backward from the output layer to the input layer, effectively "learning" from its mistakes and becoming more accurate over time.

Training the Model

Epochs:

Training is done over multiple epochs, where the entire dataset is passed through the network multiple times to adjust weights.



Optimization:

Gradient descent optimizes the model by finding the minimum point of the loss function, where the error is lowest.

Computational Requirements

Deep learning models require massive computational resources

GPUs (Graphical Processing Units):

GPUs are often used to train deep learning models as they can handle parallel computations efficiently, speeding up the training process.

Cloud Computing:

Cloud platforms, like Google Cloud, AWS, and Azure, offer distributed computing power that can scale to meet the computational needs of deep learning models, especially when managing multiple GPUs on-premise becomes costly or resource-intensive.

Deep Learning Frameworks

JAX:

A relatively new framework from Google, designed for high-performance machine learning with easy-to-use automatic differentiation and GPU/TPU support.

PyTorch:

A flexible and user-friendly framework developed by Facebook.

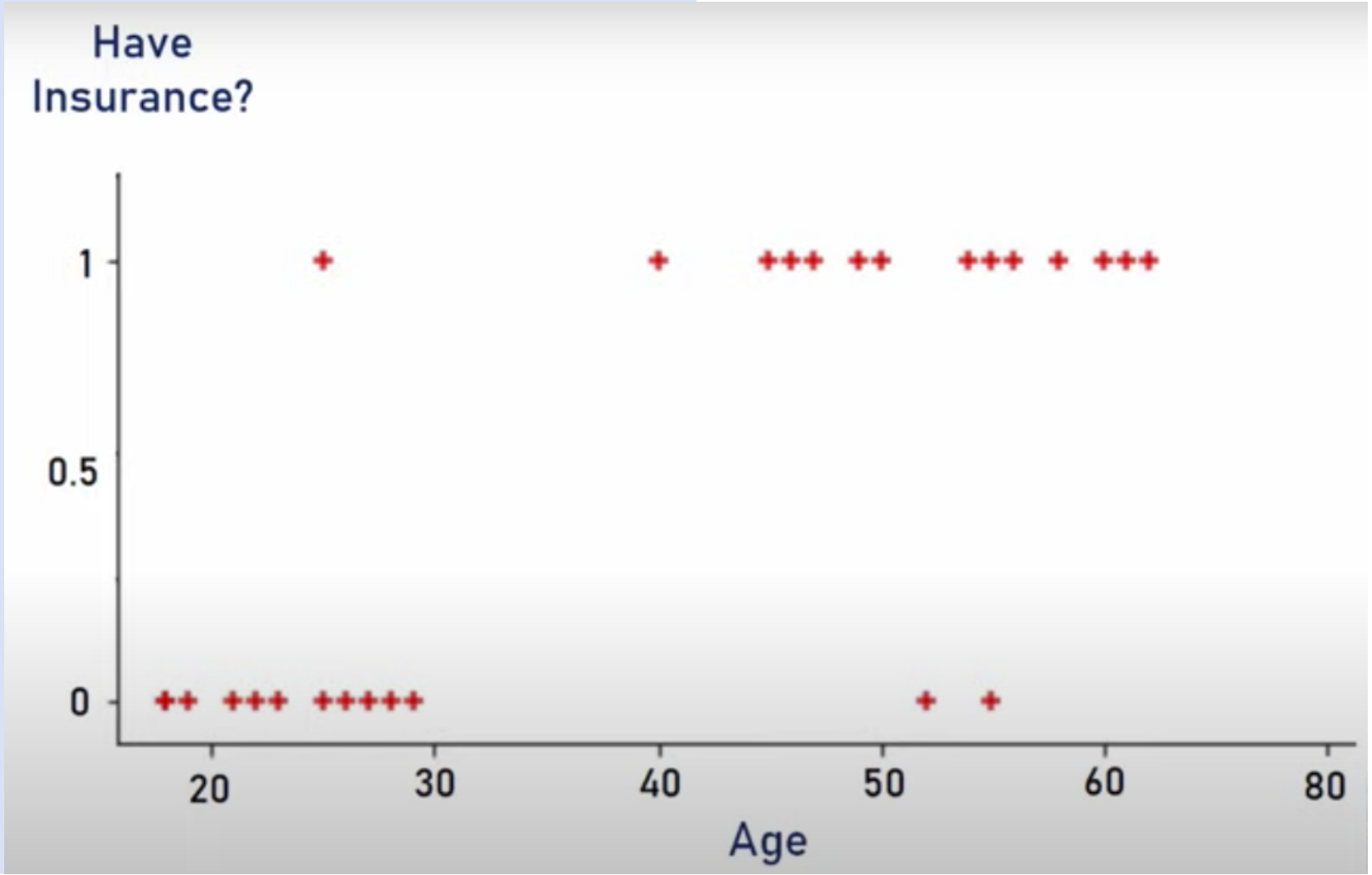
It's popular in research due to its dynamic computational graph and straightforward debugging.

TensorFlow:

Developed by Google, TensorFlow is widely used in both academia and industry due to its scalability and production-ready features.

REGRESSION

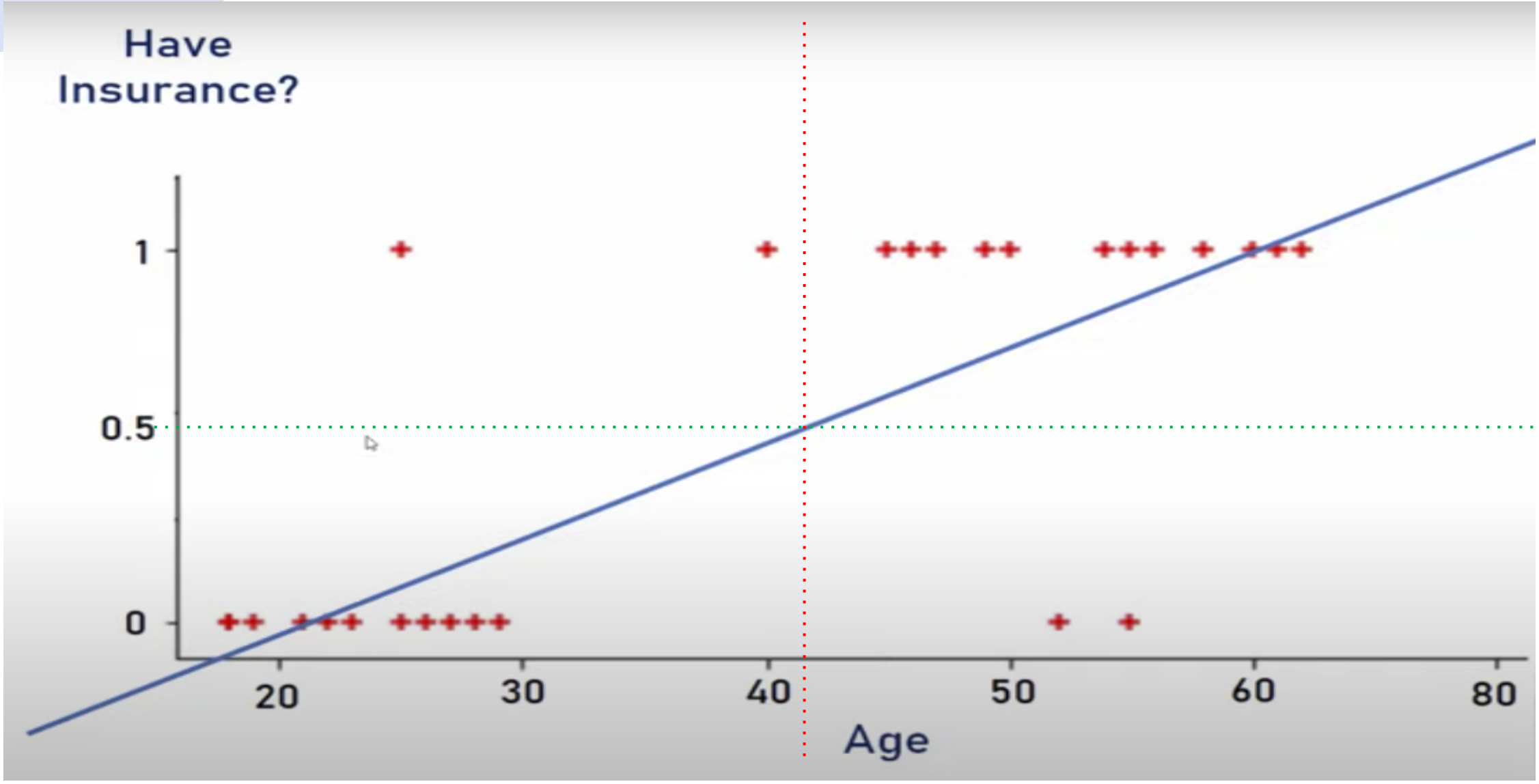
Given an age of person , produce a function that predict if person will buy Insurance or not

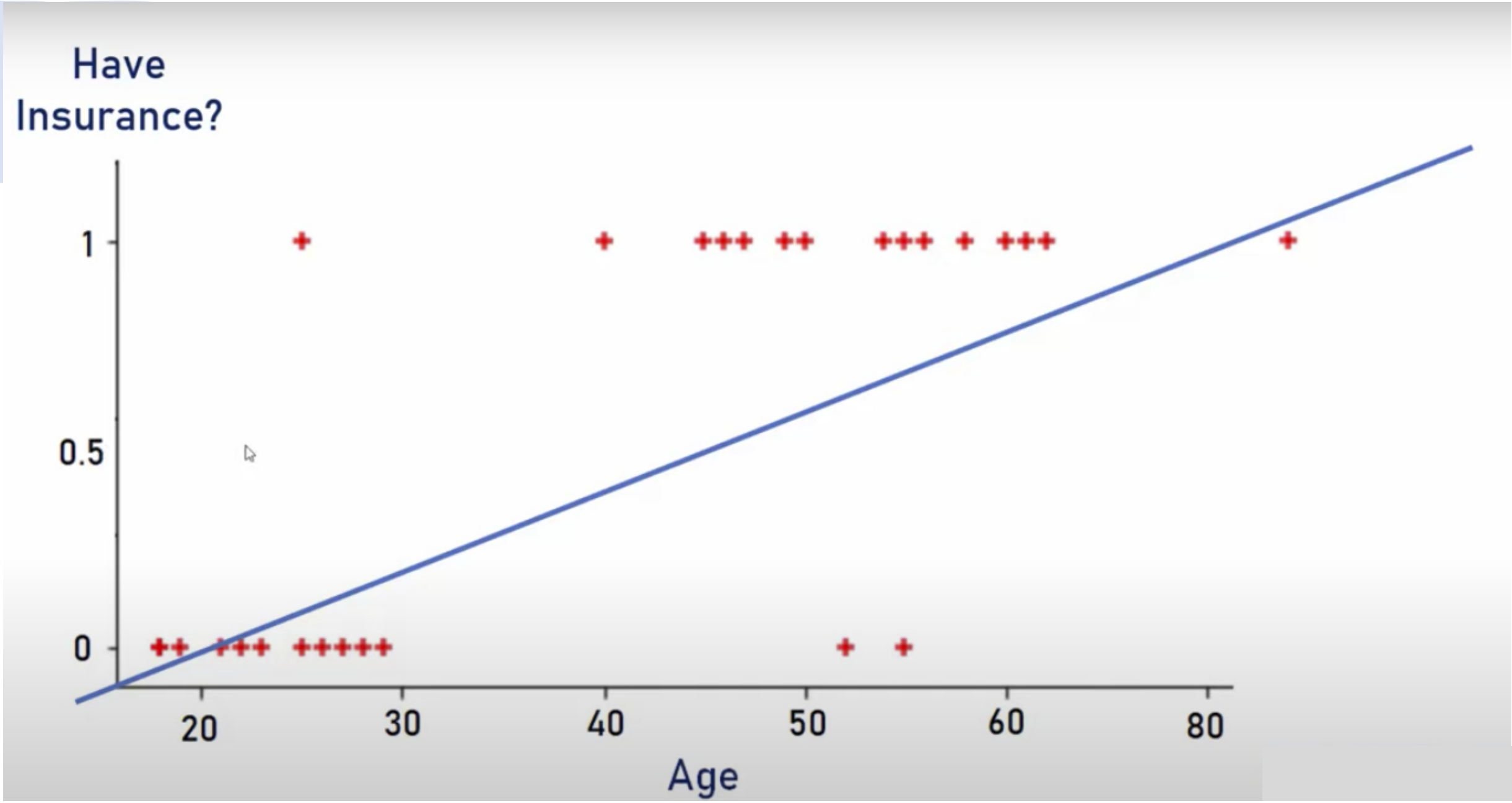


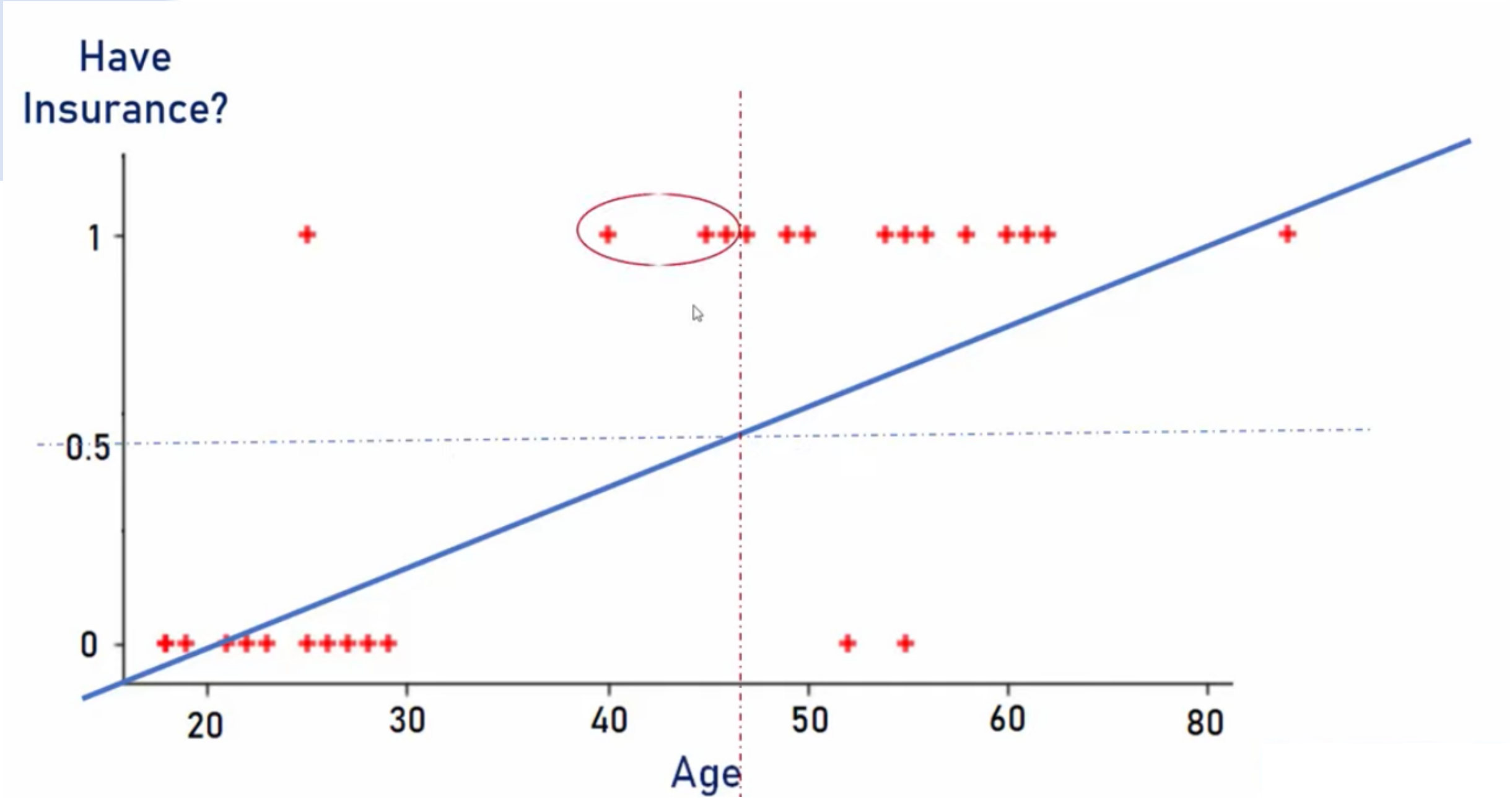
AGE	HAVE_INSURANCE
22	0
25	0
47	1
52	0
46	1
56	1
55	0
60	1
62	1
61	1
18	0
28	0
27	0
29	0
49	1

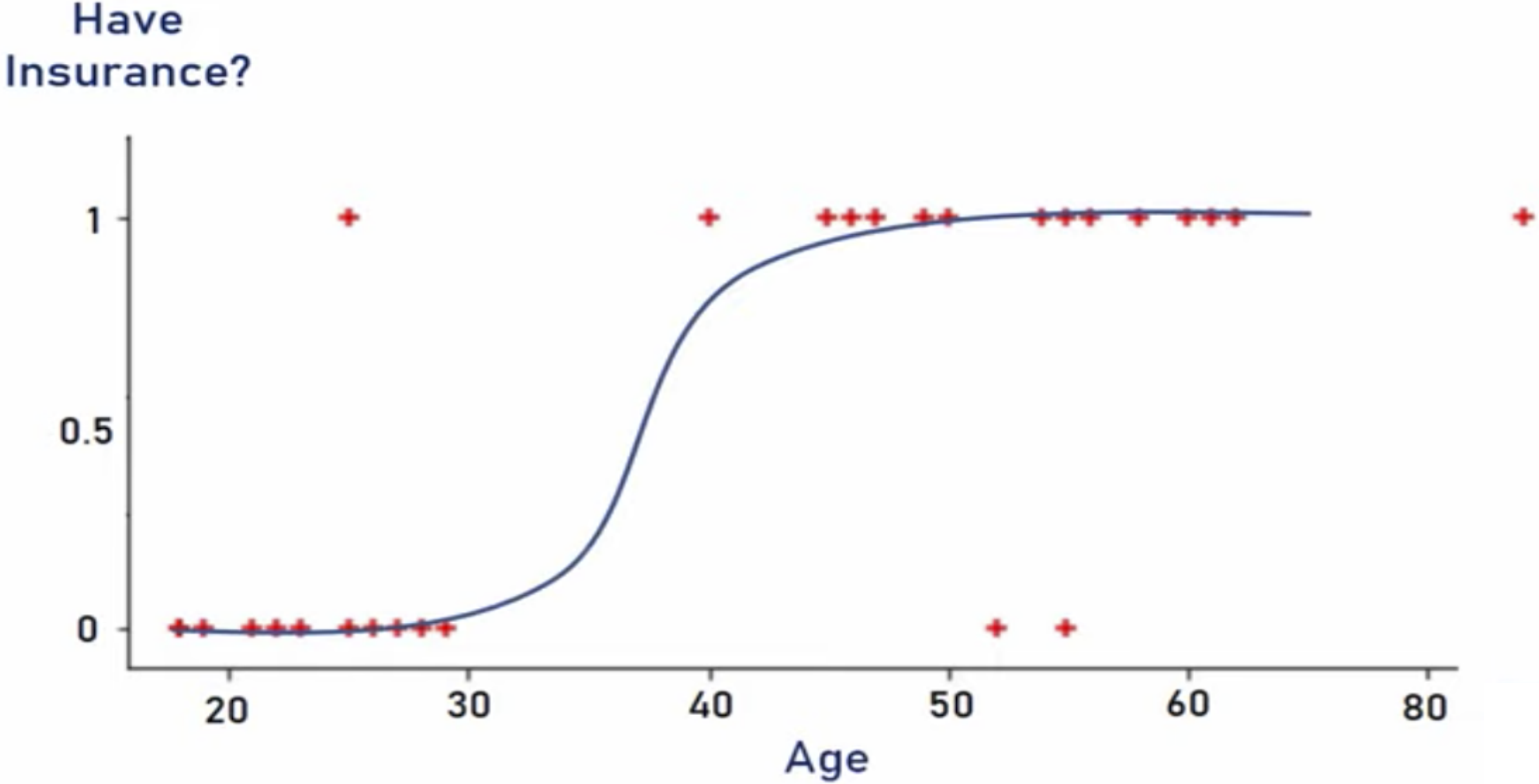
Linear regression

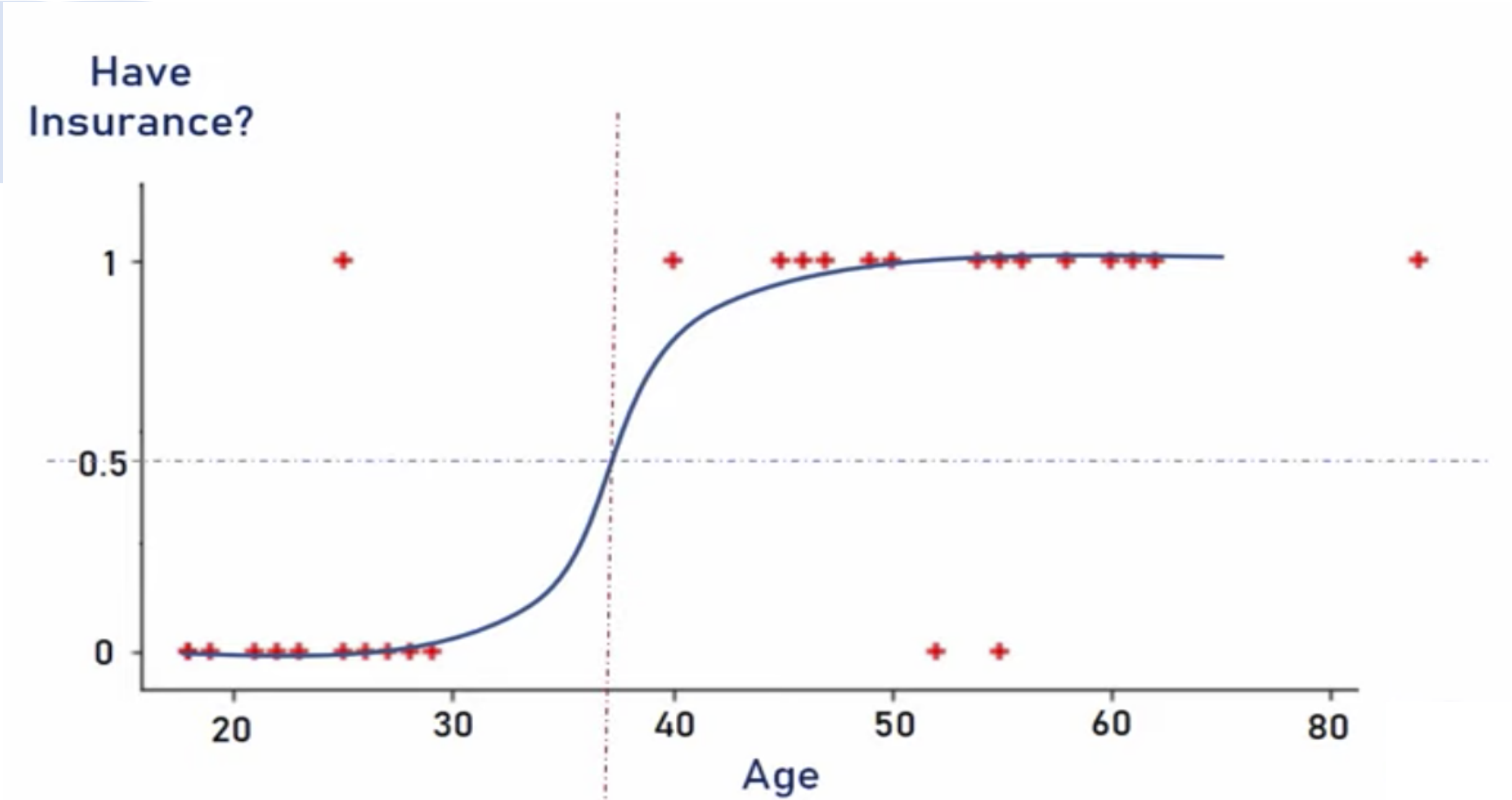
Any value above 0.5 will buy insurance
Any one with age above 42 will buy insurance

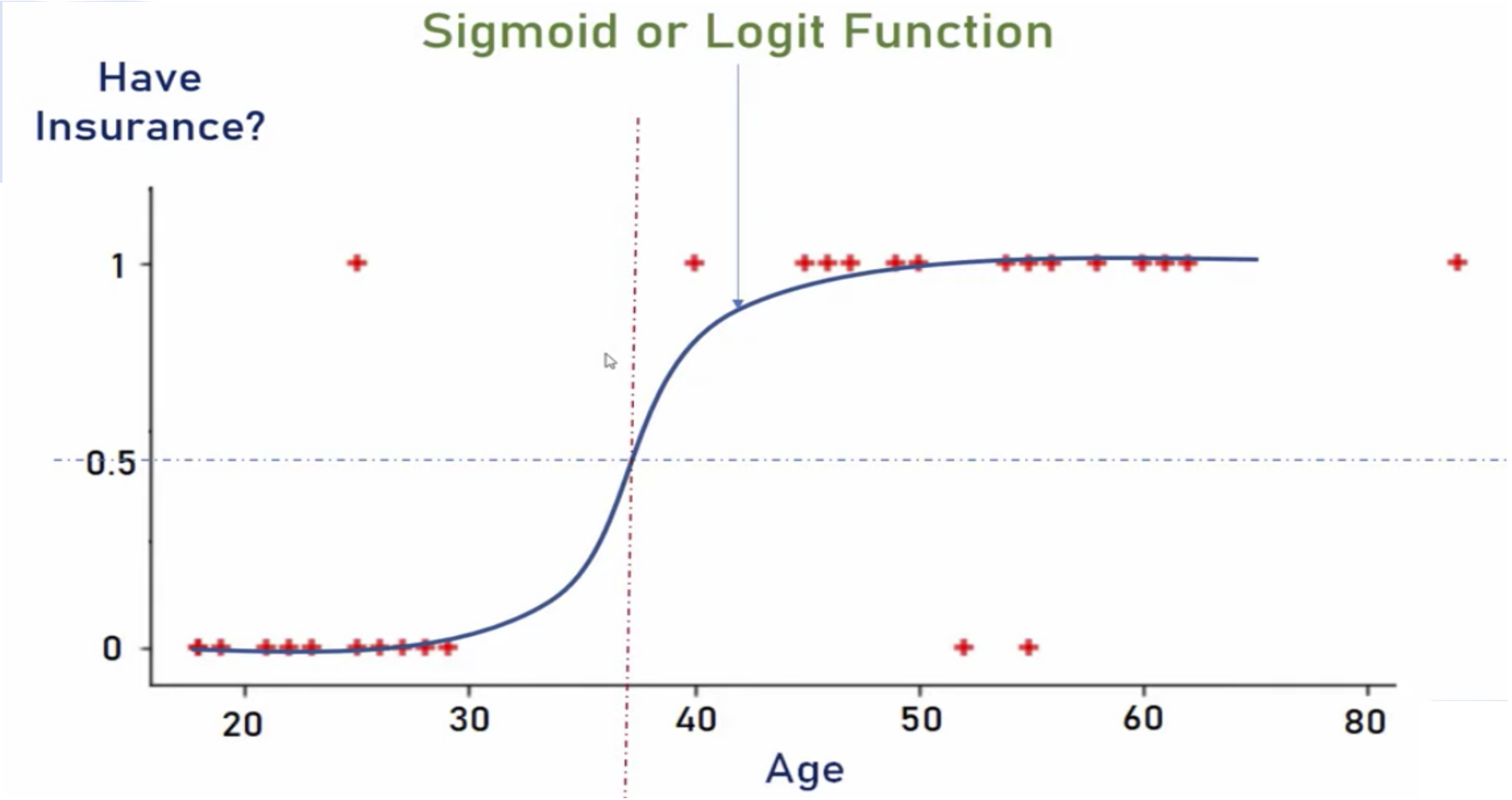



Have
Insurance?
1
0.5
0
20
30
40
50
60
80
Age


Have Insurance?
1
0.5
0
20
30
40
50
60
80
Age


Have
Insurance?
1
0.5
0
20
30
40
50
60
80
Age


Have Insurance?
1
0.5
0
20
30
40
50
60
80
Age


Sigmoid or Logit Function
Have Insurance?
1
0.5
0
20
30
40
50
60
80
Age

Sigmoid Function

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

e = Euler's number ~ 2.71828

$$\text{Sigmoid } (200) = \frac{1}{1+e^{-200}} = \text{almost close to } 1$$

$$\text{Sigmoid } (-200) = \frac{1}{1+e^{200}} = \text{almost close to } 0$$

So, the sigmoid function converts input into range of 0 to 1.

Step 1

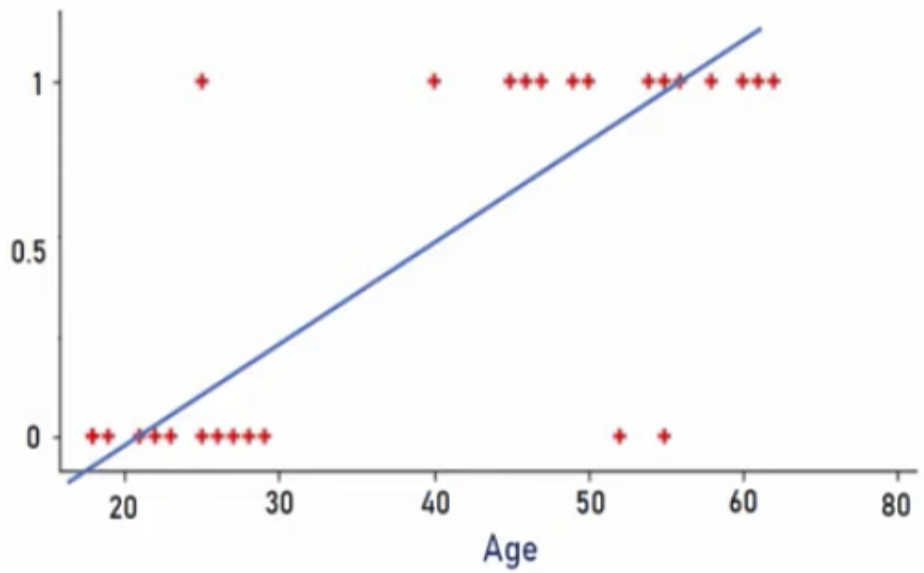
$$y = m * x + b$$

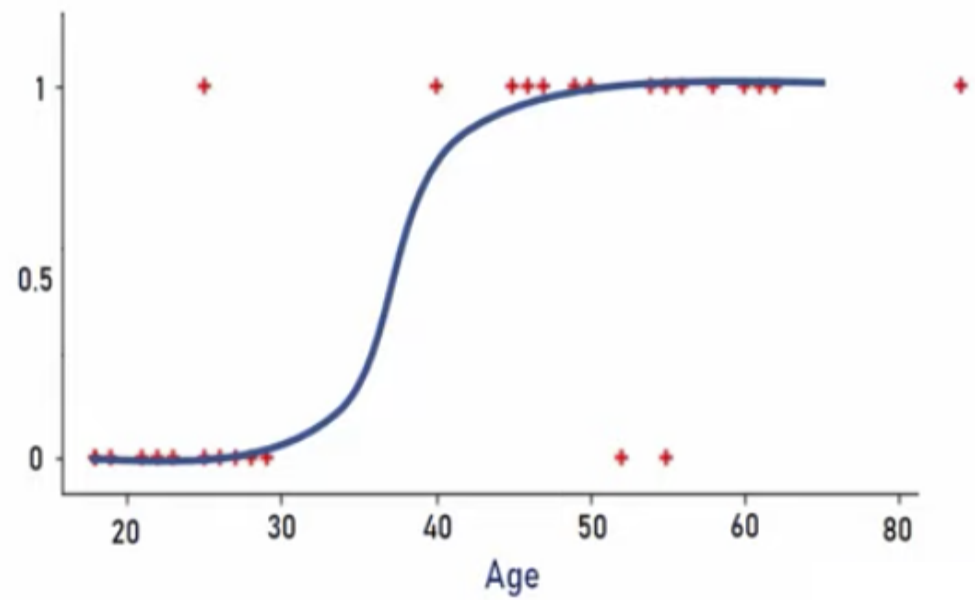
Age

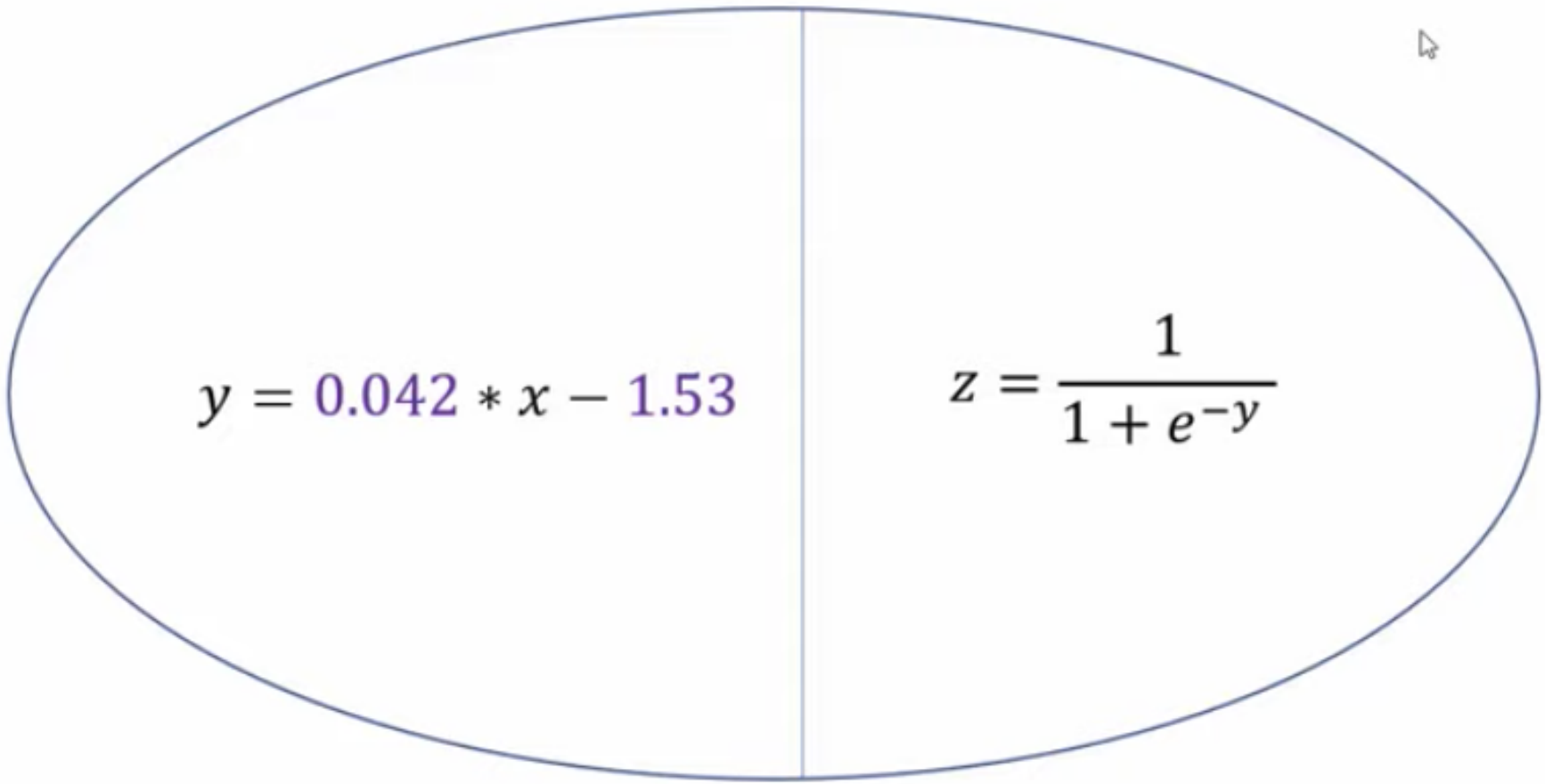
Step 2

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

If person will buy insurance

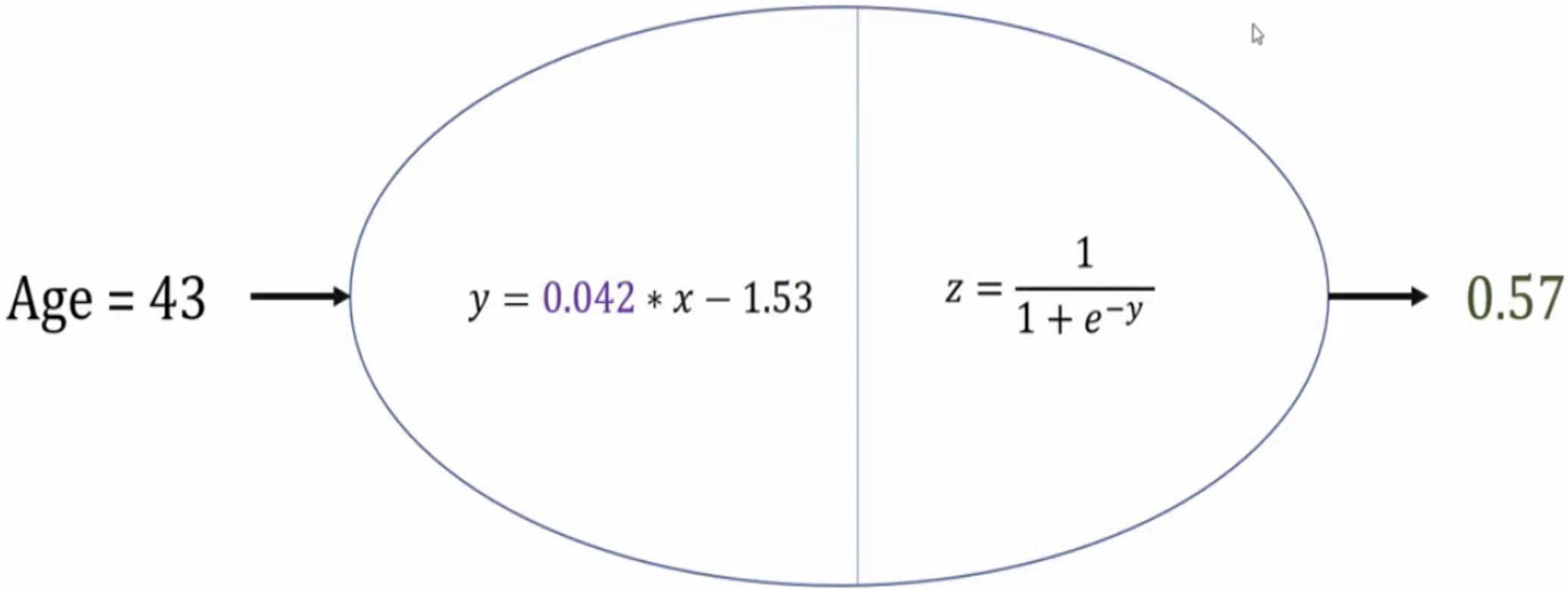


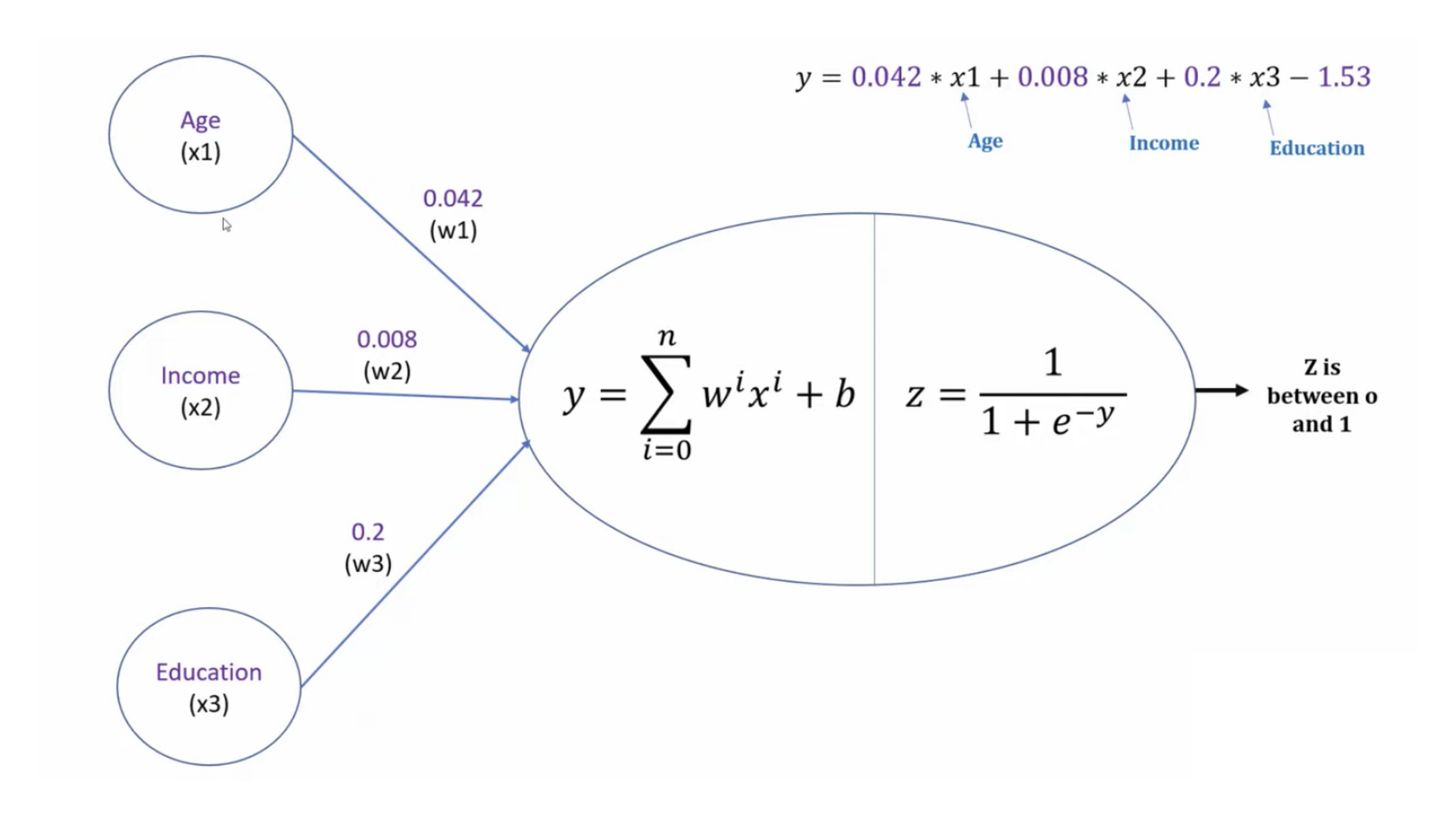



$y = 0.042 * x - 1.53$
$z = \frac{1}{1 + e^{-y}}$

value < 0.5 = person will not buy insurance

value >= 0.5 = person **will** buy insurance

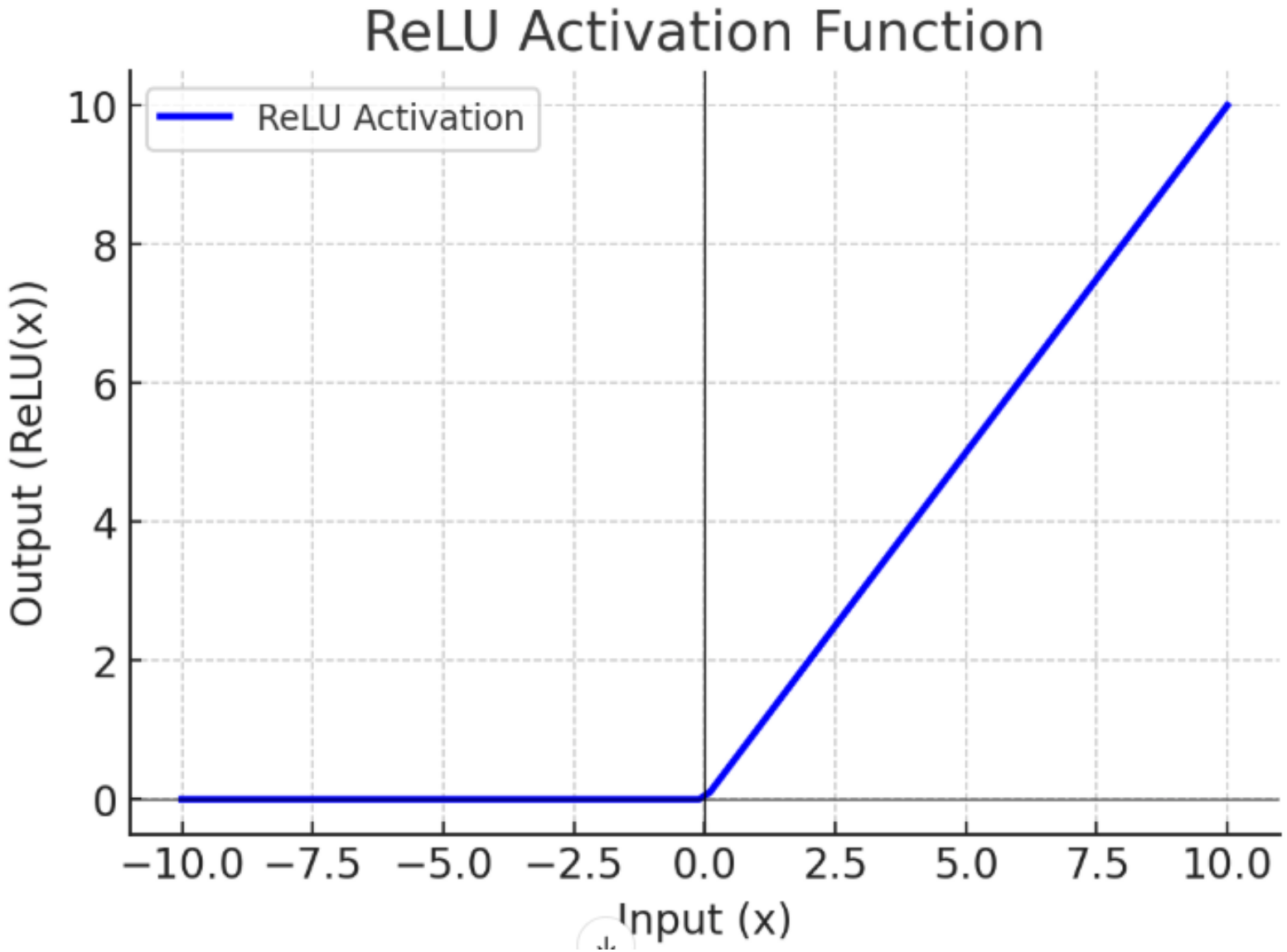



$y = 0.042 * x1 + 0.008 * x2 + 0.2 * x3 - 1.53$
Age
Income
Education
Age
(x1)
Income
(x2)
Education
(x3)
0.042
(w1)
0.008
(w2)
0.2
(w3)
$y = \sum_{i=0}^{n} w^i x^i + b$
$z = \frac{1}{1 + e^{-y}}$
Z is between o and 1

ReLU (Rectified Linear Unit) Activation Function in Neural Networks

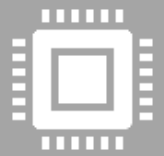
- The **ReLU (Rectified Linear Unit)** activation function is one of the most widely used activation functions in deep learning. It is defined as:
- f(x)=max (0,x)This means:
- If x is positive, the output is xType equation here..
- If x is negative or zero, the output is 0.

$$f(x)=\begin{cases} x, & if\ x > 0 \\ 0 & If\ x < 0 \end{cases}$$


ReLU Activation Function
ReLU Activation
Output (ReLU(x))
Input (x)
10
8
6
4
2
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Properties of ReLU

Non-linear: ReLU introduces non-linearity, allowing neural networks to learn complex patterns.



Computationally Efficient: Simple operation (max(0,x)) makes it fast to compute.

Sparse Activation: Negative values are mapped to 0, reducing the number of active neurons and improving efficiency.

Advantages of ReLU

Prevents Vanishing Gradient: Unlike sigmoid and tanh, ReLU does not saturate for large positive values.

Efficient Computation: Simple max operation speeds up training.

Works Well in Deep Networks: ReLU helps deep neural networks converge faster than sigmoid or tanh.

Limitations and variant of ReLU

Limitations

- **Dying ReLU Problem**: Neurons can become inactive if they always output 0 (for negative inputs), leading to dead neurons.
- **Unbounded Output**: Large positive values can cause instability in training.

Variants of ReLU

- **Leaky ReLU**: f(x)=max(0.01x,x) (solves dying ReLU issue)
- **Parametric ReLU (PReLU)**: Similar to Leaky ReLU but learns the slope parameter

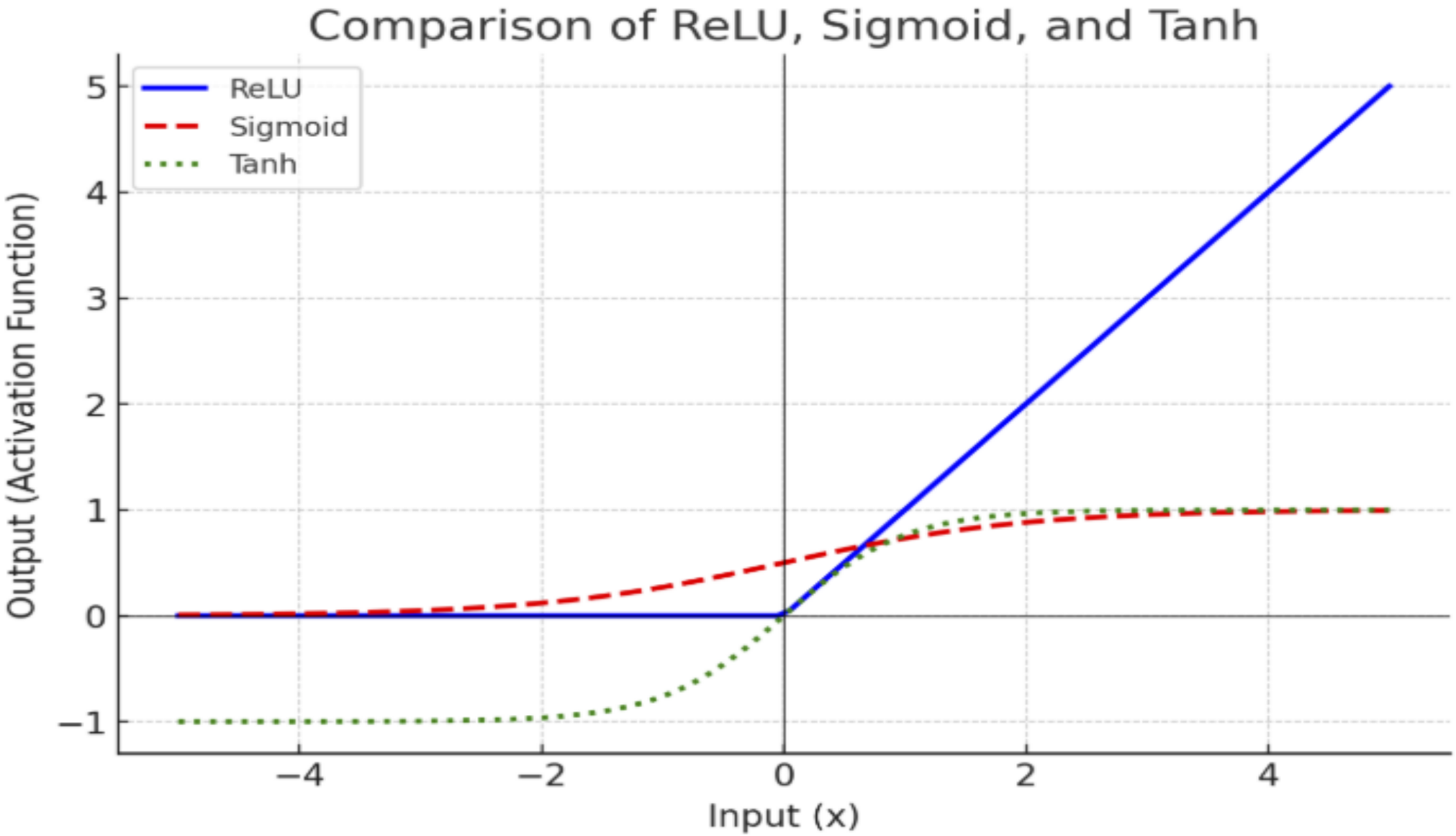
Tanh

•• Tanh (Hyperbolic Tangent)

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

Maps input between **-1 and 1**.Centered around **zero**, unlike sigmoid.

Comparison between different Activation Function



Comparison between different Activation Function

Feature	ReLU	Sigmoid	Tanh
Mathematical Formula	$f(x)=\max(0,x)$		
Output Range	$[0,\infty)$	$(0,1)$	$(-1,1)$
Used in	Deep neural networks	Binary classification	Hidden layers of RNNs, some classifiers
Computational Cost	Very low (simple max operation)	Higher (exponential calculation)	Higher (exponential calculation)
Zero-centered?	❌ No	❌ No	✅ Yes
Vanishing Gradient Issue?	❌ No (for positive values)	✅ Yes	✅ Yes
Exploding Gradient Issue?	❌ No	✅ Yes	✅ Yes
Dying Neuron Problem?	✅ Yes (for negative values)	❌ No	❌ No

Improving Accuracy Over Time

The combination of forward propagation (to make predictions) and backpropagation (to adjust weights and biases) allows deep learning models to continuously improve their accuracy

Over time, as the model processes more data and fine-tunes its parameters, it becomes more adept at making precise predictions or classifications

This cyclical process of learning, predicting, and improving underpins the effectiveness of deep learning models

DERIVATIVES

Slope	**Derivative**
Used for linear equation	Used for nonlinear equation
It is a constant	It is a function

PRACTICE

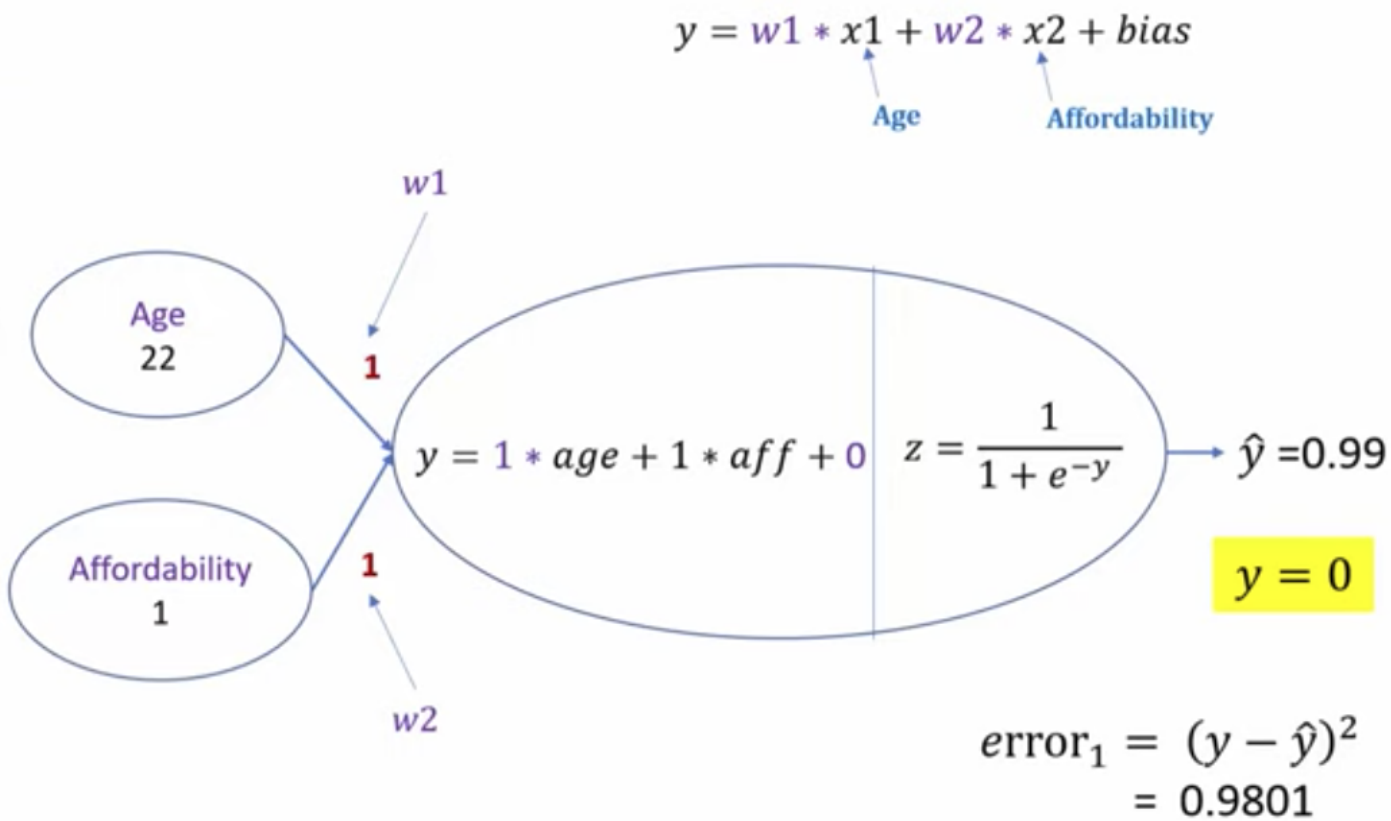
Function	Derivative
$2x4-7y^{20}+z+5$	
$x-3+2y^{1}$	
$\cos(x)\sin(x)$	
$(x+5)(x-2)$	
7	
$9x$	

ANSWERS

Function	Derivative
$2x4-7y^{20}+z+5$	
$x-3+2y^{1}$	
cos(x)sin(x)	−sin2(x)+cos2(x) (using the product rule)
(x+5)(x−2)	(x+5)+(x−2)=2x+3 (using the product rule)
7	0 (constant derivative is zero)
9x	9

Application of derivative in the neural network

age	affordability	have_insurance
22	1	0
25	0	0
47	1	1
52	0	0
46	1	1
56	1	1
55	0	0
60	0	1
62	1	1
61	1	1
18	1	0
28	1	0
27	0	0



Explanation

- **Input Data:**
  - **Age:** Numeric input representing the age of the individual.
  - **Affordability:** Binary input (0 or 1) indicating whether the individual finds the insurance affordable.
- **Neural Network Equation:**
  - The neural network computes the output using the equation:

    $$y = w1 \cdot x1 + w2 \cdot x2 + bias$$
  - w1,w2 Weights for the features x1 (age) and x2 (affordability).
- bias: Constant added to the weighted sum.
- Here, w1=1 1, w2=1 and bias=0.
- Substituting values:

  $y = 1 \cdot age + 1 \cdot affordability + 0$

Explanation

- **Sigmoid Activation Function:**
  - After computing y, the output z is passed through the sigmoid activation function to transform it into a probability:
  - Sigmoid (z) = $\frac{1}{1+e^{-z}}$
  - The sigmoid function outputs values between 0 and 1, representing the predicted probability y^\hat{y}y^ of having insurance.

- **Example Computation (Highlighted Row):**
- **Age = 22, Affordability = 1, Have Insurance = 0.**
  - Compute y: $y = 1 \cdot 22 + 1 \cdot 1 + 0 = 23$
- Compute z (using the sigmoid function):

$$z = 11 + e - 23 \approx 0.99$$

So, the predicted probability $\hat{y} = 0.99$

- **Actual Output (y):**
- The actual output y is 0 (no insurance).
- The prediction ($\hat{y}=0.99$) is incorrect.
- **6. Error Calculation:**
- The error is calculated using the squared error formula:
- $\text{Error}=(y-\hat{y})^2$
- Substituting values:
  - $\text{Error}=(0-0.99)^2=0.9801$

Multilayer Networks as Composition Functions

A multilayer neural network evaluates compositions of functions computed at individual nodes.

Example: A path of length 2, where function f(·) follows g(·), represents a composition f(g(·)).

Illustrative Exampl

$$f(g(w)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-w)}\right]}$$

Consider a simple computational graph with two nodes, each applying the sigmoid function to the input weight w:

This is a composition of two sigmoid functions, and computing its derivative with respect to ww is already non-trivial.

Complexity in Multilayer Networks

General Case:

- In layer m, functions g1(·),g2(·),...,gk(·)$g1(\cdot), g2(\cdot), \ldots, gk(\cdot)$ are computed.
- These feed into a node in layer m+1, which computes f(·).
- The composition function at the layer m+1 node is: f(g1(·),g2(·),...,gk(·))
- This is a **multivariate composition function**, which becomes increasingly complex as the network grows deeper.

Recursive Nesting:

- The loss function L depends on the network's output, which is a recursively nested function of the weights in earlier layers.
- For a network with 10 layers and 2 nodes per layer, the loss function becomes a summation of 2^{10} (1024) recursively nested terms.
- Computing partial derivatives of such a function directly is computationally infeasible.

Breakthrough with Backpropagation Algorithm and Resurgence

- In the 1980s, Rumelhart et al. proposed the backpropagation algorithm.
- This rekindled interest in neural networks.
- However, challenges such as computational inefficiency, stability issues, and overfitting were encountered.
- Due to these challenges, neural network research again lost favor.
- **Resurgence of Neural Networks in the 21st Century**
- Several factors led to the renewed popularity of neural networks:
  - Increased availability of data.
  - Improved computational power (e.g., GPUs, TPUs).
  - Algorithmic advancements such as pretraining and modifications to backpropagation.
- Faster experimentation and reduced testing cycles enabled better algorithmic adjustments.

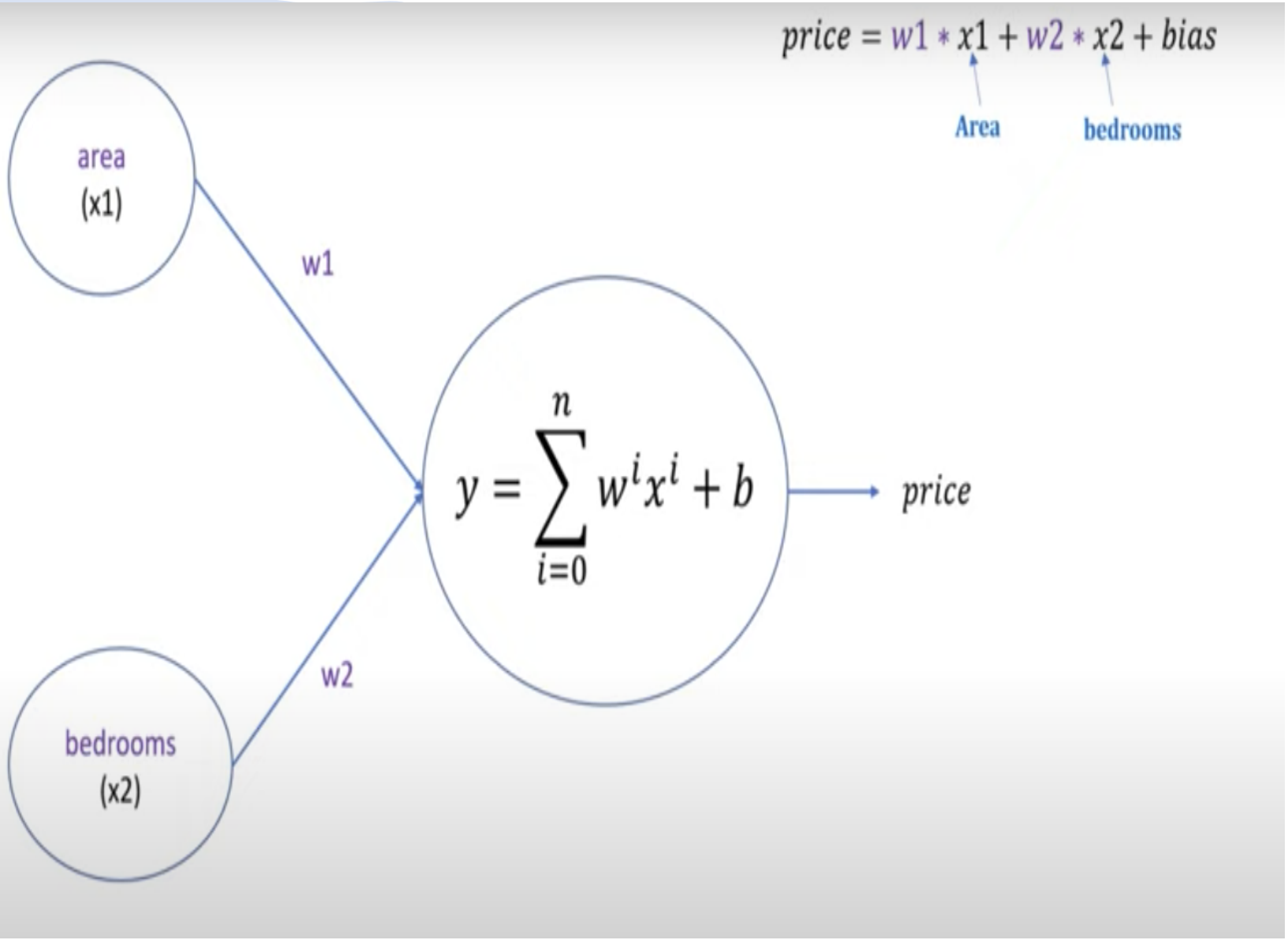
Need for an Iterative Approach

Directly computing derivatives of deeply nested functions is impractical due to exponential growth in complexity.

Use an **iterative approach** based on **dynamic programming** and the **chain rule** of differential calculus.

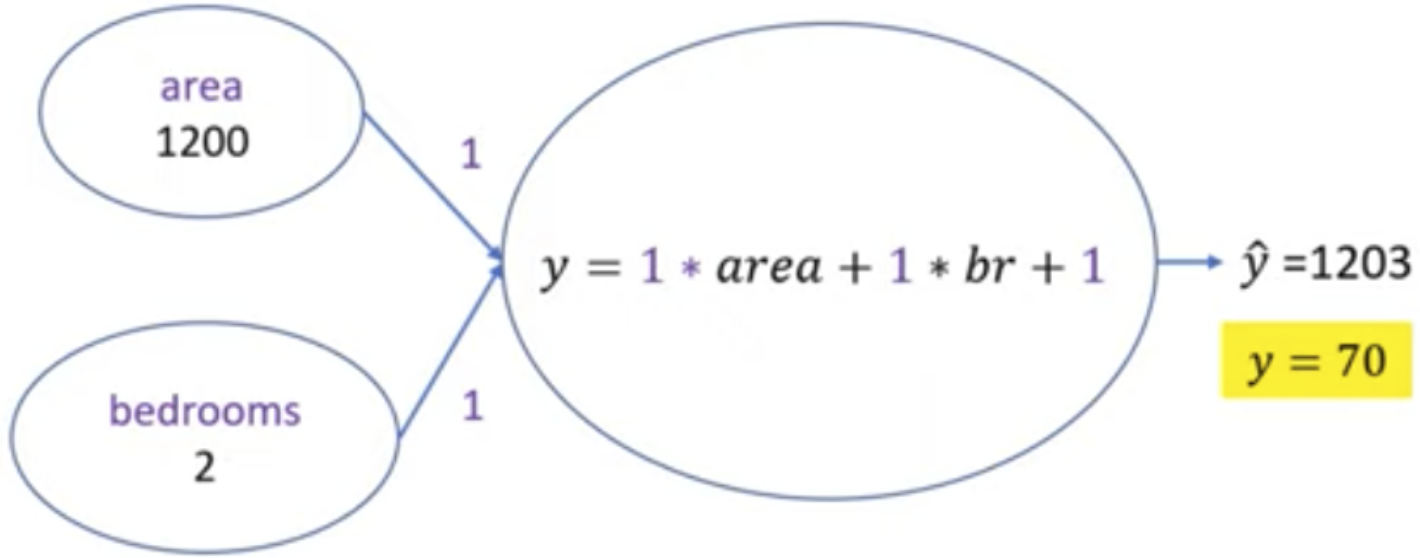
The Chain Rule

Chain Rule Basics:	For a composition of functions $y=f(g(x))$, the derivative of y with respect to x is: This rule can be extended to multivariate and deeply nested functions.
Application to Neural Networks:	The chain rule allows us to compute gradients layer by layer, starting from the output and moving backward to the input. This approach is called **backpropagation**.



Area	Bedrooms	Price
1200	2	70
1500	3	120
2100	4	230
1100	3	105
1300	2	90
900	1	55

area	bedrooms	price
1200	2	70
1500	3	120
2100	4	230
1100	3	105
1300	2	90
900	1	55



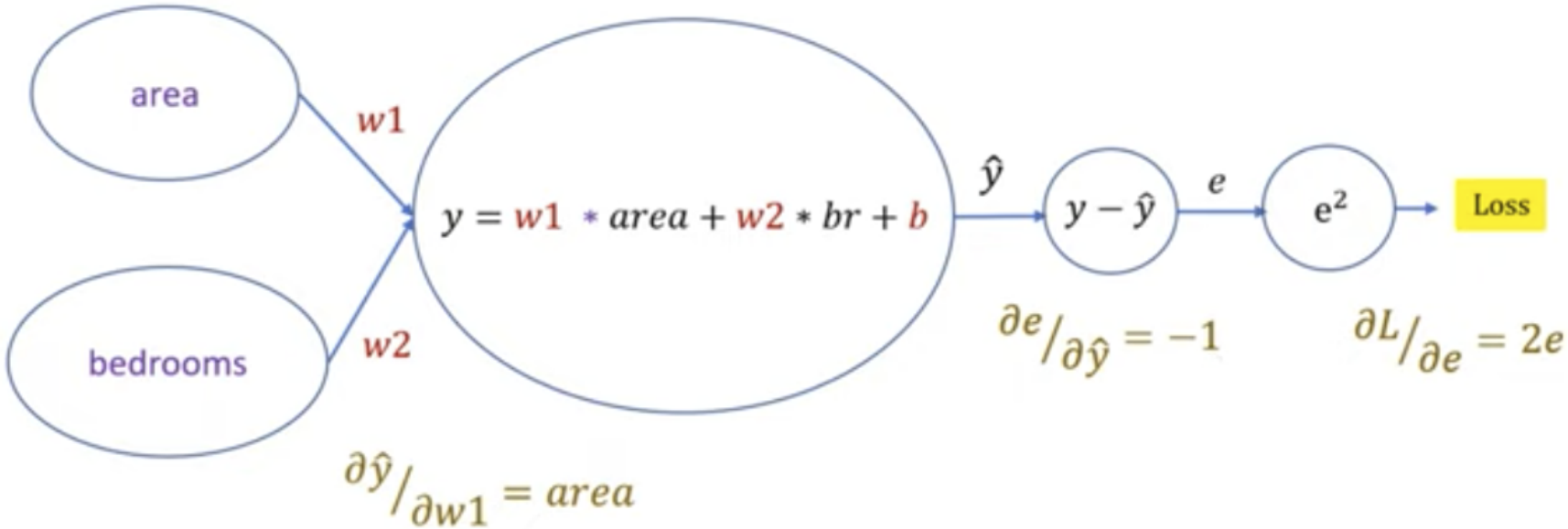
$error1 = y - \hat{y}$ = 1133

$squared\ error1$ = 1283689

Total squared error = $sq\ error_1 + \cdots + \text{sq}\ error_6$

Mean squared error = $(sq\ error_1 + \cdots + \text{sq}\ error_6)$ / 6

$$Loss = Mean\ Squared\ Error$$



$$\partial L/\partial w1 = ?$$

$$\partial L/\partial w1 = \partial L/\partial e * \partial e/\partial \hat{y} * \partial \hat{y}/\partial w1$$

Chain Rule

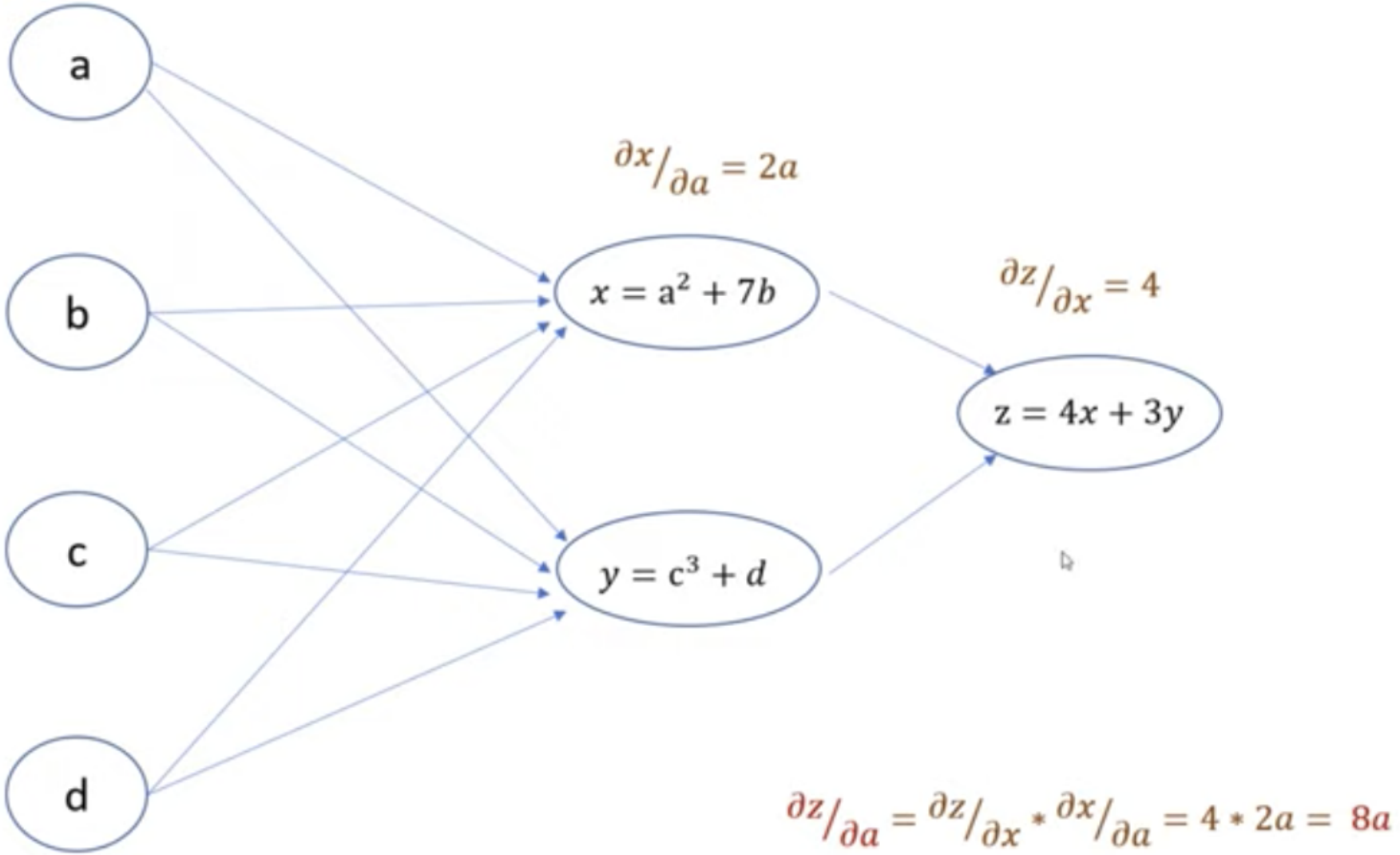
$$\partial L/\partial w1 = -2e * area$$

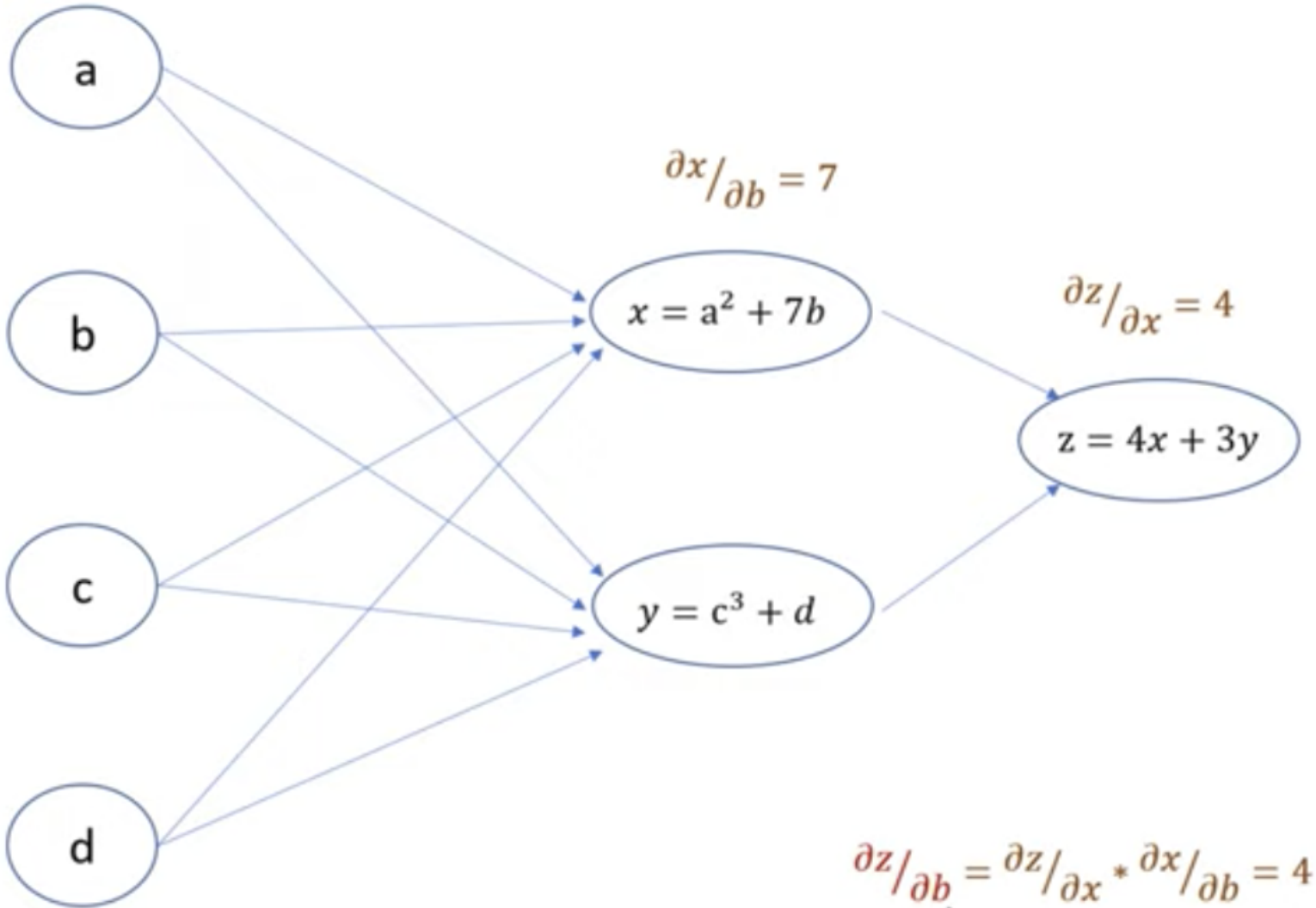
$$w1 = w1 - something$$

$$w1 = w1 - learning\ rate * {}^{\partial}/_{\partial w1}$$

$$w2 = w2 - learning\ rate * {}^{\partial}/_{\partial w2}$$

$$b = b - learning\ rate * {}^{\partial}/_{\partial b}$$


a
b
c
d
$x = a^2 + 7b$
$y = c^3 + d$
$z = 4x + 3y$
$\partial x/\partial a = 2a$
$\partial z/\partial x = 4$
$\partial z/\partial a = \partial z/\partial x * \partial x/\partial a = 4 * 2a = 8a$



$$\partial z/\partial b = \partial z/\partial x * \partial x/\partial b = 4 * 7 = 28$$

Dynamic Programming and Backpropagation

Dynamic Programming:

A method to solve complex problems by breaking them into simpler subproblems and storing intermediate results.

In neural networks, this means computing and storing gradients at each layer during the backward pass.

Backpropagation:

Forward Pass: Compute the output of the network by evaluating functions layer by layer.

Backward Pass:

- Compute the gradient of the loss with respect to the output.
- Use the chain rule to propagate gradients backward through the network.
- At each layer, compute gradients with respect to the weights and inputs, using stored intermediate results from the forward pass.

Example: Backpropagation in a Simple Network

- **Network Structure**:
  - Input x, hidden layer with function g(w), output layer with function f(g(w)).
- **Gradient Computation**:
  - Compute $\frac{\partial L}{\partial f}$, the gradient of the loss with respect to the output.
  - Compute $\frac{\partial f}{\partial g}$, the gradient of f with respect to g.
  - Compute $\frac{\partial g}{\partial w}$, the gradient of g with respect to w.
  - Apply the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial g} \frac{\partial g}{\partial w}$$

Chain Rule in Computational Graphs

- The chain rule is essential in computational graphs for calculating partial derivatives iteratively.
- This iterative approach is the foundation of dynamic programming in differential calculus.
  - **Dynamic Programming:** Used for computing derivatives iteratively.
  - **Chain Rule of Differential Calculus:** Allows for calculating derivatives in a composition of functions.

Simple Computational Graph with Two Nodes

- **Mathematical Formulation:**
- Given functions:

$$\frac{\partial f(g(w))}{\partial w} = \frac{\partial f(g(w))}{\partial g(w)} \cdot \frac{\partial g(w)}{\partial w}$$

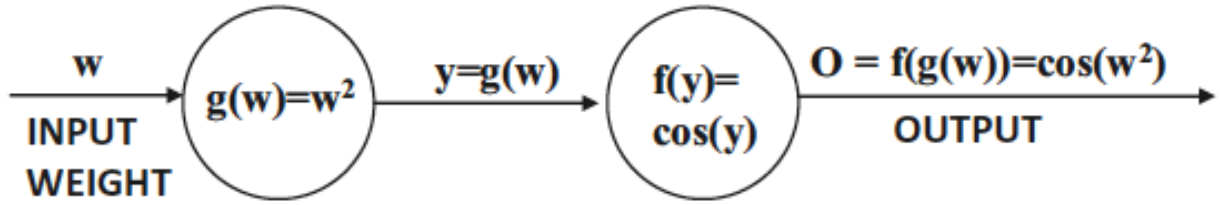


Figure 3.1: A simple computational graph with two nodes

Simple Computational Graph with Two Nodes

- **Application of Chain Rule:**
  - **Univariate Chain Rule:** Calculates derivatives for functions like $\cos(w^2)$
  - Example:

$$\frac{\partial f(g(w))}{\partial g} = -2w.\text{sine}(w^2)$$

Multivariate Chain Rule

- When a hidden layer receives inputs from multiple units, the multivariate chain rule is applied.
- f(g1(w), . . . gk(w)), in which a unit computing the multivariate function f(.) gets its inputs from k units computing g1(w) . . . gk(w). In such cases,
- the multivariable chain rule needs to be used. The multivariable chain rule is defined as follows:

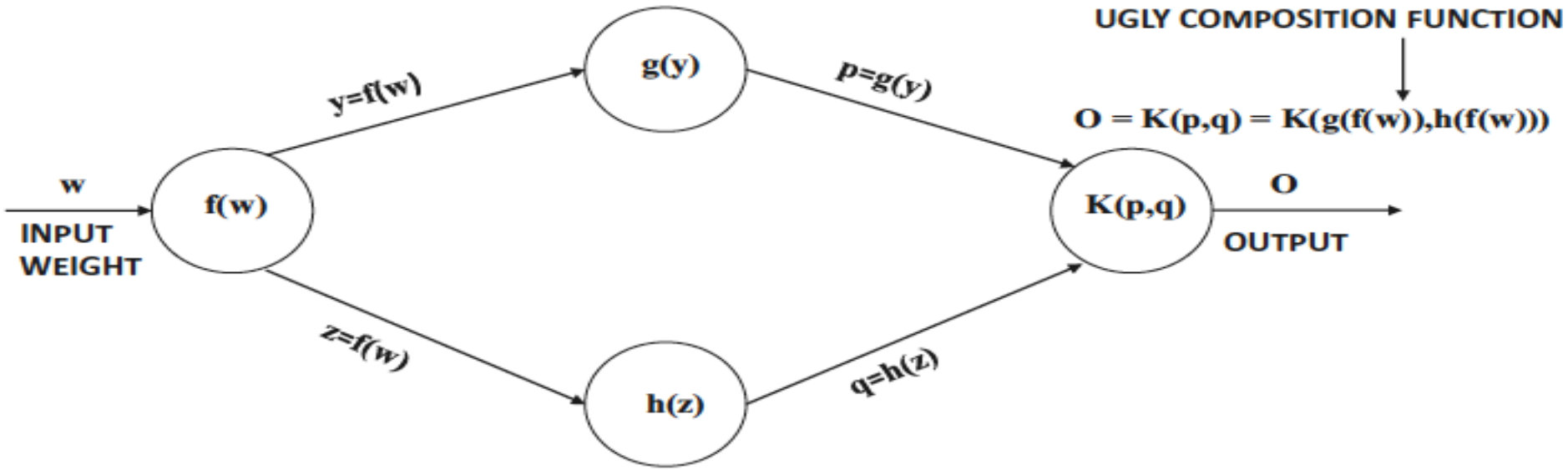
$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w} \qquad (3.3)$$

Multivariate Chain Rule

- Examle :Consider a directed acyclic computational graph in which the ith node contains variable y(i).
- The local derivative z(i, j) of the directed edge (i, j) in the graph is defined as z(i, j) = $\frac{\partial(j)}{\partial(i)}$.
- Let a non-null set of paths P exist from variable w in the graph to output node containing variable o.
- Then, the value of $\frac{\partial o}{\partial w}$is given by computing the product of the local gradients along each path in P, and summing these products over all paths.

$$\frac{\partial O}{\partial w} = \sum_{p \in P} \prod_{(i,j \in P} z(i,j)$$

Multivariate Chain Rule

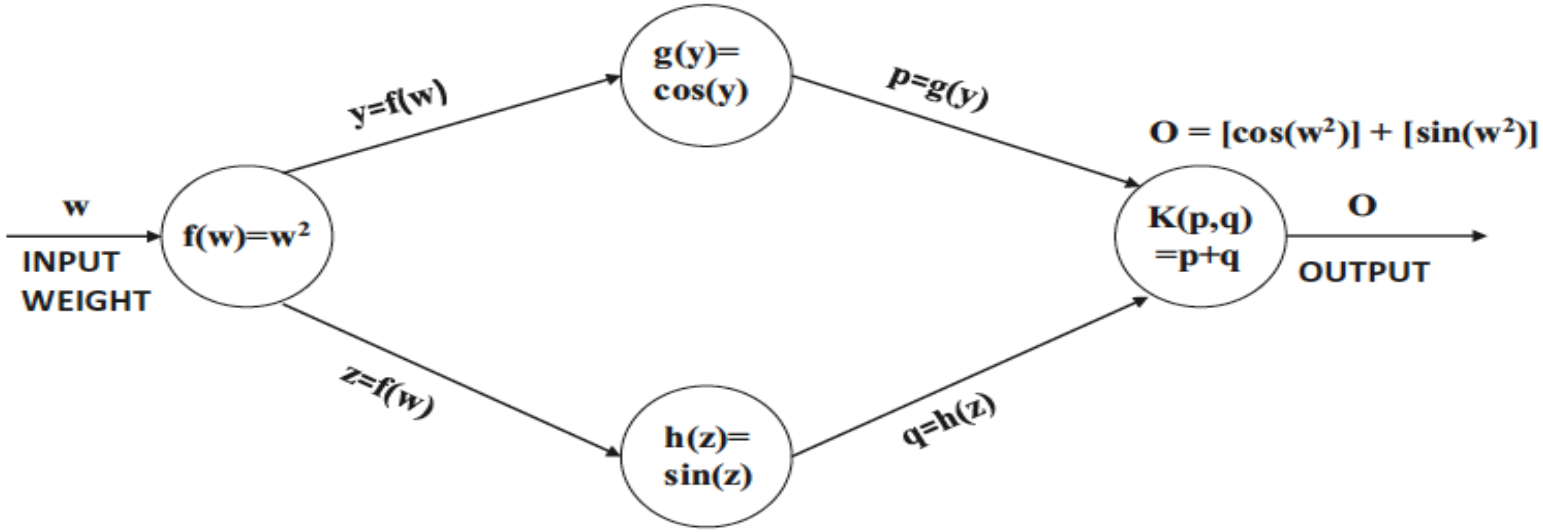


$$\frac{\partial o}{\partial w} = \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial w} \quad \text{[Multivariable Chain Rule]}$$

$$= \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial w} \quad \text{[Univariate Chain Rule]}$$

$$= \underbrace{\frac{\partial K(p,q)}{\partial p} \cdot g'(y) \cdot f'(w)}_{\text{First path}} + \underbrace{\frac{\partial K(p,q)}{\partial q} \cdot h'(z) \cdot f'(w)}_{\text{Second path}}$$

Appling chain rule



$$\frac{\partial o}{\partial w} = \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{-\sin(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w}$$

$$= -2w \cdot \sin(y) + 2w \cdot \cos(z)$$

$$= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)$$

Figure 3.3: An example of the chain rule in action based on the computational graph of

Dynamic programming

The goal is to compute a sum over all paths from a source node w to an output node o

where each edge (i,j) has an associated value z(i,j) (interpreted as a local partial derivative).

The summation, given an exponential number of paths, is computed efficiently using **dynamic programming**.

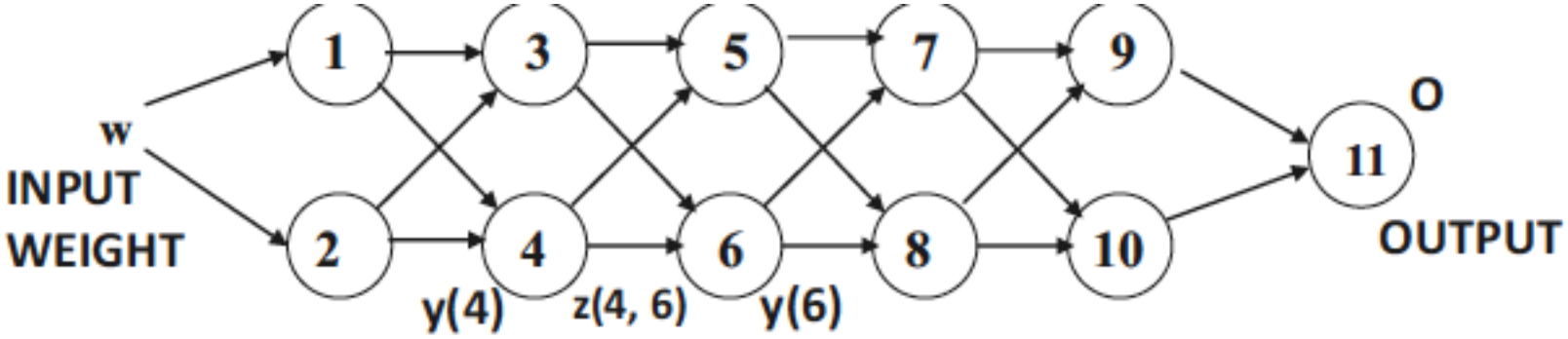
Dynamic Programming Update Rule

- **Input and Output**:
  - The process starts with an input value at the source node w.
  - Computations propagate along the edges, with each edge modifying the value using its associated z(i,j)
  - Each node i computes its aggregated value S(i,o) which depends on its outgoing edges:

$$S(i,o)=\sum_{j\in A(i)} S(j,o)\cdot z(i,j)$$

- The computation proceeds **backward**, starting from the output node where S(o,o)=1.

Dynamic Programming Update Rule

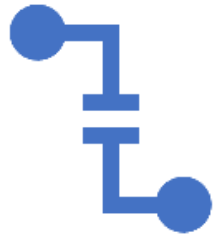


EACH NODE i CONTAINS y(i) AND EACH EDGE BETWEEN i AND j CONTAINS z(i, j)
EXAMPLE: z(4, 6)= PARTIAL DERIVATIVE OF y(6) WITH RESPECT TO y(4)

Dynamic Programming Application

Forward Pass:

In the forward computation (e.g., during inference in a neural network), values flow from the input node www to the output node o, combining contributions along edges.



Backward Pass:

During backpropagation (gradient computation), the algorithm propagates gradients backward, starting from the output node o, and calculates derivatives efficiently using the dynamic programming update rule:

$$S(i,o)=\sum_{j\in A(i)} S(j,o)\cdot z(i,j)$$

Application to Neural Networks

Weights and Gradients:

- z(i,j) often includes terms like the **weight of the edge** (i,j) and the **derivative of an activation function** at the connected node.
- For example:
  - Post-activation variables:
- z(4,6)=weight × activation derivative at Node 6
  - Pre-activation variables:
- z(4,6)=activation derivative at Node 4 × weight

Gradient Descent:

- The aggregated value S(w,o) is used to update weights in gradient descent, enabling efficient learning.

Connection to Chain Rule

- The dynamic programming update is identical to the **multivariable chain rule** used in differentiation.
- This is the mathematical foundation for **gradient-based learning** in neural networks.

Benefits of Backpropagation

Efficiency:

Avoids redundant computations by reusing intermediate results.

Scales well to deep networks with many layers and parameters.

Modularity:

Each layer's gradient computation is independent, making it easy to extend to complex architectures.

Backpropagation and Post-Activation Variables

*Two Phases of Backpropagation :**Forward Phase***

- **Objective**: Compute intermediate variables (hidden units) and output o for a given input X, using current weights.
- **Steps**:
  - Cascade computations forward across the layers.
  - Compute:
    - The output o
    - The loss L based on the difference between o and the true label y.
    - The derivative $\frac{\partial L}{\partial o}$ for single or multiple outputs.

Backward Phase:

- **Objective**: Compute the gradients of the loss L with respect to weights.
- **Steps**:
  - Start with $\frac{\partial L}{\partial o}$ (initialization).
  - Use the **multivariable chain rule** to propagate derivatives backward through the network layers.
  - Aggregate the contributions of all paths connecting hidden nodes to the output node o for gradient computation.

Gradient evaluation

- **1.Recursive Relation for Gradients:**
  1. For a hidden node h_r:

$$\Delta(h_r,o)=\frac{\partial L}{\partial h}=\sum_{h\in \text{later nodes}}\Delta(h,o)\cdot\frac{\partial h}{\partial h_r}$$

2.Weight Gradients:

1. For weights connecting h_{r-1} and h_r:

$$\frac{\partial L}{\partial h}(h_{r-1},h_r)=\Delta(h_r,o)\cdot h_{r-1}\cdot\Phi'(a_{hr})$$

where:

1. $\Phi'(a_{hr})$: Derivative of the activation function at a_{hr}.
2. h_{r-1} : Output of the previous layer.

Gradient evaluation

- **1.Bias Gradients:**
  1. Bias gradients are computed by setting $h_{r-1}=1$ in the weight gradient formula.

2.Activation Derivatives:

1. For an edge (h_r,h) with weight $w(h_r,h)$:

$$\frac{\partial h}{\partial h_r}=\Phi'(ah)\cdot w(h_r,h)$$

3.Dynamic Programming Recursion:

- Gradients are recursively computed starting from the output layer and moving backward:
- Initial value: $\Delta(o,o)=\frac{\partial L}{\partial o}$.
- Recursion: Update each $\Delta(h_r,o)$ by summing over all paths connecting $h_r$to ooo.

Key Observations

Error Propagation:

The algorithm backpropagates errors (i.e., $\Delta(h_r,o)$ layer by layer, multiplying them with intermediate variables.

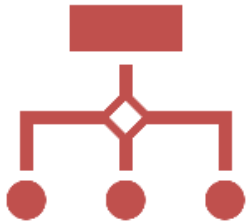
Generality:

The method generalizes easily to multiple outputs by aggregating contributions from all outputs.

Efficiency:

The computational cost is linear in the number of edges in the network.

Summary

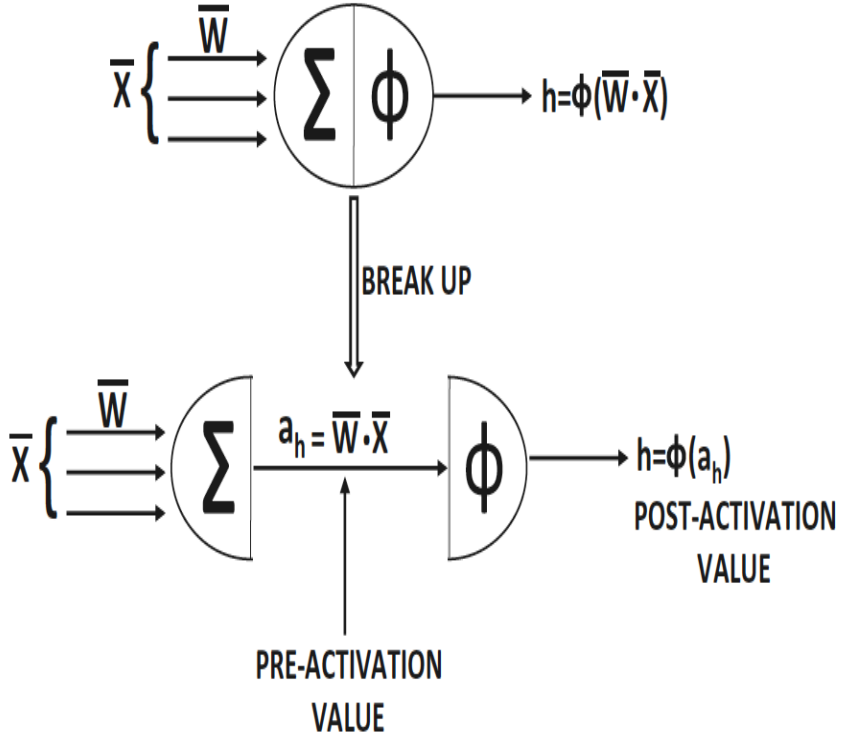


Pre-activation value (a_h):
Computed as a linear combination of inputs.

Post-activation value ($h=\Phi(a_h)$):
Output after applying the activation function.

Pre-activation vs Post-activation

- **Pre-activation values**:
  - Denoted as a_{hr}, these are computed before applying the activation function: $\mathbf{h_r = \Phi(a_{hr})}$
- **Post-activation values**:
  - Denoted as h_r, these are computed after applying the activation function.



Recurrence Relation Using Pre-Activation Values

•• Instead of using post-activation values, we use pre-activation values for backpropagation:

$$\delta(h_r, o) = \frac{\partial L}{\partial a_{hr}}$$

- This helps in writing a recurrence relation that is independent of activation functions.

Chain Rule with Pre-Activation

- The **weight updates** follow:

$$\delta(h_r,o)=\Phi'(a_{hr})\sum_{h:hr\rightarrow h} w(h_r,h)\cdot\delta(h,o)$$

- **$\Phi'(a_{hr})$**: The derivative of the activation function.
- **Summation over all children nodes** in the computational graph.
- **Weight (w)** determines the contribution of each child node.
- **Key Advantage:**
  The activation gradient **$\Phi'(a_{hr})$** is outside the summation, making the calculations easier and more structured.

Steps of Backpropagation Using Pre-Activation Values

- **Forward Pass**: Compute all hidden activations and final output **o**, and loss **L**.
- **Initialize**: Start with the output node by computing **$\delta(o, o) = \partial L/\partial o \cdot \Phi'(a_o)$**.
- **Backward Pass**: Compute **$\delta(h_r, o)$** using recurrence from Equation

$$\delta(h_r,o)=\Phi'(a_{hr})\sum_{h:hr\to h} w(h_r,h)\cdot\delta(h,o)$$

- **Weight Updates**:
  - Compute gradient w.r.t weights:

$$\frac{\partial L}{\partial w(h_{r-1},h_r)} = \delta(h_r,o)\cdot h_{r-1}$$

  - Compute gradient w.r.t biases by setting **$h_{r-1} = 1$**.
- **Perform Stochastic Gradient Descent (SGD)** using the computed gradients.

Advantages of Using Pre-activation Variables

- **Decoupling**:
  - The activation gradient ($\Phi'(a_{hr})$) is separated from the linear transformation, making implementation more efficient.
  - Linear transformations are matrix multiplications.
  - Activation gradients involve elementwise operations.
  - This structure allows better utilization of GPU hardware.
- **Simplicity in Recurrence**:
  - The activation gradient is outside the summation, making it easier to specialize the recurrence for different activation functions.

Updates for Various Activations

$$\delta(h_r,o)=\Phi'(a_{hr})\sum_{h:hr\rightarrow h} w(h_r,h)\cdot\delta(h,o)$$

- The equation provides a framework for computing updates for different activation functions.
- Different activation functions have different derivatives, leading to specific update rules.

Update Rules for Different Activations

(a) Linear Activation

- Update Rule:

$$\delta(h_r, o) = \sum_{h:h_r \Rightarrow h} w(h_r, h)\delta(h, o)$$

(b) Sigmoid Activation

- Sigmoid derivative:

$$\frac{d}{dx}(\sigma(x)) = \sigma(x)(1 - \sigma(x))$$

- Update Rule:

$$\delta(h_r, o) = h_r(1 - h_r) \sum_{h:h_r \Rightarrow h} w(h_r, h)\delta(h, o)$$

(c) Tanh Activation

- Tanh derivative:

$$\frac{d}{dx}(\tanh(x)) = 1 - \tanh^2(x)$$

- Update Rule:

$$\delta(h_r, o) = (1 - h_r^2) \sum_{h:h_r \Rightarrow h} w(h_r, h)\delta(h, o)$$

Update Rules for Different Activations

(d) ReLU Activation

- Case-wise update:

$$\delta(h_r, o) = \begin{cases} \sum_{h:h_r \Rightarrow h} w(h_r, h)\delta(h, o), & \text{if } h_r > 0 \\ 0, & \text{otherwise} \end{cases}$$

(e) Hard Tanh Activation

- Case-wise update:

$$\delta(h_r, o) = \begin{cases} \sum_{h:h_r \Rightarrow h} w(h_r, h)\delta(h, o), & \text{if } -1 < h_r < 1 \\ 0, & \text{otherwise} \end{cases}$$

Special Case: Softmax Activation

- Unlike other activations, **Softmax** considers multiple inputs rather than a single one.
- Softmax function:

$$o_i = \exp(v_i) / \sum_{j=1}^{k} \exp(v_j)$$

- Softmax is mostly used in **output layers** and paired with **cross-entropy loss**.

Softmax with Cross-Entropy Loss

- Cross-entropy loss:

$$L = -\sum_{i=1}^{k} y_i \log(o_i)$$

- Gradient calculation:

$$\frac{\partial L}{\partial v_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial o_j} \cdot \frac{\partial o_j}{\partial v_i}$$

  - **For i = j:**

$$\frac{\partial o_i}{\partial v_i} = o_i(1 - o_i)$$

  - **For i ≠ j:**

$$\frac{\partial o_j}{\partial v_i} = -o_i o_j$$

- Final simplified update:

$$\frac{\partial L}{\partial v_i} = o_i - y_i$$

Decoupling Linear Transformation and Activation

- A neural network can be structured by treating linear computations and activation computations as separate **layers**.
- This allows a more **modular and systematic** approach to forward and backward propagation.

Decoupled View of Vector-Centric Backpropagation

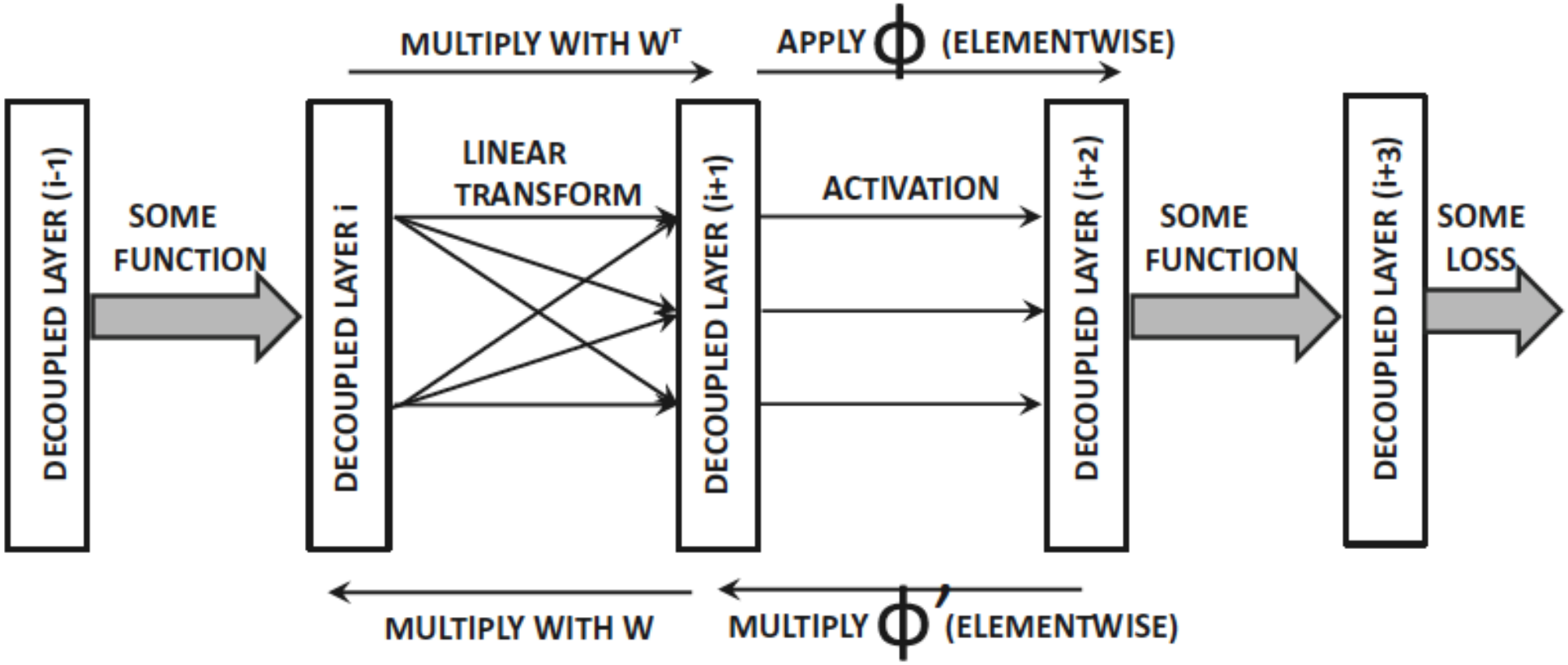


Figure 3.7: A decoupled view of backpropagation

Decoupled View of Vector-Centric Backpropagation

In traditional deep learning architectures, each layer of a neural network is typically seen as a **single unit** that performs both a **linear transformation** (i.e., weighted sum of inputs) and a **nonlinear activation function**.

However, a more modular and systematic approach is to **decouple** these operations into separate layers.

This approach enhances clarity, improves optimization, and makes neural network implementation more efficient.

Several deep learning frameworks (e.g., TensorFlow, PyTorch) follow this principle by defining **separate layers** for linear transformations (fully connected layers) and activation functions.

Comparison of Traditional vs. Decoupled Representation in Neural Networks

Aspect	Traditional Representation (Coupled View)	Decoupled Representation (Layer-wise View)
Forward Propagation	$z_{i+1}=\Phi(W^T z_i+b)$	Linear Layer: $z_{i+1}=W^T z_i+b$ Activation Layer: $z_{i+2}=\Phi(z_{i+1})$
Operations	Linear transformation and activation occur together.	Linear transformation and activation are separate layers.
Modularity	Less modular, harder to analyze and modify.	More modular, allowing independent analysis and modifications.
Flexibility	Changing activation or weight structure requires modifying the entire layer.	Changing activation functions or weight transformations is easier.
Optimization	Mixed gradients make optimization complex.	Better gradient flow and parameter tuning.
Backward Propagation (Gradient Computation)		Through Activation: $g_{i+1}=g_{i+2}\odot\Phi'(z_{i+1})$ Through Linear Transformation: $g_i=Wg_{i+1}$
Gradient Computation Complexity	Activation derivative and weight gradients are entangled in one step.	Gradients flow separately through activation and linear layers.
Implementation in Deep Learning Frameworks	Activation is included inside fully connected layers.	Libraries like TensorFlow and PyTorch define separate Linear and Activation layers.

Forward and Backward Propagation in Decoupled View

Step	Mathematical Expression	Description
Forward Propagation		
Linear Transformation	$z_{i+1} = W^T z_i$	Computes the weighted sum of inputs using the weight matrix W.
Activation Function Application	$z_{i+2} = \Phi(z_{i+1})$	Applies the activation function Φ element-wise to the transformed input.
Backward Propagation		
Through Activation Function	$g_{i+1} = g_{i+2} \odot \Phi'(z_{i+1})$	Computes the gradient of the loss with respect to activations using element-wise multiplication (⊙).
Through Linear Transformation	$g_i = W g_{i+1}$	Propagates the gradient backward through the linear layer using matrix multiplication.

Advantages of Decoupling

Simplicity: Reduces the complexity of backpropagation equations.

Efficient Computation: Easier to optimize using vectorized GPU implementations.

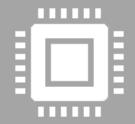
Modular Design: Makes implementing different activation functions and transformations easier.

Different Backpropagation Updates

Function Type	Forward Propagation	Backward Propagation
Linear (Many-to-Many)	$z_{i+1} = W^T z_i$	$g_i = W g_{i+1}$
Sigmoid (One-to-One)	$z_{i+1} = \text{sigmoid}(z_i)$	$g_i = g_{i+1} \odot z_{i+1} \odot (1 - z_{i+1})$
Tanh (One-to-One)	$z_{i+1} = \tanh(z_i)$	$g_i = g_{i+1} \odot (1 - z_{i+1}^2)$
ReLU (One-to-One)	$z_{i+1} = z_i \odot I(z_i > 0)$	$g_i = g_{i+1} \odot I(z_i > 0)$
Max (Many-to-One)	Maximum of inputs	Set non-max inputs to 0, copy max input

Loss Functions on Multiple Output Nodes and Hidden Nodes

The discussion on loss functions in neural networks often assumes a single output node.



However, in real-world applications, loss functions are computed over multiple output nodes.

This scenario influences how gradients are initialized and propagated during backpropagation.

Additionally, in some learning paradigms, hidden nodes also contribute to the loss function, requiring slight modifications in the backpropagation process.

Loss Computation Over Multiple Output Nodes

•• In networks with multiple output nodes, the gradient at each output node **o** is initialized as:

$$\frac{\partial L}{\partial a_o}, = \delta(o, O)$$

where $\delta(o, O)$ represents the gradient flow for each output node o in the output set O.

- Backpropagation then propagates this gradient to hidden nodes h, computing:

$$\frac{\partial L}{\partial a_h} = \delta(h, O)$$

for each hidden node h .

- This ensures that all parameters are updated correctly based on the loss computed at multiple output nodes.

Loss Functions at Hidden Nodes

In certain architectures, such as sparse autoencoders and contractive autoencoders, loss functions are also applied to hidden nodes.

This encourages specific properties in the hidden representations:

Sparse Autoencoder: Encourages sparsity in activations.

Contractive Autoencoder: Uses a penalty term to enforce robustness in representation.

When hidden nodes have associated loss functions, the gradient propagation must account for losses at both hidden and output nodes.

This requires aggregating the gradient flows resulting from different losses.

Gradient Flow with Loss at Hidden Nodes

- Let L_{hr} be the loss associated with hidden node h_r .
- The overall loss function L considers losses from both output and hidden nodes.
- The gradient flow for h_r is given by: $\frac{\partial L}{\partial a_{hr}} = \delta(h_r, N(h_r))$
  - Where $N(h_r)$ is the set of nodes influenced by h_r , including both output and hidden nodes.
- The backpropagation update is performed as:

$$\delta(h_r, N(h_r)) \leftarrow \Phi\,(ah_r) \sum_{h:h_r \Rightarrow h} w(h_r, h)\,(h, N(h))$$

  - Follows the standard backpropagation rule but does not yet incorporate the loss contribution from h_r.
- To account for this, an additional step updates the gradient flow:

$$\delta(h_r, N(h_r)) \leftarrow \delta(h_r, N(h_r)) + \Phi\,(h_r)\,\frac{\partial L_{hr}}{\partial h_r}$$

  - Ensures that the hidden node's loss contributes correctly to the overall gradient computation.

Setup and Initialization Issues

The setup of a neural network involves several key considerations, including hyperparameter selection, feature preprocessing, and initialization.

These aspects play a crucial role in ensuring efficient training and improving model performance.

Neural networks typically have large parameter spaces compared to other machine learning models, which makes preprocessing and initialization more significant.

Tuning Hyperparameters

Hyperparameters (e.g., learning rate, regularization weight) regulate model design and differ from trainable parameters (weights).

Tuning Process: Use a **validation set** to prevent overfitting.

Grid Search: Tests combinations of predefined hyperparameter values but is computationally expensive.

Random Search: Samples values within a range, often in **logarithmic space** for learning rates and regularization terms.

Bayesian Optimization: Efficient but slow for large models.

Early Stopping: Runs are terminated early if performance is poor, optimizing computation time.

Feature Preprocessing

Feature preprocessing ensures stable training and better performance. Methods include:

Mean-centering: Subtracts column-wise means to remove bias.

Feature Normalization:

Standardization: Dividing by standard deviation to achieve zero mean, unit variance.

Min-Max Scaling: Normalizes values between [0,1] to handle varying feature magnitudes.

Whitening: Uses **Principal Component Analysis (PCA)** to decorrelate and scale features. Reduces dimensions and numerical instability by filtering low-variance directions. Affects network architecture by altering input dimensions.

Challenges in Neural Network Optimization

Unstable Backpropagation

Small variations in parameters (e.g., weight initialization) can cause instability.

Deep networks are more prone to instability, making training difficult.

Multivariable Optimization Challenges

Training a neural network involves optimizing a large number of weights.

Standard gradient descent methods must be carefully tuned to ensure convergence.

Gradient Descent Limitations

The gradient provides only a local rate of change, requiring careful step-size selection:

- **Small steps** → Slow convergence.
- **Large steps** → Potential divergence.

Impact of Initialization and Input Normalization

Poor weight initialization and improper feature scaling exacerbate training difficulties.

Proper techniques such as batch normalization help mitigate these issues.

Addressing Optimization Challenges

Modern algorithms tailor gradient-descent steps to be more robust.

Adaptive optimization techniques (e.g., Adam, RMSprop) mitigate some of these issues.

Proper weight initialization and input normalization techniques improve training stability.

Further algorithmic refinements continue to enhance neural network training efficiency.

Common problem in neural network training

- **Steepest Descent Basics:**
  - The most common way to train neural networks is by adjusting the network's parameters (weights and biases) to minimize a "loss function." This loss function measures how well the network is performing.
  - The steepest descent method uses the *gradient* of the loss function to figure out how to change the parameters.
  - The gradient points in the direction of the *steepest ascent* of the loss function, so we move in the *opposite* direction (the *steepest descent*) to try and reduce the loss.
- **The Problem: Not Always the Best Direction:**
  - While the gradient *theoretically* points in the best direction for infinitesimal steps, in practice we take steps of a *finite* size.
  - This means that even if we start moving in the direction of steepest descent, after a small step, we might actually be moving *uphill* (increasing the loss) instead of downhill.
- **Oscillations and Zigzagging:**
  - This can lead to oscillations, where the training process zigzags back and forth, making slow progress. Imagine trying to walk down a mountain but constantly veering off course and having to correct yourself.

Common problem in neural network training

- **High Curvature and Ill-Conditioning:**
  - The problem is especially bad when the loss function has high curvature.
  - Think of a very uneven, bumpy surface. In such cases, the steepest descent direction can change dramatically with even small changes in position.
  - This is also related to *ill-conditioning*, where the partial derivatives of the loss function with respect to different parameters vary wildly in magnitude.
  - This means some parameters might be very sensitive to changes while others are much less so, making it hard to find a good learning rate that works well for all parameters.
- **The Need for Better Strategies:**
  - The excerpt concludes by stating that it will discuss more sophisticated learning strategies designed to address these problems, especially in ill-conditioned scenarios. These strategies are necessary to make training more efficient and robust.

Gradient Decent

- Gradient descent is an iterative optimization algorithm to find the minimum of a function. Here that function is our Loss Function.

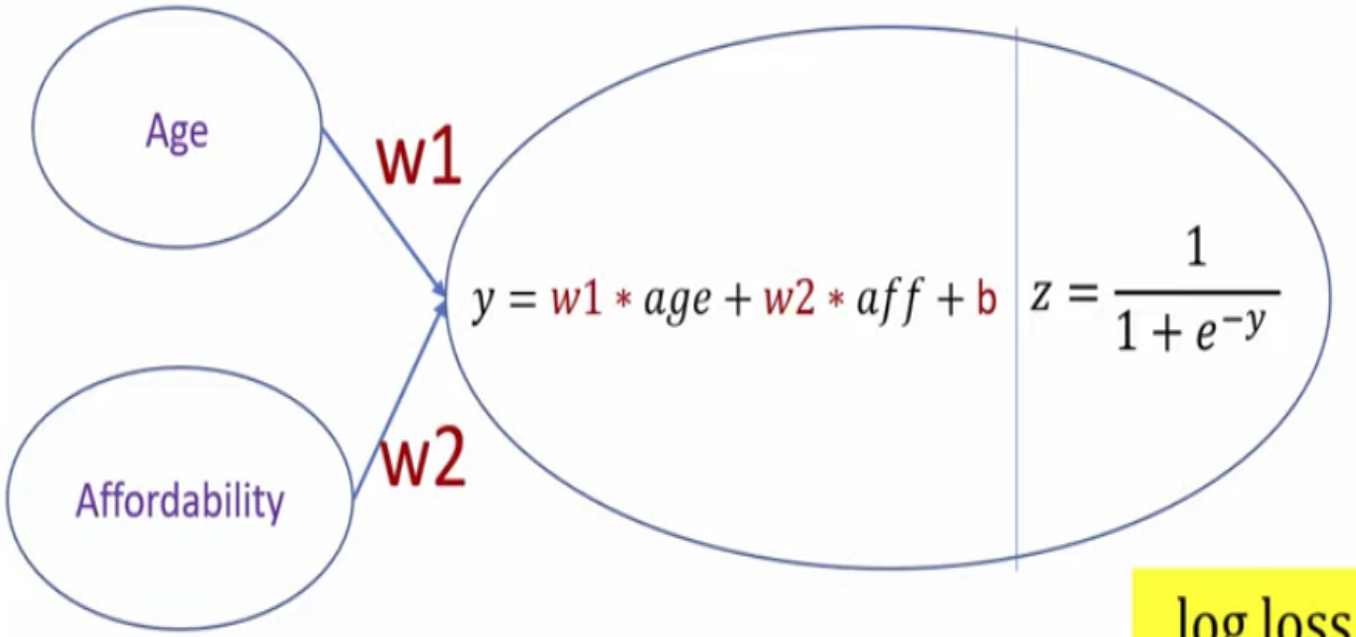
x	y
2	-0.5
3	0
5	1
7	2
9	3

Technique called Gradient Descent can help you find these parameters

$$y = x * 0.5 - 1.5$$

After **first epoch**

age	affordability	have_insurance
22	1	0
25	0	0
47	1	1
52	0	0
46	1	1
56	1	1
55	0	0
60	0	1
62	1	1
61	1	1
18	1	0
28	1	0
27	0	0



log loss = 4.31

Adjusting weight for 2nd epoch

$$w1 = w1 - learning\ rate * {}^{\partial}/_{\partial w1}$$

$$w2 = w2 - learning\ rate * {}^{\partial}/_{\partial w2}$$

$$b = b - learning\ rate * {}^{\partial}/_{\partial b}$$

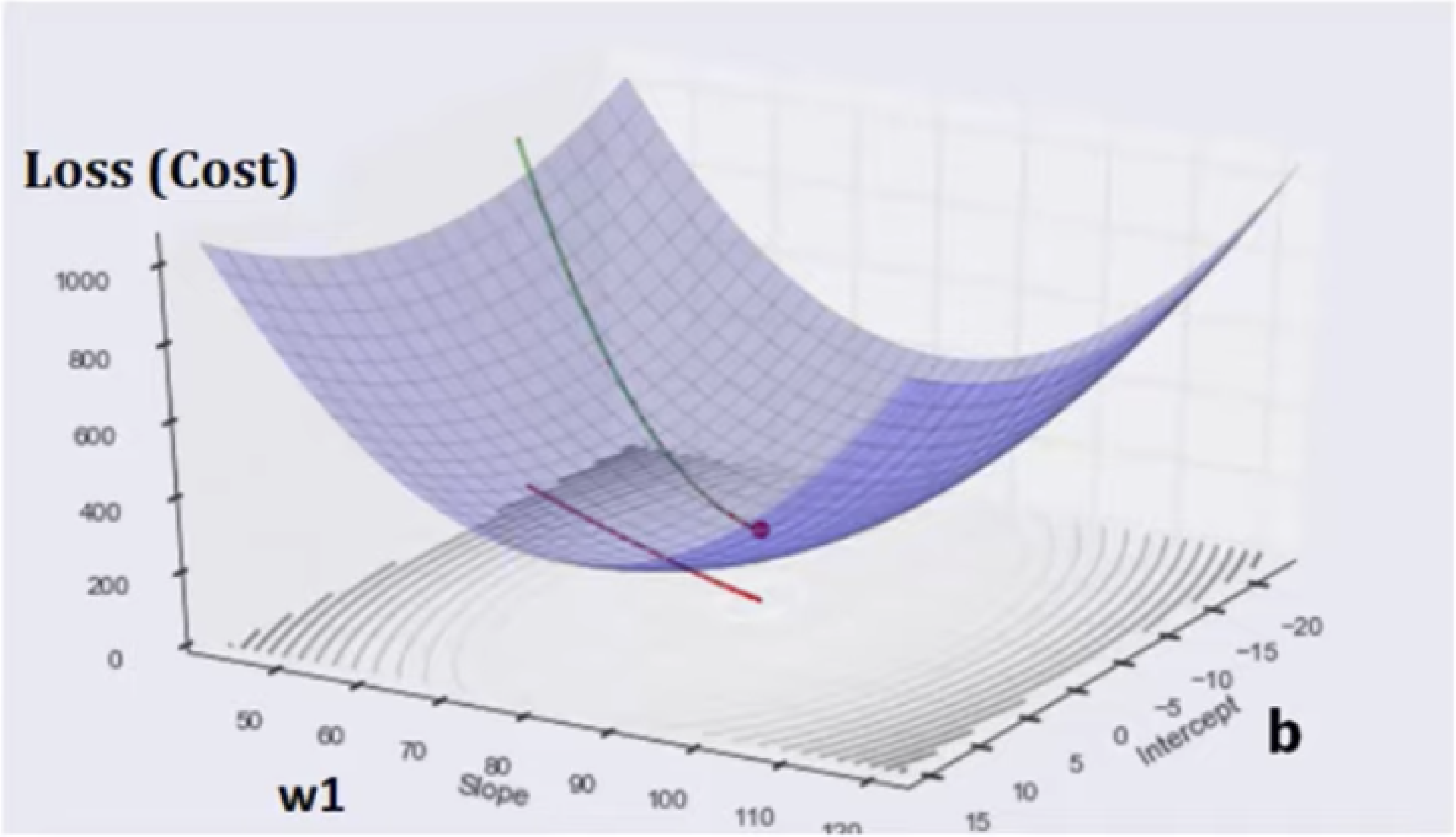
Weight and biases derivative

$$w1 = w1 - learning\ rate * {}^{\partial}/_{\partial w1}$$

$${}^{\partial}/_{\partial w1} = \frac{1}{n}\sum_{i=1}^{n} x_i\,(\hat{y}_i - y_i)$$

$$b = b - learning\ rate * {}^{\partial}/_{\partial b}$$

$${}^{\partial}/_{\partial b} = \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)$$


Loss (Cost)
1000
800
600
400
200
0
50
60
70
80
90
100
110
w1
Slope
15
10
5
0
−5
−10
−15
−20
Intercept
b

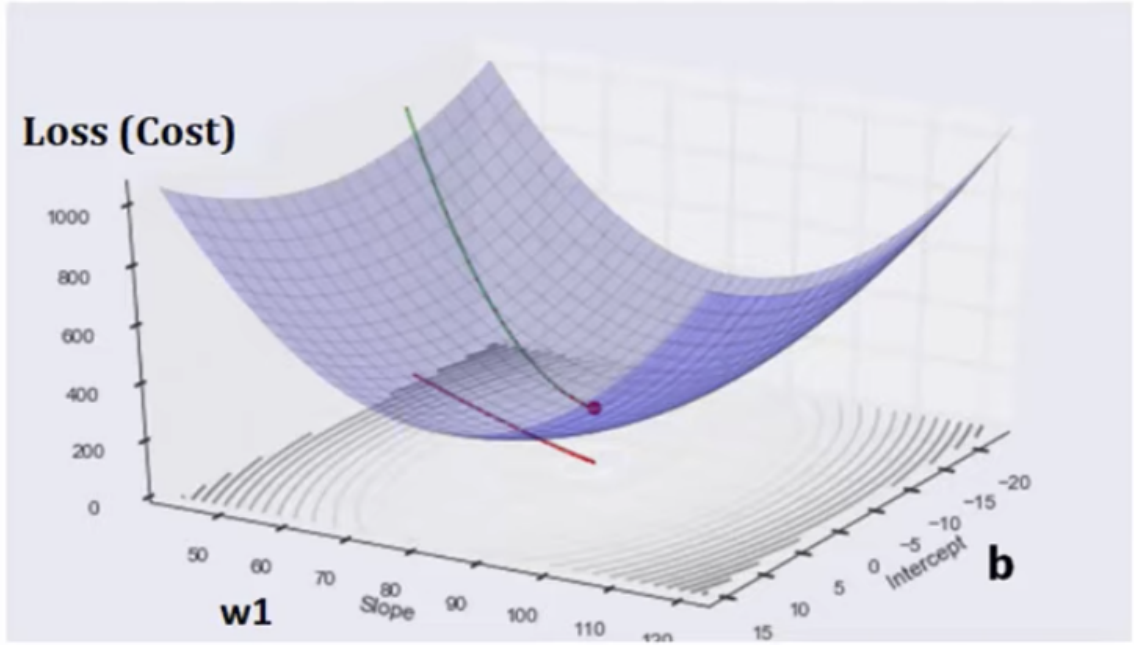
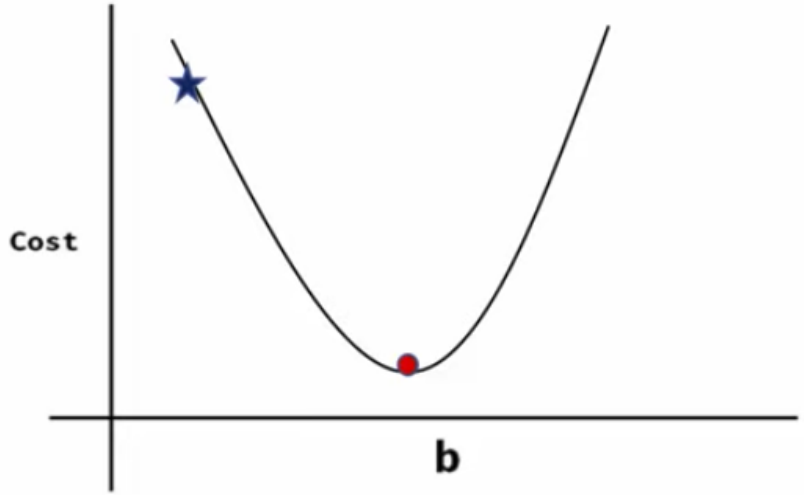
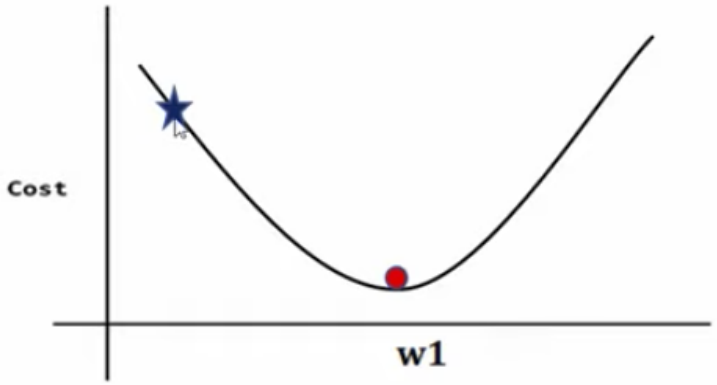
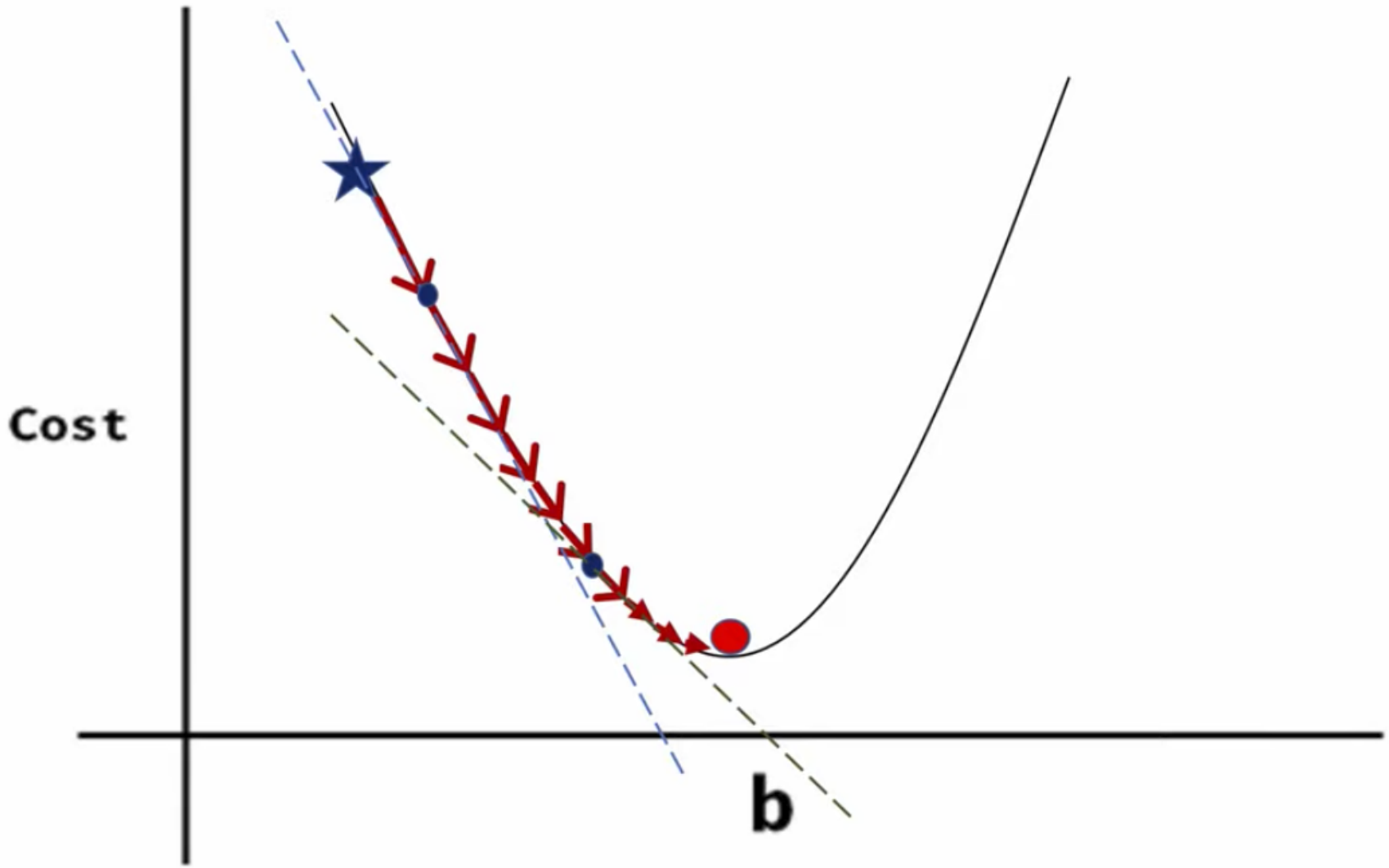


Image credit: https://am207.github.io/2017/wiki/gradientdescent.html






Cost
b

Example

Imagine a valley and a person with no sense of direction who wants to get to the bottom of the valley.

He goes down the slope and takes large steps when the slope is steep and small steps when the slope is less steep.

He decides his next position based on his current position and stops when he gets to the bottom of the valley which was his goal.


steep slope
Value of D is high
So take large steps
slope is less steep
Value of D is low
So take small steps
Goal

Learning Rate Decay

In neural network training, the learning rate is a crucial hyperparameter that controls the step size taken during optimization.

A constant learning rate can lead to training challenges, motivating the use of *learning rate decay*.

The Learning Rate Dilemma:

Small Learning Rate:

- Slow convergence, especially in early training stages.
- The model might take a very long time to even approach a reasonable solution.
- *Example:* Imagine trying to find the bottom of a valley by taking tiny steps. You'll eventually get there, but it will take forever.

Large Learning Rate:

- Fast initial progress, but the model might oscillate around the optimal solution or even diverge.
- The model overshoots the minimum and bounces back and forth without settling.
- *Example:* Imagine trying to find the bottom of the valley by taking huge leaps. You might jump over the valley entirely or keep bouncing between the slopes without ever reaching the bottom.

Why Learning Rate Decay?

Learning rate decay addresses this dilemma by starting with a larger learning rate for rapid initial progress and then gradually decreasing it to enable finer adjustments and convergence.

This allows the model to quickly explore the parameter space and then settle into a more precise minimum.

Common Decay Methods

Decay Method	Formula/Description	Advantages	Disadvantages
Exponential Decay	$\alpha t = \alpha 0 * \exp(-k * t)$	Smooth decay	Requires tuning of decay constant k
Inverse Decay	$\alpha t = \alpha 0 / (1 + k * t)$	Simple to implement	Decay rate might be too aggressive initially
Step Decay	Reduce by a factor every N epochs	Easy to implement, clear control	Can be jerky, requires careful interval choice
Validation Loss	Reduce when validation loss plateaus	Adaptive, based on actual performance	Requires validation set
Manual Adjustment	Manually change during training	Direct control	Time-consuming, subjective

Exponential Decay:

- Formula:

$$\alpha_t = \alpha_0 * \exp(-k * t)$$

- α_t: Learning rate at epoch t
- α_0: Initial learning rate
- k: Decay constant (controls the rate of decay)
- t: Epoch number
- *Example:*
  - Let $\alpha 0 = 0.1$, $k = 0.01$.
  - After 10 epochs,

$$\alpha 10 = 0.1 * \exp(-0.01 * 10) \approx 0.09.$$

Inverse Decay:

- Formula:

$$\alpha t = \alpha 0 / (1 + k * t)$$

- *Example:*
- Let $\alpha 0 = 0.1$, $k = 0.01$.
- After 10 epochs,

$$\alpha 10 = 0.1 / (1 + 0.01 * 10) \approx 0.091.$$

Step Decay:

- Reduce the learning rate by a fixed factor at specific intervals.
- *Example:*
- Multiply the learning rate by 0.5 every 5 epochs.
- If $\alpha 0 = 0.1$, then after 5 epochs, $\alpha 5 = 0.05$; after 10 epochs, $\alpha 10 = 0.025$, and so on.

Monitoring Validation Loss:

Track the loss on a validation set (a portion of the training data not used for training).

Reduce the learning rate when the validation loss plateaus or stops improving.

This is a common and effective technique because it adapts the learning rate based on actual performance.

Example: If the validation loss doesn't decrease for 3 consecutive epochs, reduce the learning rate by a factor of 0.1.

Manual Adjustment:

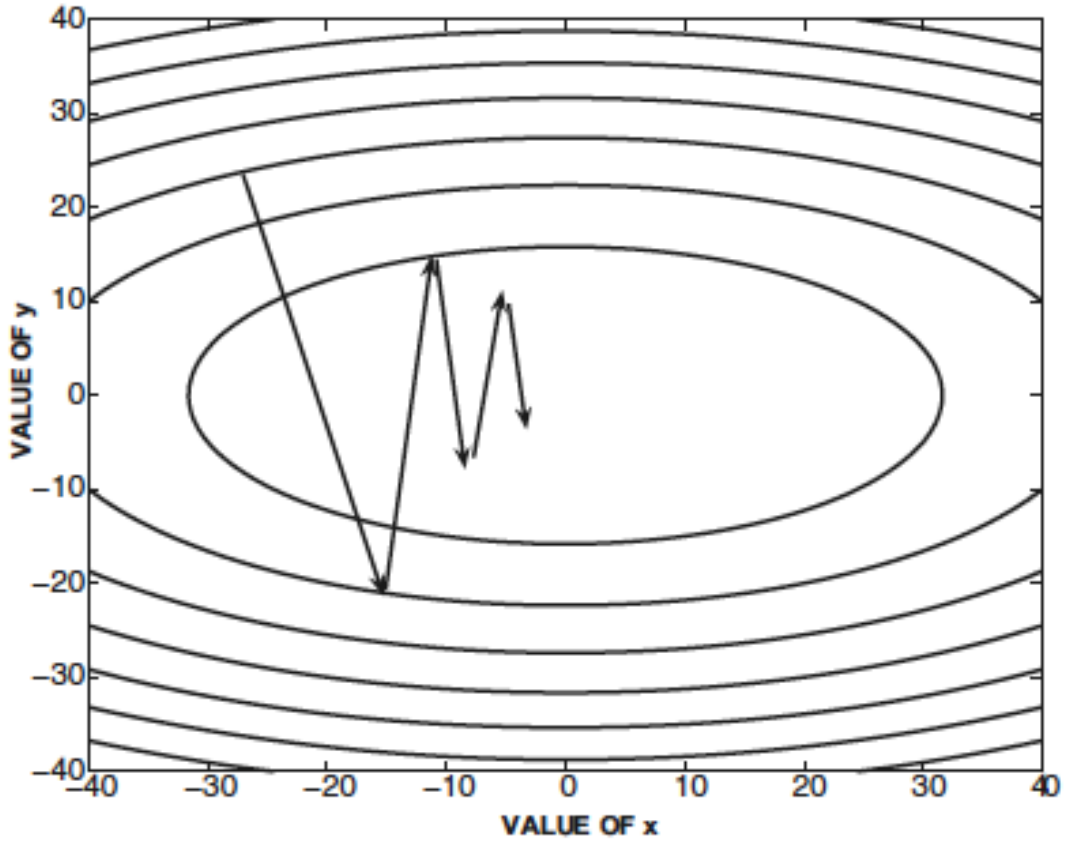
Manually adjust the learning rate during training based on observed progress.

Less common with advanced optimization algorithms.

Example: If you notice the training is oscillating, you might manually decrease the learning rate.

Momentum-Based Optimization Techniques

- **Problem of Zigzagging in Gradient Descent**
  - In gradient descent, updates can oscillate or zigzag, especially in loss surfaces with high curvature or steep slopes.
  - Contradictory gradient steps cancel each other out, reducing the effective movement in the correct (long-term) direction.
- **Example**: Figure illustrates how zigzagging slows convergence.



(b) Loss function is elliptical bowl

$$L = x^2 + 4y^2$$

Intuition Behind Momentum

- Instead of taking steps based solely on the current gradient, use an "averaged" direction of the last few steps to smooth out oscillations.
- This helps maintain consistent movement in the correct direction, even if individual steps are contradictory.

Mathematical Formulation

- **Standard Gradient Descent Update:**

$$V \leftarrow -\alpha \frac{\partial L}{\partial W} \;, W \leftarrow W+V$$

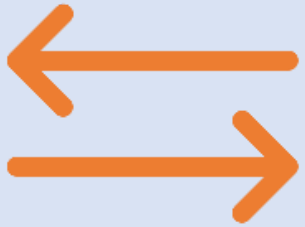
  - α: Learning rate.
  - $\frac{\partial L}{\partial W}$: Gradient of the loss function L with respect to parameters W.

- **Momentum-Based Update:**

$$V \leftarrow \beta V - \alpha \frac{\partial L}{\partial W} \;, W \leftarrow W+V$$

  - β : Momentum parameter (smoothing factor), typically $\beta \in (0,1)$
  - V: Velocity vector, which accumulates past gradients.

Role of the Momentum Parameter (β)



Larger β:

Picks up a consistent velocity in the correct direction.

Smoothens oscillations more effectively.

Smaller β:

Acts like "friction," slowing down the velocity.

Resembles standard gradient descent when β=0β=0.

Benefits of Momentum

Accelerated Learning:

Moves faster in directions that consistently point toward the optimal solution.

Reduces oscillations in irrelevant directions.

Improved Navigation.

Helps escape flat regions of the loss surface (fig a.)

Avoids getting stuck in local optima.

Efficiency:

Reaches the optimal solution in fewer updates. Fig b

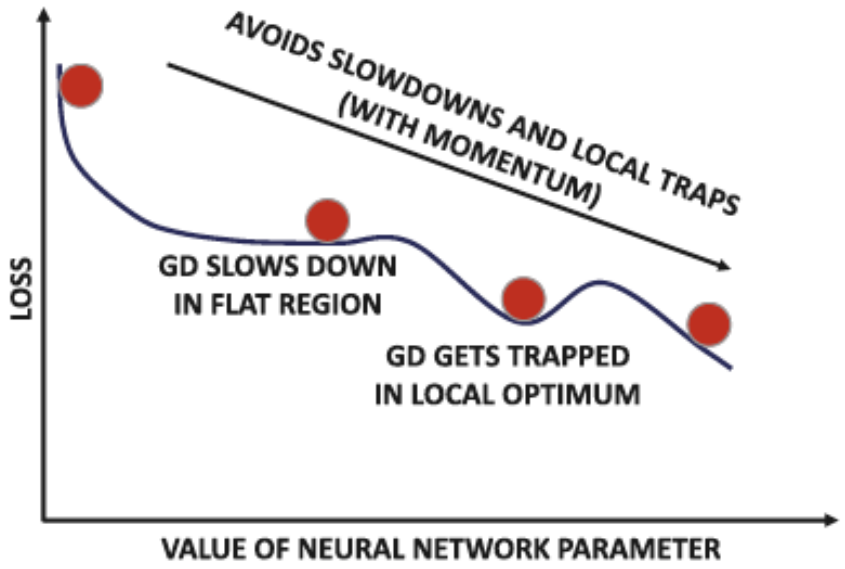


Fig a : Effect of momentum in navigating complex loss surfaces. The annotation “GD”
indicates pure gradient descent without momentum. Momentum helps the optimization
process retain speed in flat regions of the loss surface and avoid local optima.

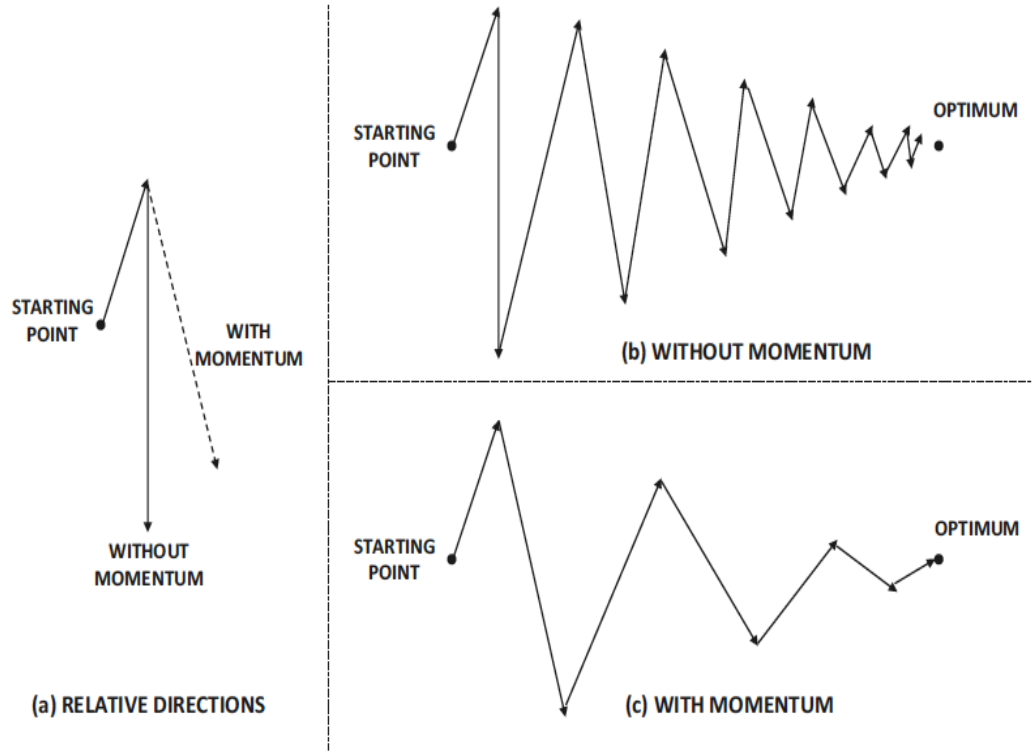


Fig B Effect of momentum in smoothing zigzag updates

Practical Interpretation

Velocity (V):

Represents the accumulated direction of past gradients.

Exponential Smoothing:

Recent gradients have more influence, while older gradients are gradually forgotten.

Trade-off:

High β: Faster convergence but risk of overshooting.

Low β: Slower convergence but more stable updates.

Nesterov Momentum

Nesterov Momentum is an improvement over the standard momentum method in gradient descent.

The key idea is to compute the gradient not at the current position W, but at an anticipated future position $W+\beta V$, where β is the momentum coefficient and V is the velocity term from the previous update.

Mathematical Formulation:

•• Compute the velocity update:

$$V \leftarrow \beta\, V - \alpha \frac{\partial L\,(W+\beta\, V)}{\partial W}$$

- Here, the gradient is computed at W+βV, rather than WWW, allowing the update to incorporate information about the expected movement.
- Update the parameter:

$$W \leftarrow W+V$$

Key Benefits:

Better Lookahead:

- By computing the gradient at a future position, Nesterov momentum anticipates changes in direction and adjusts the update accordingly.

Faster Convergence:

- It reduces error to $O(1/t^2)$ .compared to $O(1/t)$ in standard momentum-based methods.

Stability Near Minima:

- Helps avoid overshooting and allows smoother convergence.

Limitations

Works best with mini-batch gradient descent with moderate batch sizes.

Using very small batch sizes can make it ineffective due to high variance in gradients.

Parameter-Specific Learning Rates

In momentum-based optimization methods, the primary aim is to **utilize the consistency of the gradient direction** to accelerate parameter updates.

An alternative approach is to assign **different learning rates to different parameters** based on their gradient behavior.

This method helps in addressing the issue of **zigzagging** in optimization.

Methods

Gradient Behavior and Learning Rate Adjustment:

- **Large Partial Derivatives:** Often lead to oscillations and zigzagging.
- **Small Partial Derivatives:** Indicate more stable, consistent movement in the same direction.
- By adjusting the learning rate for each parameter individually, optimization can be improved.

Delta-Bar-Delta Method:

- One of the earliest methods for adjusting parameter-specific learning rates.
- Tracks whether the **sign of a parameter's gradient** remains the same or changes:
  - **If the sign remains consistent:** The learning rate increases.
  - **If the sign frequently flips:** The learning rate decreases.
- This method is **suitable for batch gradient descent** but not for **stochastic gradient descent (SGD)**, as the noisy updates in SGD can lead to instability.

Challenges and Improvements for Mini-Batch Methods:

- Traditional parameter-specific learning rate methods struggle with **stochastic noise** in SGD.
- Several improved methods have been introduced to ensure **stability and effectiveness** even when mini-batches are used.

Consider optimizing a deep neural network

Some weights might have large, fluctuating gradients due to complex features →

they need a **lower learning rate** to prevent instability.

Other weights may have small, steady gradients →

they can benefit from a **higher learning rate** to speed up convergence.

Modern Adaptive Learning Rate Methods

- To address the limitations of traditional parameter-specific learning rate methods like **Delta-Bar-Delta**, modern optimization techniques adapt learning rates dynamically for each parameter.
- Some widely used methods include:
  - AdaGrad (Adaptive Gradient Algorithm)
  - RMSprop (Root Mean Square Propagation)
  - Adam (Adaptive Moment Estimation)

AdaGrad (Adaptive Gradient Algorithm)

AdaGrad is an adaptive learning rate optimization algorithm

Modifies the learning rate for each parameter individually based on the accumulated squared gradients.

Parameters that have **larger gradients** (i.e., change rapidly) should have **smaller learning rates**

While parameters with **smaller gradients** (i.e., change consistently in one direction) should have **larger learning rates**.

This ensures that the model **adapts to different scales of parameters**,

Making it particularly useful for problems where gradients vary significantly.

Mathematical Formulation

- AdaGrad modifies the standard gradient descent update rule as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

- where:
- θ_t = current parameter value at time step t
- η = initial learning rate (fixed hyperparameter)
- g_t = gradient of the loss function with respect to θ_t
- $Gt = \sum_{i=1}^{t} g_i^2$ = accumulated squared gradients
- ϵ = small constant (e.g., 10^{-8}) to avoid division by zero

Accumulated Squared Gradient:

$$G_t = G_{t-1} + g_t^2$$

- This accumulation causes the denominator to **continuously increase over time**.
- As a result, the learning rate for frequently updated parameters **shrinks over time**
- While rarely updated parameters retain **larger learning rates**.

How AdaGrad Works

- **Tracks past gradients:**
  - Instead of using a fixed learning rate for all parameters, AdaGrad keeps track of the squared gradients of each parameter.
- **Adapts learning rates:**
  - The learning rate for each parameter is divided by the square root of the sum of its past squared gradients.
- **Effectively slows down updates:**
  - This makes large-gradient parameters move **less aggressively** and small-gradient parameters move **more actively**.
- **Useful for sparse data:**
  - In problems like **natural language processing (NLP)** and **recommendation systems**, some features occur frequently while others are rare.
  - AdaGrad automatically **assigns larger learning rates** to less frequent features, making training more efficient.

Advantages and Disadvantage of Adagrad

Advantages

- **Adaptive Learning Rate:** Each parameter gets an individualized learning rate, preventing overshooting in high-gradient areas.
- **Effective for Sparse Data:** Works exceptionally well for NLP and recommendation systems, where some features appear more frequently than others.
- **No Manual Learning Rate Scheduling:** Unlike standard gradient descent, where you need to decay the learning rate manually, AdaGrad does this **automatically** based on gradient history.

Limitations

- **Monotonically Decreasing Learning Rate:**
  - Since G_t **keeps accumulating**, the learning rate becomes **too small over time**, causing updates to slow down drastically.
  - This can lead to premature convergence or getting stuck before reaching the optimal solution.
- **Not Ideal for Deep Learning:**
  - In deep neural networks, the diminishing learning rate prevents the model from effectively escaping saddle points or local minima.
- As a result, **RMSprop and Adam (which modify AdaGrad's accumulation method) are preferred for deep learning**.

RMSprop (Root Mean Square Propagation)

- A modification of AdaGrad that **introduces exponential decay** in the sum of past squared gradients to prevent aggressive decrease in learning rate.
- **Mathematical Formula:**

$$v_t = \beta v_{t-1} + (1-\beta) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot g_t$$

- where:
  - v_t = exponentially weighted moving average of squared gradients
  - β= decay factor (typically 0.9)

Pros and Cons

Pros:	Cons:
• Prevents learning rate from becoming too small over time. • Works well in non-convex problems like deep learning.	• Sensitive to hyperparameter β.

Adam (Adaptive Moment Estimation)

- Combines **momentum (first moment)** and **RMSprop (second moment)** to adjust learning rates adaptively.
- **Mathematical Formula:**

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta 2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot \hat{m}_t$$

where:

- m_t = first moment (momentum-like term)
- v_t= second moment (RMSprop-like term)
- β_1= decay rate for first moment (default: 0.9)
- β_2= decay rate for second moment (default: 0.999)

Pros and Cons

Pros:	Cons:
• Combines best aspects of momentum and adaptive learning rates. • Works well across a wide range of deep learning problems. • Less sensitive to learning rate tuning.	• Can **over-adapt** in some cases, leading to suboptimal solutions. • May not always generalize well to all datasets.

Deep Learning frameworks

Feature	TensorFlow	PyTorch
Developer	Google	Meta (formerly Facebook)
Primary Language	Python, C++	Python, C++
Ease of Use	More complex, requires defining computational graphs	More intuitive, dynamic computational graphs
Computational Graph	Static (TF 1.x), Dynamic (TF 2.x with eager execution)	Dynamic (computation defined at runtime)
Debugging	More difficult due to static graphs (TF 1.x); easier in TF 2.x	Easier due to Pythonic and dynamic nature
Deployment	Strong support (TensorFlow Serving, TensorFlow Lite, TensorFlow.js)	Less mature but improving (TorchServe, ONNX, mobile support)
Performance	Optimized for production, supports XLA for acceleration	Strong performance, but lacks built-in XLA optimization
GPU Acceleration	Native support via CUDA and TensorRT	Native support via CUDA, CuDNN

Deep Learning frameworks

Feature	TensorFlow	PyTorch
TPU Support	Full support (Google Cloud TPUs)	Limited TPU support via XLA
Production Readiness	Highly optimized for large-scale deployment	More research-focused, but gaining production capabilities
Model Training	Uses Keras (high-level API) for ease of use	Offers more flexibility for research experiments
Ecosystem	Extensive (TensorFlow Extended, TensorFlow Hub)	Growing ecosystem but smaller than TensorFlow
Community & Adoption	Widely adopted in industry and large-scale applications	Popular in academia and research labs
ONNX Support	Supports ONNX for interoperability	Stronger ONNX integration for cross-framework usage
Mobile & Edge AI	TensorFlow Lite for mobile deployment	PyTorch Mobile (still developing)
Best For	Production AI, Large-scale models, Mobile AI	Research, experimentation, fast prototyping