Introduction To Deep Learning & Neural Networks with keras (M2) (1)

Created	@December 10, 2025 11:43 AM
Module Code	IBMM02: Introduction to Deep Learning

Module 2: Basics of DL

Gradient Descent

1. Goal: Find the Best Weight (w)

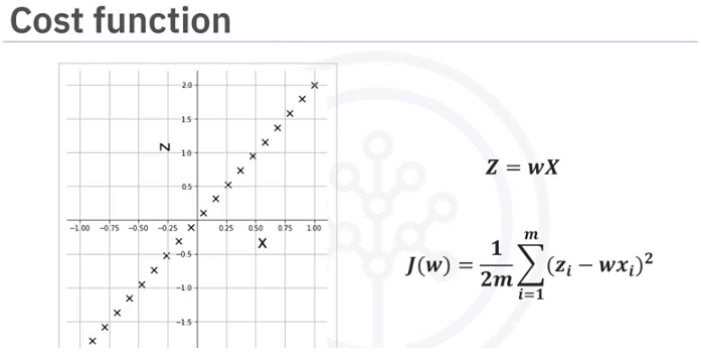
We have data where **z ≈ 2x** and we want to find the value of **w** in the equation:

$\hat{z} = w \cdot x$

To measure how “bad” a weight is, we use a **cost function**:

This is a **parabola**, and the lowest point (minimum) gives the **best weight**.

For this dataset → minimum occurs at **w = 2**.



✨ 2. Why Not Just Plot It?

When we have many weights (like in neural networks), we **cannot visualize the cost function**, so we need an algorithm to find the minimum.

That algorithm is **gradient descent**.

Gradient descent is an iterative optimization algorithm for finding the minimum of a function.

A cost function tells you **how wrong your model currently is.**

If predictions are bad → cost is high.
If predictions are good → cost is low.

Example for simple linear regression:

$$J(w) = \sum (z - wx)^2$$

This means:

- Take actual value **z**
- Take predicted value **wx**
- Find difference (error)
- Square it (to make it positive)
- Add up errors for all points

Goal: find the value of **w** that makes J(w) as small as possible.

Visualize J(w) like a bowl (a parabola):

- The lowest point = where J(w) is minimum
- That w = BEST weight

⚙️ 3. What Gradient Descent Does

Gradient descent updates the weight repeatedly:

- **η (eta)** = learning rate
- **gradient** = slope of the cost curve at current w
- You always move **opposite the gradient** to go downhill.

The gradient tells you:

- Which direction increases the cost the fastest.

So to go DOWN (minimize cost):

👉 **Move in the opposite direction of the gradient.**

---

4. Intuition

- If the slope is **positive**, move left.
- If the slope is **negative**, move right.
- When slope = 0 → minimum → stop.

Because:

- Slope tells you the **direction** to move
- Learning rate tells you the **size** of the step
- Repeating step-by-step gradually moves you to the lowest point of the cost function

Even complex neural networks rely on this same idea.

---

5. Iterations Example

Starting at **w = 0** with learning rate **0.4**:

- **Iteration 1:** Big step because slope is very steep → weight moves close to 2
- **Iteration 2–4:** Smaller steps as slope becomes shallower → weight converges toward 2

Each iteration:

- reduces the cost
- improves the line fitting z = 2x

---

6. Learning Rate Matters

- **Too large:** jumps over the minimum → may diverge
- **Too small:** tiny steps → extremely slow convergence

Choosing η is critical.

Learning Rate

If η is too small:

You take baby steps → very slow learning

If η is too big:

You jump too far → overshoot the minimum → model diverges

Just right:

Steady, efficient descent to the minimum

⭐ Key Takeaways

- Gradient descent finds the weight values that minimize the loss.
- It moves in small steps in the direction opposite the gradient.
- Learning rate determines how big each step is.
- The process repeats until the weight reaches the minimum.

Backpropogation

A neural network performs two things:

1. Forward Propagation → make a prediction

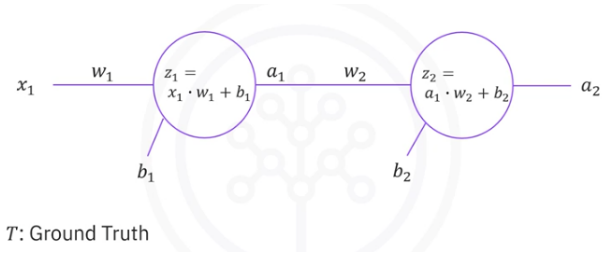
Input → weights → neuron → activation → next layer → output.

2. Backward Propagation → learn from the error

Output → compute error → flow it backward → adjust weights so next time error is smaller.

This loop continues for many epochs until the network is good.

Forward Propagation (The Prediction Phase)



For one neuron:

1. Compute weighted sum:

$$z = wx + b$$

2. Pass through activation:

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

This **a** is the output of the neuron.

In your example network:

- first neuron computes a_1
- second neuron computes a_2

This a_2 is the **prediction**.

Error / Loss Function: How Wrong Are We?

If ground truth (label) = **T**

and prediction = **a_2**

Then error:

This is the value we want to *minimize* by changing weights and biases.

Why We Need Backpropagation

Weights affect activations

Activations affect predictions

Predictions affect error

So to reduce error, we need:

But E is **not directly** a function of w.

It is a function through multiple layers.

This is where **the chain rule** becomes our hero.

The Chain Rule (What Backprop Actually Is)

The structure:

Weight w_2 → affects z_2 → affects a_2 → affects E

So:

We want:

$$\frac{\partial E}{\partial w_2}$$

Given:

- $E = (T - a_2)^2$
- $a_2 = \sigma(z_2)$
- $z_2 = w_2 a_1 + b_2$

This gives the gradient.

Then gradient descent updates weight:

$$w_2 = w_2 - \eta \frac{\partial E}{\partial w_2}$$

where η = learning rate.

Let's Break Down Each Term (Slow and Clear)

⭐ Step 1:

(they drop constants — that's okay.)

⭐ Step 2:

Activation derivative:

⭐ Step 3:

Weighted sum:

⭐ Combine all three:

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

Substitute:

$$\frac{\partial E}{\partial w_2} = [-(T - a_2)] \cdot [a_2(1 - a_2)] \cdot [a_1]$$

That's how **every weight** learns.

🌟 Updating the Bias b_2

Same idea, but:

$$\frac{\partial z_2}{\partial b_2} = 1$$

So:

$$\boxed{\frac{\partial E}{\partial b_2} = -(T - a_2)\, a_2(1 - a_2)}$$

Why Updating w_1 Is More Complicated

w_1 affects:

$w_1 \rightarrow z_1 \rightarrow a_1 \rightarrow z_2 \rightarrow a_2 \rightarrow E$

So chain rule gets longer:

$w_1 \rightarrow z_1 \rightarrow a_1 \rightarrow z_2 \rightarrow a_2 \rightarrow E$

So chain rule gets longer:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

One Full Training Loop

1. Forward pass

Compute:

- z_1, a_1
- z_2, a_2 (prediction)

2. Compute error

3. Backpropagate

Compute gradients for:

- w_2, b_2
- w_1, b_1

4. Update

5. Repeat

Until:

- many epochs completed OR
- error becomes small

That’s backpropagation.

Vanishing Gradient

---

1. What is happening inside a neural network?

Every neuron does:

$a=\sigma(z)$

Sigmoid output range:

$0<a<1$

This means **every activation is a small number**.

---

2. What happens during backpropagation?

The gradient for a weight often looks like:

$$\frac{\partial E}{\partial w} = (\text{some error}) \times a \times (1 - a) \times (\text{previous activation})$$

Here's the key:

- a is between 0 and 1
- 1−a is also between 0 and 1
- Multiply them → even **smaller**
- Multiply again for earlier layers → even **smaller**

You keep multiplying numbers like:

0.8 × 0.4 × 0.3 × 0.2 × 0.1 ...

This **shrinks extremely fast**.

---

3. Why does it shrink?

Because sigmoid derivative:

Maximum value is **0.25**.

Meaning:

Every time you move one layer backward, you multiply by **≤ 0.25**.

So gradients become:

- Output layer: small
- Previous layer: smaller
- Earlier layer: **tiny**
- First layer: **almost zero**

This is called the **vanishing gradient**.

---

4. What does this cause?

❌ Earlier layers barely learn

Because the gradient reaching them is almost zero.

❌ Training becomes extremely slow

Network struggles to adjust early features.

❌ Accuracy becomes bad

Model cannot learn deep hierarchical patterns.

❌ Deep networks become useless with sigmoid

---

6. Why weights near the input (like W1) get the smallest gradient?

Because W1 is **furthest** from the output.

Gradient must pass through:

- output layer derivative
- hidden layer derivative
- previous hidden layer derivative
- etc.

Each one multiplies by a number < 1.

So by the time the gradient reaches W1:

$\partial W1 \partial E \approx 0$

Thus → W1 hardly updates.

---

7. What's the solution?

Use activation functions whose derivatives:

- Do NOT shrink too much
- Do NOT squash everything between 0 and 1

Example:

ReLU :

ReLU′(z)=1 for z>0

This keeps gradient stable and does **NOT vanish**.

This is why deep learning **boomed after ReLU**.

Activation Function

Neural networks learn by adjusting weights using gradients.

But **without activation functions**, the entire network becomes:

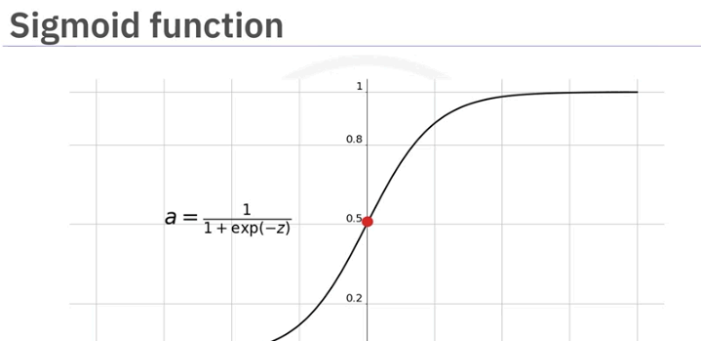
- linear
- useless
- unable to learn complex patterns

So activation functions make the network **non-linear**, allowing it to learn curves, edges, patterns, decisions, boundaries — EVERYTHING.

Now let's break each activation function down quickly but deeply.

---

1. Sigmoid (σ)



Range: 0 to 1

Shape: S-curve

Used earlier historically, rarely used now

✔️ Pros

- Smooth

- Gives probability-like output (0–1)
- Good for **binary classification output layer**

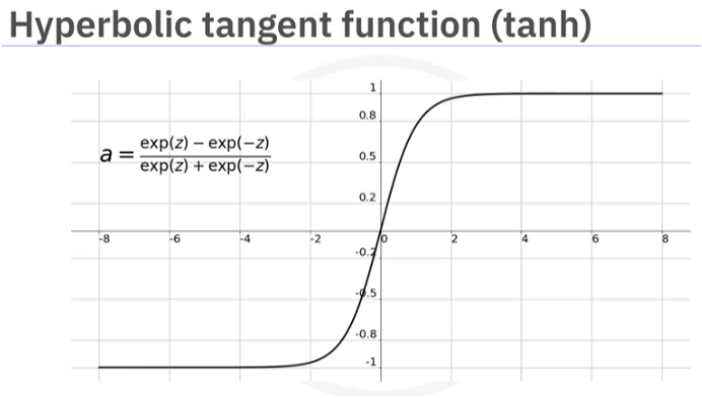
❌ Cons

- **Vanishing Gradient** (flattens after +3 and −3)
- All outputs are **positive** → not symmetric → slows training
- Not used in hidden layers today

Deep networks avoid sigmoid in hidden layers.

---

2. Tanh (hyperbolic tangent)



Range: −1 to +1

Shape: S-curve centered at 0

Better than sigmoid because it's symmetric.

✔️ Pros

- Zero-centered output → easier training
- Still smooth

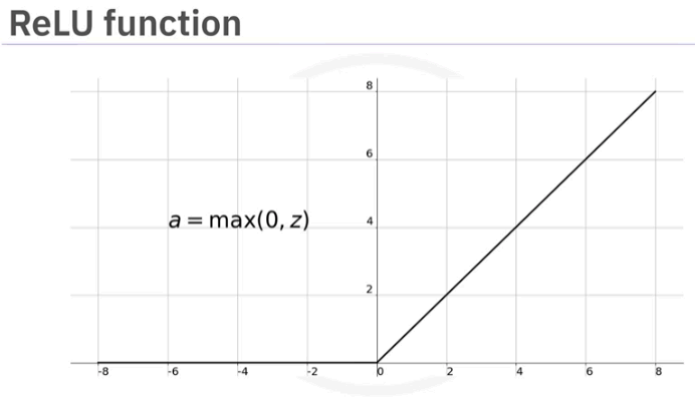
❌ Cons

- **Still suffers from vanishing gradients**
- Not preferred for deep networks

Still not ideal for deep layers.

---

3. ReLU (Rectified Linear Unit)



Range: 0 to ∞

✔️ Pros

- **No vanishing gradient for positive values**
- Super fast
- Sparse activation → efficient
- Simple
- Made deep learning explode in 2012

❌ Cons

- Neurons can "die" if z < 0 always
- But usually not a big issue

BEST CHOICE for hidden layers.

---

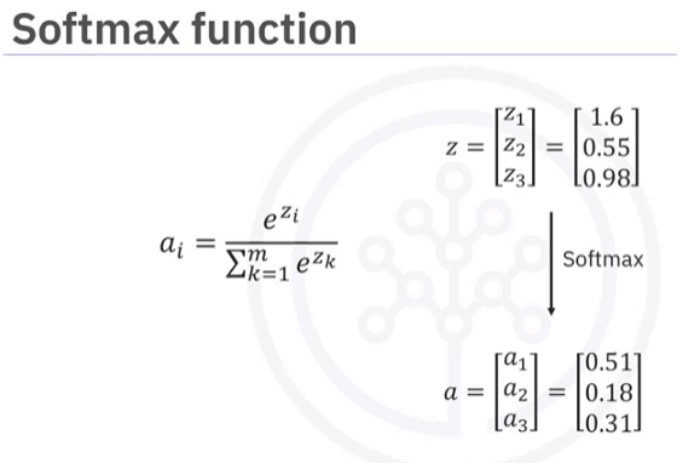
4. Leaky ReLU

$$LeakyReLU(z) = \begin{cases} z & z > 0 \\ 0.01z & z < 0 \end{cases}$$

Fixes “dead ReLU” problem

A small slope for negative region so gradient is not zero.

5. Softmax (for multi-class classification)



Use softmax **only in the output layer** when you want **probabilities** across multiple classes.

Given outputs:

[1.6, 0.55, 0.98]

Softmax converts to:

[0.51, 0.18, 0.31]

Sum = 1.0

Perfect probability distribution.

✔️ Used in:

- MNIST
- Image classification
- Any multi-class output