Weight Initialization Techniques

Objective: Set the **initial weights** of a neural network in a way that enables **efficient training**, avoids **vanishing/exploding gradients**, and accelerates **convergence**.

Bad Initialization Can Cause:

❌ Vanishing gradients (weights → 0)
❌ Exploding gradients (weights → ∞)
❌ Slow convergence (stuck in poor local minima)

❌ **Symmetry problem**: if weights are identical, all neurons learn the same features

😇Ideal Weight Initialization Properties

- Break **symmetry** between neurons
- Keep **activation values and gradients** in a reasonable range (not too small/large)
- Ensure efficient **flow of gradients** during backpropagation

🔴Zero Initialization (Don't Use!)❌

- **Problem**: All weights = 0 → neurons learn identical features (symmetry problem).
- **Never use** in practice.
- **Fails** because all neurons learn the same thing
- ***Why It's Bad? (Interview Question)***: If all weights are 0, every neuron in a layer learns the same thing during training.

- The network can’t tell neurons apart, so it doesn’t learn anything useful.
- No training will take place.
  - Weights will forever be zero

🔴Non-Zero Constant Initialization (Don’t Use!)❌

- Using **the same non-zero constant** (e.g., $W = 0.5$ for all weights) is **just as bad as zero initialization.**

Symmetry Problem (Identical Neurons)

- **Issue**: If all weights in a layer are initialized to the **same value**, every neuron in that layer computes the **same gradient** during backpropagation.
- **Result**: All neurons **learn identical features** → wasted capacity.
- **Example**:
  - If $W_1 = W_2 = 0.5$, then $\partial Loss/\partial W_1 = \partial Loss/\partial W_2$ → weights stay identical forever.

No Signal Differentiation

- **Issue**: Constant weights mean **no randomness** in forward passes.
- **Result**: The network can’t break symmetry, leading to **redundant feature learning**.

Vanishing/Exploding Gradients (Depending on Constant Value)

- If the constant is **too small** (e.g., 0.01):
  - Signals shrink exponentially in deep networks → **vanishing gradients**.
- If the constant is **too large** (e.g., 2.0):
  - Signals grow exponentially → **exploding gradients**.

Random Initialization

To break symmetry and allow neurons to learn diverse features:

1. **Small random values** (e.g., Gaussian noise).
2. **Scale based on layer size** (Xavier/He initialization).

- **Problem**:
  - If weights are **too small** → signals vanish.
  - If weights are **too large** → gradients explode.

✅Xavier/Glorot Initialization (Default in *Keras*)

```
from keras.initializers import GlorotUniform

Dense(128, kernel_initializer=GlorotUniform())

                #OR

Dense(64, activation='tanh', kernel_initializer='glorot_uniform') # Default in Ke
ras
Dense(64, activation='tanh', kernel_initializer='glorot_normal')  # Gaussian
```

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in} + n_{out}}}\right)$$

👨‍🏫 **Best for: Sigmoid, Tanh activations.**

📐 Goal: Keep variance of activations constant across layers

- Scales weights based on **input & output layer sizes** (n_{in}, n_{out}).
    - The idea is to set the weights such that the variance of the outputs of each neuron is the same as the variance of the inputs, which helps in maintaining gradient scale across layers.

How It Works?

- Weights are drawn randomly from a distribution (usually Gaussian or uniform).
- The range is scaled by `sqrt(1 / (fan_in + fan_out))`, where:
    - `fan_in` = number of input neurons.
    - `fan_out` = number of output neurons.
- **Why?** This keeps signal strength stable across layers (no exploding/vanishing gradients).

In Keras:

By default, Keras uses Xavier/Glorot initialization when you use the `Dense` layer without specifying a `kernel_initializer`.

Formula (for uniform distribution):

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$$

For normal distribution:

$$W \sim \mathcal{N}\left(0, \frac{1}{n_{in}+n_{out}}\right)$$

Why It's Good?

- It keeps the signals (math calculations) from getting too big or too small as they pass through layers. This helps avoid vanishing or exploding gradients!

Named after Xavier Glorot, a researcher who came up with this idea!

✅ He Initialization (Best for ReLU/LeakyReLU)

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

What It Does?

- Similar to Xavier, but tweaked for ReLU activation
- **It sets weights to slightly larger random numbers to account for ReLU's behavior.**

```
from keras.initializers import HeNormal
Dense(128, activation='relu', kernel_initializer=HeNormal())
```

```
#OR

kernel_initializer='he_normal'
kernel_initializer='he_uniform'
```

Why It's Good?

- **ReLU (Rectified Linear Unit) sets negative values to 0, which can make signals smaller.**
- **He initialization gives weights a little boost to keep signals strong.**

Best For:

- Use with ReLU or its variants (like Leaky ReLU).

📌 Rule of Thumb:

- **Default to** `he_normal` for ReLU/LeakyReLU.
- Use `he_uniform` only if you have a specific reason (e.g., quantization).

Named after Kaiming He, another researcher!

> 💡 **If you don't specify `kernel_initializer`, Keras uses '`glorot_uniform`' by default for Dense layers**

📌 What Happens if You Get It Wrong?

Bad Initialization	Result
All zeros	Network **doesn’t learn anything** (gradients stay identical).
All ones (or any constant)	Same as zeros—**neurons mirror each other**.
Too large random values	Gradients **explode** (training crashes).
Too small random values	Gradients **vanish** (learning stalls).