DM NOTES:

1. Root Mean Error (RME) vs Root Mean Square Error (RMSE):

- **Root Mean Error** (RME) is not commonly used in machine learning. It may just mean the average of absolute errors.
- **Root Mean Square Error (RMSE)** is:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum(y_{\text{pred}} - y_{\text{actual}})^2}$$

- It gives more weight to big errors.
  Lower RMSE = better model.

2. Gradient

- It tells **how much** and in **which direction** we should change the model's parameters (like weights) to reduce error.
- Think of it as a **guide** to reach the best model.

3. Batch:

- A **small group of data** used to train the model at one time.
- Example: If you have 1000 images and batch size = 100, then training uses 10 batches.

4. Epoch:

- **One complete pass** through the whole dataset during training.
- Example: If you train on the same 1000 images once = 1 epoch.

Batch Size

- If the batch **size is too large**, the model won't fit well to data.
- Model **misses complex patterns** especially when **epochs are few**.

Learning Rate

- If **too high** → Model **misses global minimum** (overshoots).
- If **too low** → Model learns **too slowly** but might reach global minimum.
- Common values: `1e-3`, `1e-5`
- Goal: Reach the best point (minimum loss) – **global or local minima**.

Purpose of Activation Function

- Helps the model **learn complex patterns**.

- Without activation, the model becomes a **simple linear function**.

Artificial Neural Network (ANN)

- **ANN** is also called a **shallow neural network**.
- It has **only one hidden layer**.

Computation Power in ML, DL, and LLMs

- As we move from **Machine Learning (ML)** to **Deep Learning (DL)** and **Large Language Models (LLMs)**:
    - **Computation power increases**.
    - **Why?**
        - The number of parameters in the models increases.
        - Models are built using **advanced statistics**.
        - **Data size increases**, making the network more complex.
- **GPUs are used** instead of CPUs because CPUs are not powerful enough for such heavy tasks.

Optimizer

- An **optimizer** is an algorithm in deep learning that **updates the weights** of a neural network.
- It helps in **reducing the error/loss** during training.
- **Examples of optimizers**:
    - **GD (Gradient Descent)**
    - **SGD (Stochastic Gradient Descent)**
    - **Adam**
    - **RMSprop**

Gradient Descent

- A method used to **train all types of neural network models** like ANN, CNN, and RNN.
- It is the **backbone of deep learning**.
- Its goal is to **reduce the error** by **updating weights**.
- The objective is to **minimize the cost/loss function** with respect to weight parameters (**w**).

“ALGO AND NUMERICAL FROM NOTES”

Hyperparameters

1. Epochs

- Each time the model goes through the full dataset once = **1 epoch**.
- We must set **enough epochs** so the model **learns well** and the **loss reduces**.

2. Learning Rate (η or eta)

- Controls **how fast** the model learns.
- If learning rate is:
    - **Too small** → Learning is **very slow**.
    - **Too large** → May **miss the best solution** (called **local minima**).
- We need an **optimal (just right)** value of learning rate.
- Learning rate should not be too big or too small.
- A good value saves time and gives better results.

Diagram:

- **Big LR (learning rate)** → Skips the best solution.

- **Small LR** → Reaches the best solution slowly.

Stochastic Gradient Descent (SGD)

Idea: Random Sampling to Save Time

- **Example**: Measuring average height of everyone in the world.
    - Too slow to measure **everyone**.
    - Instead, take a **small random sample** (like 1000 people).
    - Their average is **close enough** to the real population average.

How it applies in deep learning:

1. If we have **millions of data points**, it's slow to calculate errors for all.
2. Instead, take a **random sample** (e.g. 5000 images).
3. Calculate **average error** from this sample.
4. This **sample error** is close to the error from the full dataset.
5. Saves time and still trains the model well.

Gradient Efficiency

- The **gradient in SGD** is similar to full-batch gradient descent.
- But it's about **5000 times faster** in computation when using smaller samples.

SGD Algorithm Pseudocode

Assume:

- **N = 1 million (1M)** data points
- **Batch size = 32** (commonly used power of 2: 32, 64, 128, etc.)

Algorithm:

```
for epoch in range(epochs):
    X, Y = shuffle(X, Y)
    for i in range(N // batch_size):
        start = i * batch_size
        end = (i + 1) * batch_size
        X_b, Y_b = X[start:end], Y[start:end]
        gradient_descent(X_b, Y_b)
```

- **Split** data into small batches.
- **Train** on each mini-batch using gradient descent.

Why We Use Gradient Descent

Goals:

1. **To find the global minimum** – the best model performance.
2. **To reduce the cost function** – improve accuracy and reduce error.

Important Concepts

In training, we aim to reach the **global minima**, the point where:

- Model error is **lowest**
- Predictions are **most accurate**

But, reaching the global minimum is **not easy**.

About SGD (Stochastic Gradient Descent)

SGD is a popular optimization technique to reduce model error.

Instead of a straight path, SGD makes **random steps** through the data.

This helps the model **find patterns** in different areas of the dataset.

Randomness helps avoid getting stuck in **bad (sub-optimal) solutions**.

- But it also means that SGD may **not always find the best solution** (global minimum).
- It can get stuck in a **local minimum**.

Gradient Descent (Without Momentum)

Formula:

$$W_t = W_{t-1} - \eta \nabla L(W_{t-1})$$

- W_t: weight at time t
- η: learning rate
- ∇L: gradient of loss

What if the gradient becomes 0?

The weight **stops updating**.

Example:

If $W_t = 5 - (0) \Rightarrow W_t = 5$

→ No change in weight

SGD with Momentum

Equation:

$$V_t = PV_{t-1} - \eta \nabla L(W_{t-1})$$

$$W_t = W_{t-1} + V_t$$

- V_t: velocity (update direction and speed)
- P: momentum term (typically 0.9 or 0.99)
- η: learning rate
- Momentum helps **smooth out updates** and can **jump out of local minima**

➕ **Effect of Momentum:**

- If $P = 0$, the formula becomes **normal gradient descent**.

SGD with Momentum – Algorithm:

```
W  = random value     # initial weight
V₀ = 0                # initial velocity

for each epoch:
    V  = P * V ₋₁ - η * ∇L(W ₋₁)

    W  = W ₋₁ + V
```

“NUMERICAL FROM NOTES”

Unstable Gradient:

- Gradients are used to **update the weights and biases** of a neural network.
- Sometimes, gradients become **very small or very large**.
- During **backpropagation**, we calculate the error and send it backward from the output layer to the input layer to update the weights.
- But sometimes, the gradient:
    - Becomes **very small in the lower layers**, or
    - Becomes **too large**, causing **unstable training**.

Examples:

- `0.5 × 0.2 × 0.4 = 0.04` → Very small → **Vanishing Gradient**
- `2^3 × 4 × 5^2 = >1` → Very large → **Exploding Gradient**

Vanishing Gradient:

- Gradients become **very small** as they go backward through the layers.
- The **lower layers stop learning**, because they get **almost no updates**.

- This makes learning **very slow or even stops completely**.
- The network may get **stuck** and fail to improve performance or **understand complex patterns**.
- Training can fail to **find a good solution**.

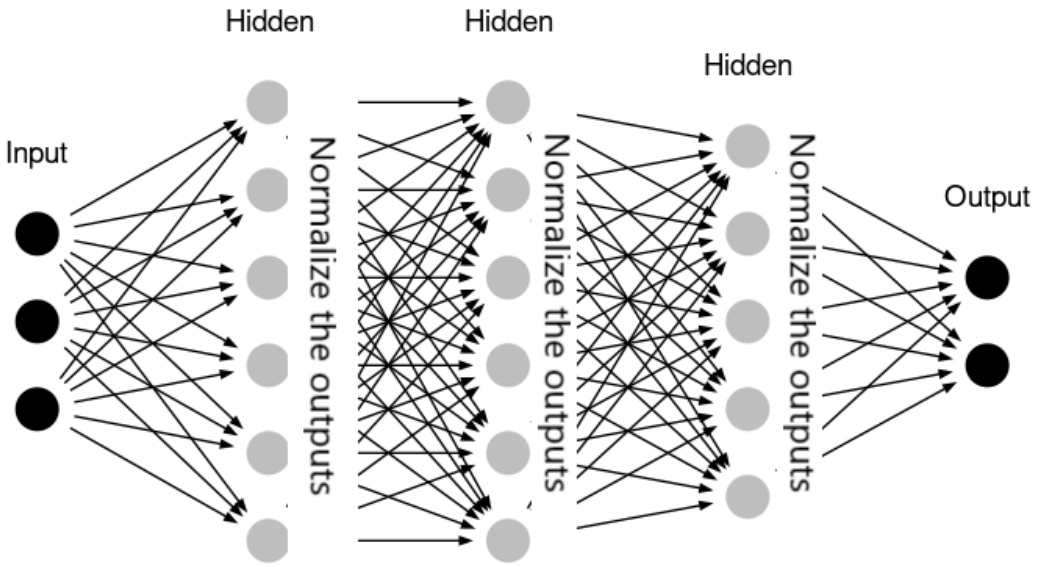
Example:

- `0.5 × 0.2 × 0.4 = 0.04` → Small value

Exploding Gradient:

- Gradients become **too large**, causing **very big updates** to the weights.
- This makes the model **unstable**, and the weights keep **growing out of control**.
- The training process may **diverge** (go in the wrong direction), causing **poor performance or wrong predictions**.
- In very bad cases, the model can **fail completely** due to **numerical overflow**.

BATCH NORMALIZATION:



Batch Normalization (BN)

What is Batch Normalization?

Batch Normalization is a technique used to normalize the inputs of each layer in a neural network, ensuring:

- **Mean = 0**
- **Standard Deviation = 1**

Why Use Batch Normalization?

- Speeds up training

- Improves model performance
- Reduces reliance on regularization methods like Dropout
- Minimizes overfitting and internal covariate shift (i.e., shifting distributions of inputs across layers)

Key Points

- Each batch is normalized independently
- Training converges faster, even if each epoch may take slightly longer due to added computations
- Especially effective in deep networks (e.g., image classification tasks)

How It Works (Per Batch of Inputs)

Let $x^{(i)}$ be the i-th input in a batch of size m.

1. **Compute Batch Mean**

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

2. **Compute Batch Variance**

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu_B)^2$$

3. **Normalize Each Input**

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

(ε is a small constant, like 10^{-8}, to avoid division by zero)

Important Notes

- Adds extra computation per layer

- Still provides major benefits for training deep models
- Typically applied **after the linear (dense/convolutional) layer** and **before the activation function**

NUMERICAL FROM MY NOTES

Tackling Unstable Gradient Problems

Unstable gradients can be reduced using the following techniques:

- **Weight Initialization**
- **Non-saturating Activation Functions** (e.g., ReLU)
- **Batch Normalization**
- **Gradient Clipping**

Gradient Clipping by Norm

To prevent very large gradients (which can make training unstable), we can clip them using a norm:

Example:

Gradient vector:

$$\begin{bmatrix} 0.9 \\ 3.2 \\ -0.2 \\ 0.5 \end{bmatrix}$$

Norm (length of vector): ≈ 4.50

Divide each value by the norm (4.50):

Resulting clipped gradient:

$$\begin{bmatrix} 0.006 \\ 0.21 \\ -0.014 \\ 0.002 \end{bmatrix}$$

This keeps the gradients under control.

Weight Initialization

One of the key methods to deal with unstable gradients.

Things Not Allowed During Weight Initialization:

1. **Don't initialize all weights with 0**
    - If all weights are zero, the model will **not learn** anything. Gradients will be the same, so no useful training happens.
2. **Don't initialize all weights with the same constant value**
    - This causes all neurons to:
        - Produce the same output
        - Have the same gradients
        - Undergo the same weight updates (backpropagation)
    - The model becomes **linear** and **fails to capture complex patterns**.
3. **Don't initialize weights with very large values**
    - Very large weights can cause exploding gradients and unstable training.

Example of Random Initialization:

```
w = np.random.rand(5, 3) * 100
```

- Here, `5, 3` is the shape of the layer (e.g., number of neurons and input size).
- `* 100` multiplies random values by 100 (not recommended — use proper initializers instead like Xavier or He initialization).

Problems When Initializing Weights Incorrectly:

1. **No convergence** – The model cannot reach the correct or best output (global minima).
2. **Exploding gradient problem** – Gradients become too large, making training unstable.
3. **Don't use very small values for weights** – For example:

$$w = np.random.rand(s1, s2) \times 10^{-6}$$

1. This leads to:
    - Slow learning (slow convergence)
    - Training takes longer
    - **Vanishing gradient problem** – Gradients become too small to update weights properly.

✅ Solutions: Use Good Weight Initialization Methods

1. **Glorot (Xavier) Initialization**
2. **He Initialization**
3. **LeCun Initialization**

📐 Formulas:

1. **Glorot (Xavier):**
   - Mean = 0
   - Variance $\sigma^2 = \frac{2}{\text{fan\_in}+\text{fan\_out}}$
2. **He Initialization:**
   - Mean = 0
   - Variance $\sigma^2 = \frac{2}{\text{fan\_in}}$
3. **LeCun Initialization:**
   - Mean = 0
   - Variance $\sigma^2 = \frac{1}{\text{fan\_in}}$

👉 **fan_in** = number of inputs to a neuron

👉 **fan_out** = number of outputs from a neuron

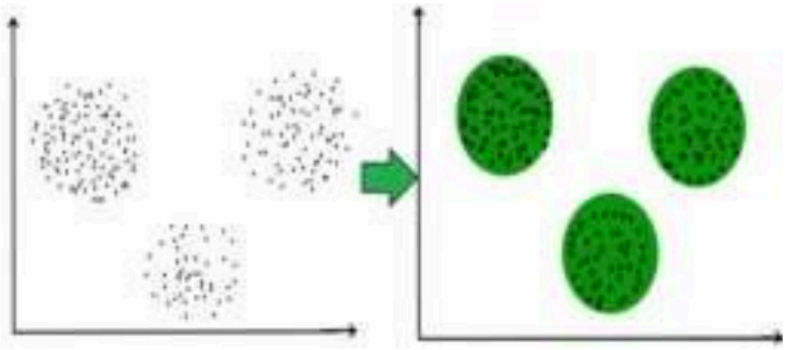
Matching Initialization Methods with Activation Functions:

Weight Initialization	Suitable Activation Functions
Glorot	Linear, Sigmoid, Tanh, Softmax
He	ReLU, Leaky ReLU
LeCun	SELU

What is K-Medoids Clustering?

K-Medoids is a clustering algorithm that groups data points into **k clusters**, just like **K-Means**, but instead of using the average (mean) of the cluster as the center, it uses a **medoid** — a real data point that is the most centrally located in the cluster.

👉 A **medoid** is the object in a cluster for which the total distance to all other points in the cluster is **minimum**.



Why Use K-Medoids?

It is **more robust to noise and outliers** than K-Means because it uses real data points (medoids) instead of average values.

K-Medoids Algorithm:

1. **Initialize**: Randomly select k data points as the initial medoids.
2. **Assign points**: Assign each data point to the nearest medoid (based on a distance measure like Euclidean distance).
3. **Update medoids**:
    - For each cluster, try swapping the current medoid with another point in the cluster.
    - Calculate the total cost (total distance from all points to their medoids).
    - If the new medoid reduces the cost, replace the old one.
4. **Repeat steps 2 and 3** until there are no changes or the cost doesn't improve.

✅ Advantages of K-Medoids:

- Works well with **noisy data** and **outliers**.
- Medoids are **real data points**, making results more interpretable.
- Suitable for **non-numeric** data if proper distance metric is used.

❌ Disadvantages of K-Medoids:

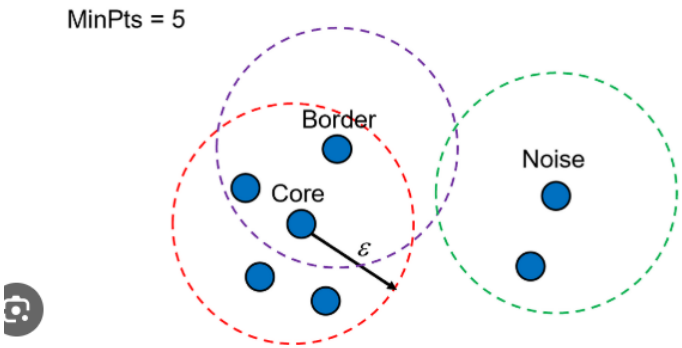
- **Slower** than K-Means, especially for large datasets (because it checks many possible swaps).
- Choosing the right **number of clusters (k)** can be difficult.
- **Not scalable** for big data without optimization.

📌 What is DBSCAN?

DBSCAN is a clustering algorithm that groups data points based on how closely packed (dense) they are.

- It can find **clusters of any shape**.
- It can also detect **noise/outliers** (points that don't belong to any cluster).

Unlike K-Means or K-Medoids, **DBSCAN doesn't need you to specify the number of clusters in advance**.



DBSCAN Algorithm Steps:

DBSCAN uses two main parameters:

- **ε (epsilon)**: The maximum distance between two points to be considered neighbors.
- **minPts**: The minimum number of points required to form a dense region (a cluster).

Steps:

1. **Start with an unvisited point**.
2. **Find all neighboring points** within distance **ε**.
3. If neighbors ≥ **minPts**:
    - Start a new cluster.
    - Add the point and all its neighbors to the cluster.

    - Repeat for all neighbors.
4. If neighbors < **minPts**, mark it as **noise** (but it may later become part of another cluster).
5. **Repeat** until all points are visited.

✅ Advantages of DBSCAN:

- Can find **clusters of any shape** (not just circular like K-Means).
- Automatically detects **outliers/noise**.
- No need to specify the number of clusters beforehand.
- Works well when clusters have **different sizes and densities**.

❌ Disadvantages of DBSCAN:

- Choosing the best **ε and minPts** values can be tricky.
- Doesn't work well if the data has **varying densities**.
- Struggles with **high-dimensional data** (like with many features).

📌 Elbow Method

Purpose:
The Elbow Method is used to **find the optimal number of clusters (k)** in **K-Means Clustering**.

🧠 Basic Idea:

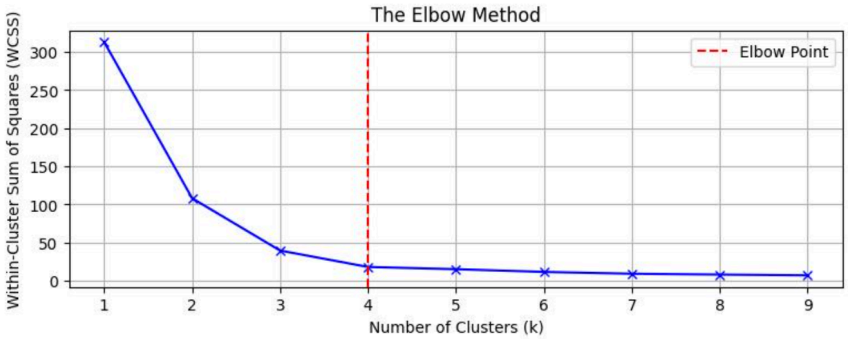
- We try different values of **k** (number of clusters).
- For each k, we calculate the **inertia** (also called **within-cluster sum of squares – WCSS**).
- As k increases, **WCSS decreases** (clusters get smaller).
- But after a point, the **decrease becomes very small** — this point is called the **elbow**.
- The **"elbow point"** is the best k (optimal number of clusters).

🔍 In the Plot:

- The curve drops quickly at first, then slows down.
- The **point where it bends (forms an "elbow")** is the best k.

✅ Conclusion:

- The Elbow Method helps choose the best number of clusters for K-Means.
- Look for the “elbow” where adding more clusters doesn’t improve much.



📌 Silhouette Score:

Purpose:
The Silhouette Score helps us find the **best number of clusters** by checking how well the data points fit **within their cluster** and how **separate** the clusters are.

Basic Idea:

- It measures **how similar a data point is to its own cluster** vs. **other clusters**.
- Score ranges from **-1 to 1**:
    - **+1**: Perfectly matched to its own cluster.
    - **0**: On or near the boundary between clusters.
    - **-1**: Wrongly assigned to the wrong cluster.

✅ Goal:

Choose the **number of clusters (k)** that gives the **highest average Silhouette Score**.

📊 Steps:

1. Try different values of k (e.g., 2 to 10).
2. For each k:
    - Fit KMeans clustering.
    - Calculate the average silhouette score.
3. Select the k with the **highest score**.

📌 Output Might Look Like:

For k = 2, Silhouette Score = 0.71

For k = 3, Silhouette Score = 0.66

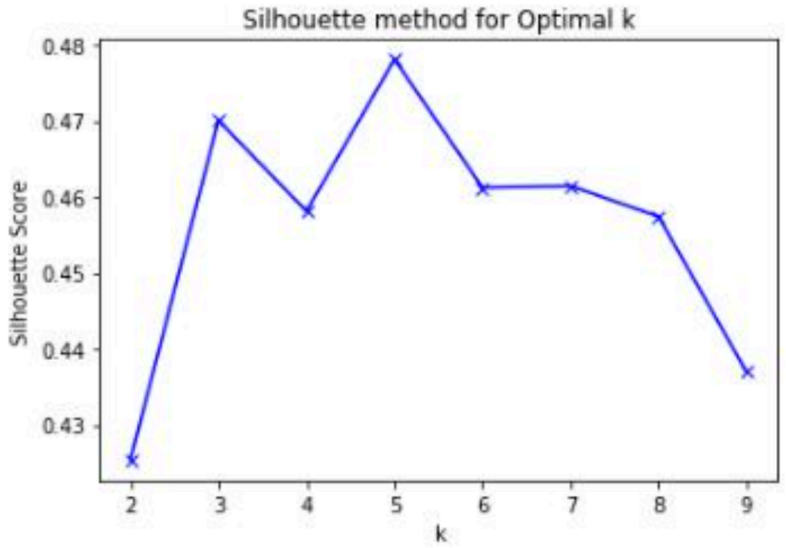
For k = 4, Silhouette Score = 0.45

For k = 5, Silhouette Score = 0.39

👉 The **best k = 2** here, because it has the **highest silhouette score**.

✅ Conclusion:

- Silhouette Score helps find the best number of clusters.
- **Higher score = better clustering.**
- Pick the **k with the highest silhouette score**.



Measuring Clustering Quality

Clustering quality tells us **how well data has been grouped**. A good clustering should have:

- **High intra-cluster similarity** (similar items are in the same group)
- **Low inter-cluster similarity** (different groups are distinct)

🔹 I. Theoretical Measures of Clustering Quality

These are based on **principles** and **ground truth labels**.

1. Dissimilarity/Similarity Metric

- Distance function **d(i, j)** is used to check how similar or dissimilar two points are.
- Examples:
    - **Euclidean distance** – for numeric data
    - **Cosine similarity** – for text or vector data

2. Cluster Completeness

- A clustering has high quality if **similar data points** are assigned to the **same cluster**.
- If two sub-clusters (s1, s2) contain same-category objects, merging them (as in clustering C2) increases quality:
    - Q(C2, Cg) > Q(C1, Cg)

3. Ragbag Criterion

- If a data point does **not fit clearly** into any cluster, it's better to put it in a **"rag bag"** (a mixed/noisy cluster).
- This **improves overall quality** by keeping clusters pure.
    - Again, Q(C2, Cg) > Q(C1, Cg) where the noisy point is moved to the rag bag.

4. Small Cluster Preservation

- Small categories should not be split into many tiny clusters, or they become **noisy and hard to detect**.
- Avoid breaking small meaningful clusters.
    - Q(C2, Cg) > Q(C1, Cg) if C2 keeps the small cluster intact.

🔹 II. Practical Evaluation Metrics

These are **numeric scores** used in real-world clustering (especially when ground truth is or isn't available).

✅ When No True Labels (Internal Metrics):

Metric	Meaning	Good Score
Silhouette Score	How similar a point is to its cluster vs. others	Close to **+1**

Dunn Index	Ratio of min inter-cluster to max intra-cluster distance	**Higher is better**
Davies-Bouldin Index	Measures similarity between clusters	**Lower is better**

✅ When True Labels Are Available (External Metrics):

Metric	Meaning	Good Score
Rand Index (RI)	% of correctly clustered pairs	High score
Adjusted Rand Index (ARI)	Adjusted for random chance	Close to **1**
Normalized Mutual Information (NMI)	Shared info between real and predicted labels	Close to **1**

Fowlkes-Mallows Index (FMI)	Measures accuracy of clustering vs truth	Close to **1**

Overfitting vs Underfitting

🎯 Goal of ML Model:

A good model should:

- **Learn patterns** from training data
- **Work well** on new (test) data
- **Avoid both overfitting and underfitting**

🔶 Overfitting

✅ What is it?

Model learns **too much**, including noise or irrelevant data.
✅ Training accuracy = High
❌ Test accuracy = Low

✅ Why it happens?

- Model is **too complex**
- **Too little data**
- **Too many features**
- Training for **too long**

✅ Example:

Like a student who **memorizes** answers but **doesn't understand**—fails in real exam.

✅ Fix it:

- **Simplify** the model
- Use **more training data**
- Add **regularization** (L1/L2)
- Use **dropout** or **early stopping**

🔷 Underfitting

✅ What is it?

Model is **too simple** and fails to learn patterns.
❌ Training accuracy = Low
❌ Test accuracy = Low

✅ Why it happens?

- Model is **too basic**
- **Too few features**
- **Training stopped too early**
- **Data not scaled properly**

✅ Example:

Like a student who didn't study at all—**fails both practice and real test**.

✅ Fix it:

- Use a **more powerful model**
- **Train longer**
- Do **feature engineering**
- Make sure **data is clean and scaled**

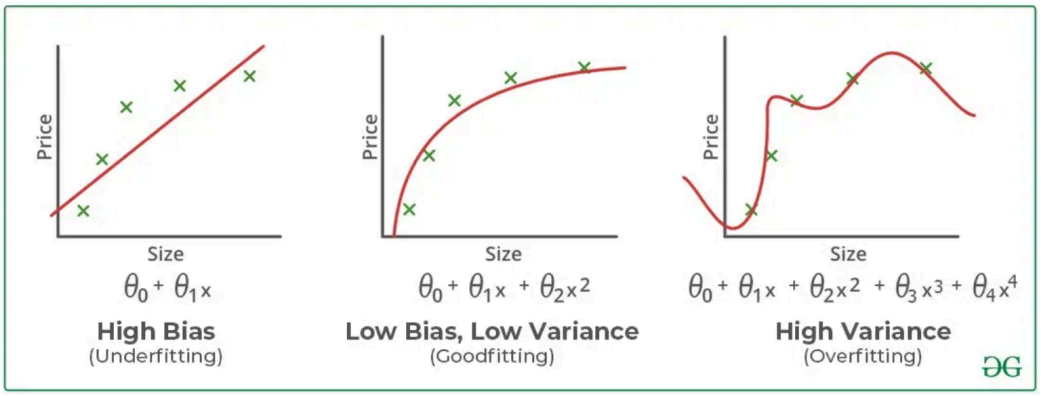
📊 Bias-Variance Tradeoff

Term	Meaning	Leads To
Bias	Assumes too little (too simple)	Underfitting
Variance	Learns too much (too complex)	Overfitting

👉 **Good Model** = Low bias + Low variance (Balanced)

Memory Trick:

Underfitting = Undertrained (Too simple)
Overfitting = Overtrained (Too complex)


Price
Size
$\theta_0 + \theta_1 x$
High Bias
(Underfitting)
Price
Size
$\theta_0 + \theta_1 x + \theta_2 x^2$
Low Bias, Low Variance
(Goodfitting)
Price
Size
$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$
High Variance
(Overfitting)