HELMHOLTZ ZENTRUM
DRESDEN ROSSENDORF

Deep Learning Neural Network Block

HIDA_2025

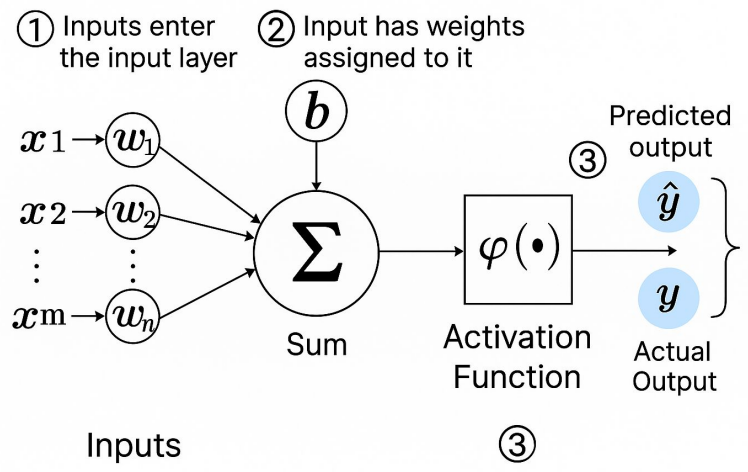
Israt Jahan Tulin, Scientific Researcher, AI Consultant Team, HZDR, Germany
Date: October 27, 2025

DRESDEN concept

Step 1: Forward Propagation

- **Inputs:** $x_1, x_2, x_3 \ldots\ldots x_m$
- Each input has an associated **weight:** w_1, w_2, w_3
- A single **neuron** in the hidden layer **computes**: $z = x_1w_1 + x_2w_2 + x_3w_3 + b$
- Output from the neuron after **activation** (e.g., sigmoid, ReLU): $\hat{y} = activation(z)$
- **Loss Function:** Measures the difference between prediction $\hat{y}$ and true label y.

  Example (MSE - Mean Squared Error): $Loss = \frac{1}{2}(y - \hat{y})^2$



Difference between predicted output and actual output

https://medium.com/@jainvidip/forward-and-backward-propagation-in-multilayered-neural-networks-a-deep-dive-d596e875dedf

Step 2: Backpropagation Concept

- **Goal:** Minimize the loss by adjusting weights.
- **Compute gradient:** Use calculus (chain rule) to compute the gradient of the loss w.r.t each weight.
- **Gradient Descent Rule:** Update weights in the opposite direction of the gradient:

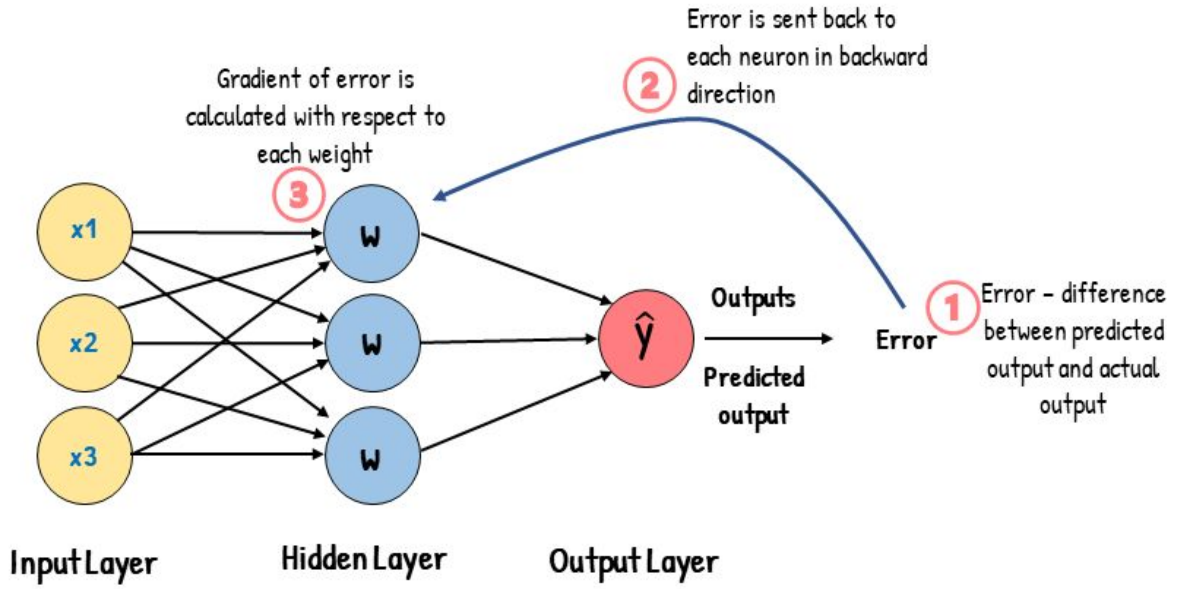
$$w_i^{new} = w_i^{old} - \eta \cdot \frac{\partial Loss}{\partial w_i}$$

where:

η = learning rate (controls how big each step is)

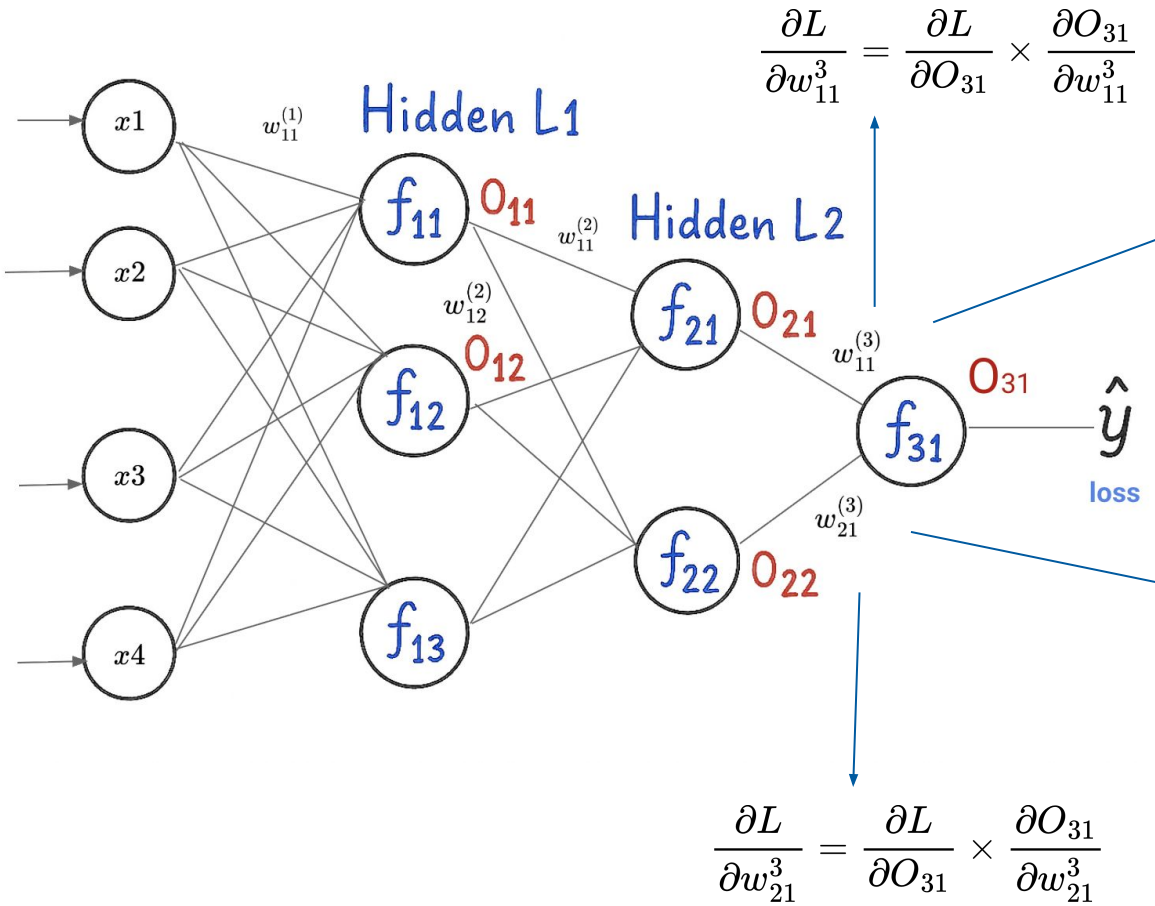
$\frac{\partial \text{Loss}}{\partial w_i}$ = how sensitive the loss is to weight w_i

- **Optimizer:** (Gradient Descent, Adam, RMSprop) do this updating process



https://vijaygadre.medium.com/deep-learning-explained-a-beginners-guide-42fd8e8a770b

CHAIN Rule in BACKPROPAGATION



$$w_{11,\ new}^{(3)} = w_{11,\ old}^{(3)} - \eta \cdot \frac{\partial L}{\partial w_{11,\ old}^{(3)}}$$

update weight | old weight | learning rate | gradient

$$w_{21,\ new}^{(3)} = w_{21,\ old}^{(3)} - \eta \cdot \frac{\partial L}{\partial w_{21,\ old}^{(3)}}$$

https://www.youtube.com/watch?v=cxPAvoIbsIk&list=PLZoTAELRMXVPGU70ZGsckrMdr0FteeRUi&index=8

CHAIN Rule in BACKPROPAGATION

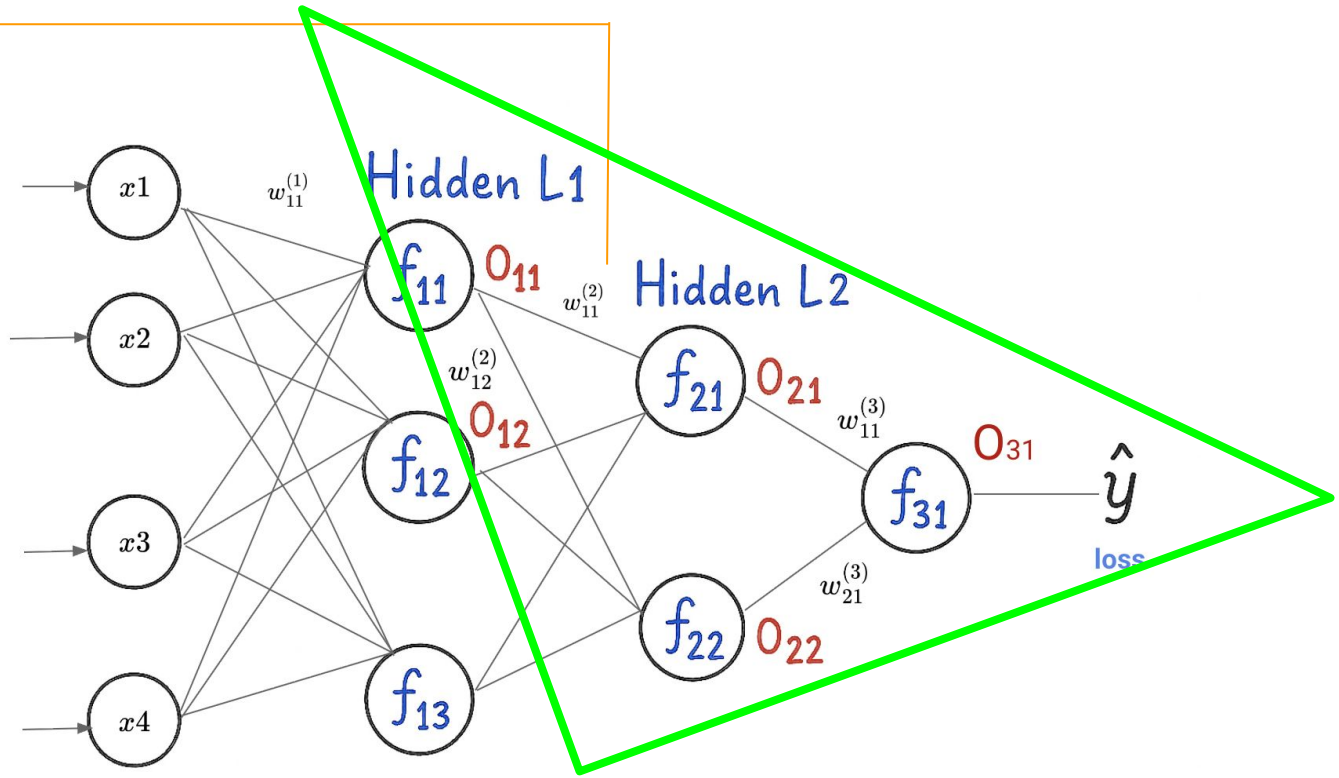
$$w^{(2)}_{11,\ new} = w^{(2)}_{11,\ old} - \eta \cdot \frac{\partial L}{\partial w^{(2)}_{11}}$$

update weight

old weight

learning rate

gradient



$$\frac{\partial L}{\partial w^{(2)}_{11}} = \left(\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w^{(2)}_{11}} \right) + \left(\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial w^{(2)}_{12}} \right)$$

https://www.youtube.com/watch?v=cxPAvoIbsIk&list=PLZoTAELRMXVPGU70ZGsckrMdr0FteeRUi&index=8

Exploding Gradients

$$\frac{\partial O_{31}}{\partial O_{21}} = \frac{\partial \left(w_{11}^{(3)} \cdot O_{21} \right)}{\partial O_{21}} = w_{11}^{(3)}$$

$$\frac{\partial L}{\partial w_{11}^{(2)}} = \left(\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial w_{11}^{(2)}} \right) + \left(\frac{\partial L}{\partial O_{31}} \cdot \frac{\partial O_{31}}{\partial O_{22}} \cdot \frac{\partial O_{22}}{\partial w_{12}^{(2)}} \right)$$

activation derivative

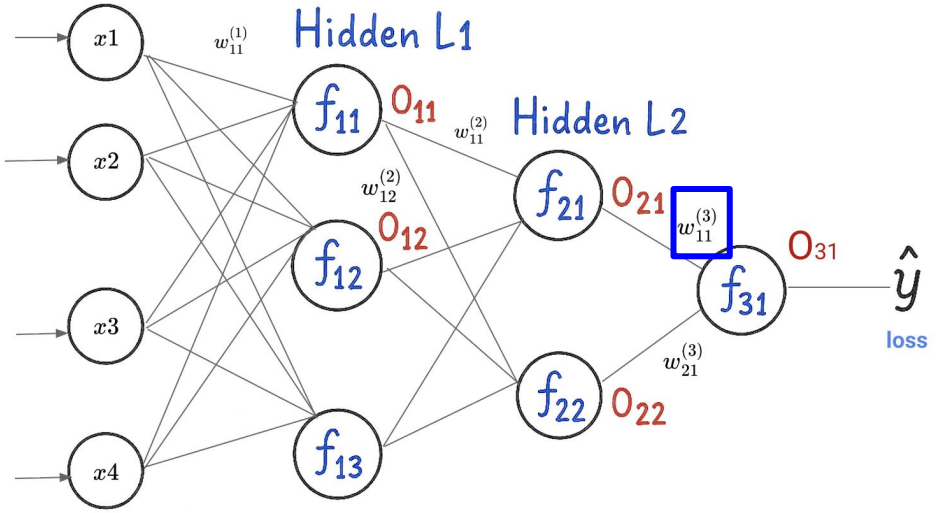
weights in layer 3

What Happens?

- If weights were very large, the outputs became very large
  - → **During backpropagation,** at each layer, they are multiplied by:
    - Weights of the layer
    - Activation derivatives
  - multiplying these large values made gradients very big → **exploding gradients**

Why Exploding Happens?

- If weights are **large (>1)**, every backward step **amplifies** the gradient.
- After many layers, the gradient becomes **huge** (exponential growth).



This made deep networks very hard to train.

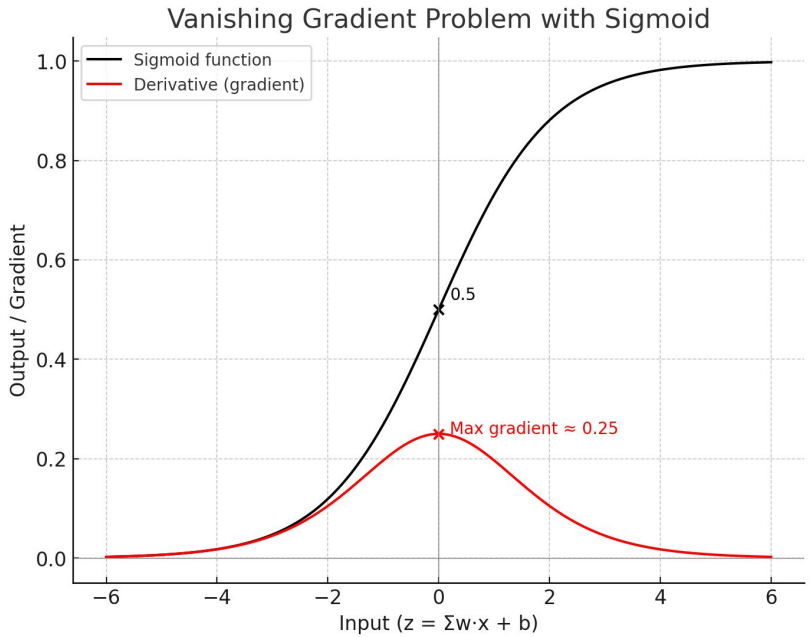
https://www.youtube.com/watch?v=cxPAvoIbsIk&list=PLZoTAELRMXVPGU70ZGsckrMdr0FteeRUi&index=8

Vanishing Gradients

What Happens?

- If weights were very small and we used sigmoid/tanh activations (which squash values close to 0)
  - → **During backpropagation,** multiplying these small values made gradients very tiny → **vanishing gradients**

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \text{gradient}$$



Why Vanishing Happens?

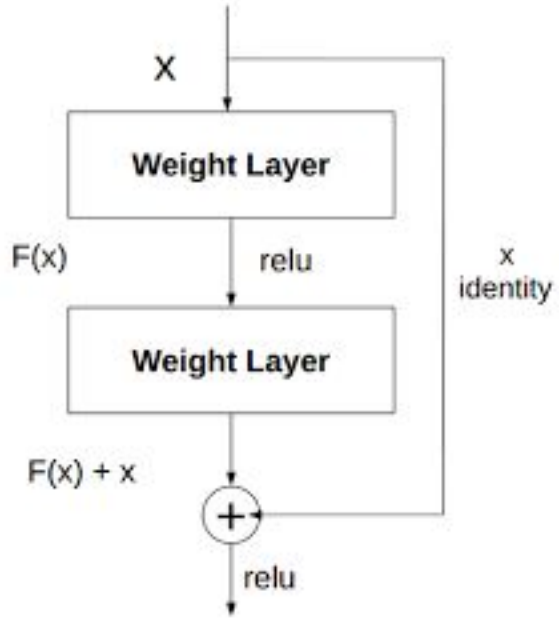
- If each gradient is small (< 1), the product shrinks toward 0.
- When the gradient becomes almost zero:
  - The update step is extremely tiny: $w_{\text{new}} \approx w_{\text{old}}$
  - Early layers receive almost no update

This makes training slows down or even stops

https://www.youtube.com/watch?v=JIWXbzRXk1I&list=PLZoTAELRMXVPGU70ZGsckrMdr0FteeRUi&index=10

What is the solution now?

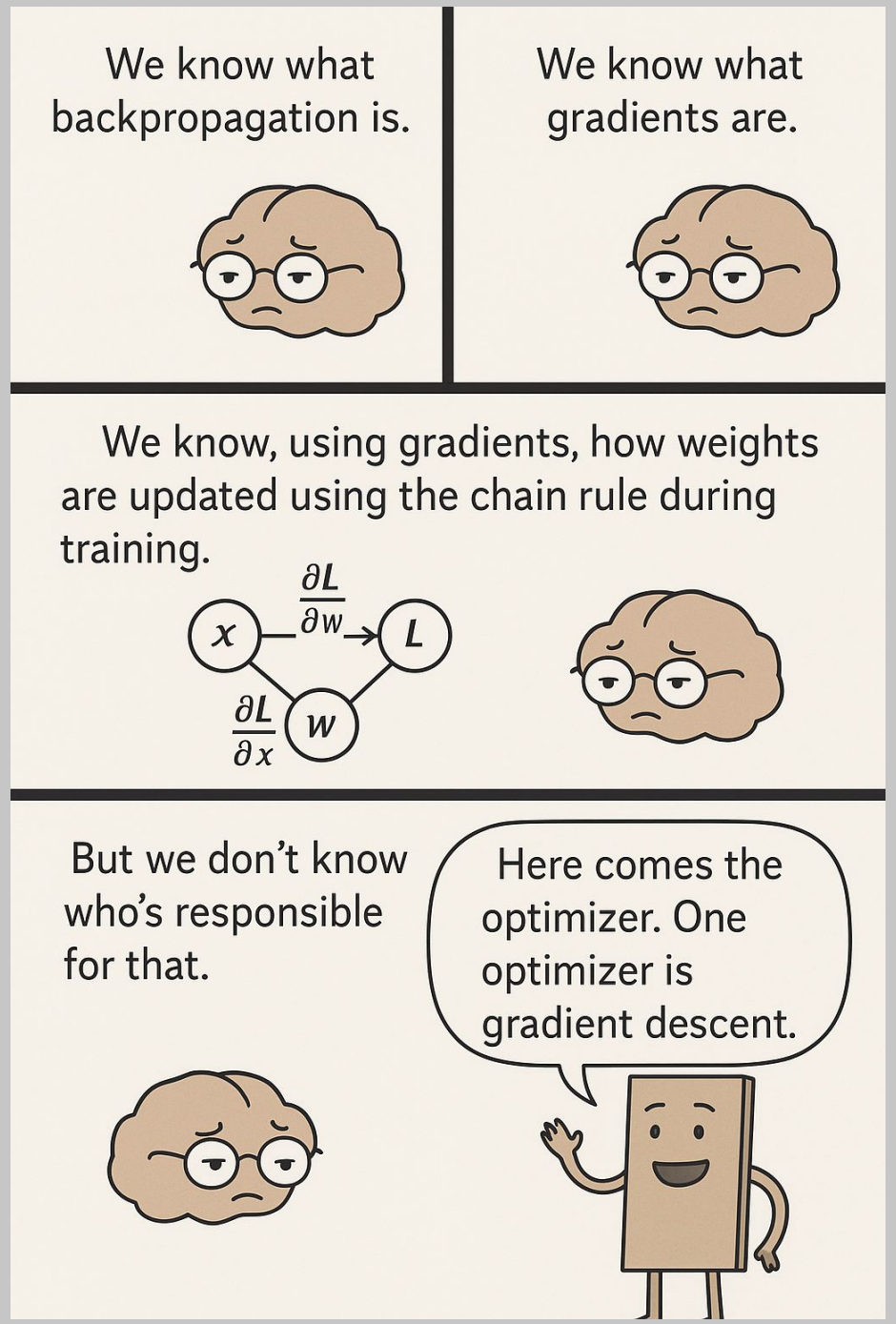
- **ReLU activations** → prevent values from shrinking too much.
- **Batch Normalization** → keeps activations well-scaled, not too large or too small, making training more stable.
    - output → BatchNorm (normalize) → Activation function (like ReLU or tanh)
- **Skip Connections** → Creates a shortcut that jumps over some layers. So, during propagation, gradient does not get multiplied repeatedly by small numbers through many layers and hence, gradient doesn't vanish. For example **ResNets**.
- **Normalize input data (standard scaler/MinMax scaler form sklearn):** Scale training data to a small range (like 0–1) → This prevents very large or very small numbers that could shrink gradients during backpropagation.
- **Proper Weight Initialization** → Xavier, He initialization



https://www.researchgate.net/publication/354907417_Scalable_deeper_graph_neural_networks_for_high-performance_materials_property_prediction

What is Optimizer

Adjusts a model’s weights to reduce the loss



Gradient Descent

Idea:

Gradient Descent is a method used to find the lowest point (minimum) of a curve or surface. Think of it like trying to find the bottom of a valley when you're walking downhill.

- Imagine you are **standing on top of a hill**
- You want to **reach the lowest point**
- You **can't see the whole hill**, but you **can feel the slope** under your feet
- So you take **small steps downhill each time**

generated by chatgpt

This method is called Gradient Descent.

Direction of Gradients vs Our Step

Gradient = points uphill

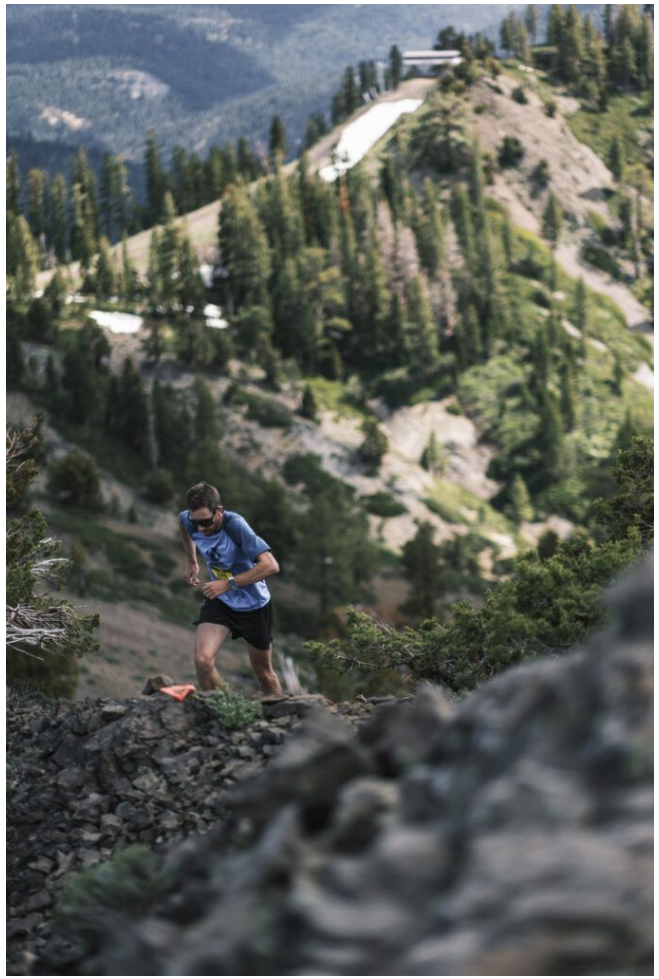
We want to go downhill → **go opposite the gradient**

The **gradient is like an arrow** showing where the ground goes **up the fastest**

If you walk **with the arrow** → **you go up the hill** (loss increases)

Our goal is to **reach the bottom** (make loss small)

So we **move opposite the arrow (gradient)** → this takes us **downhill**

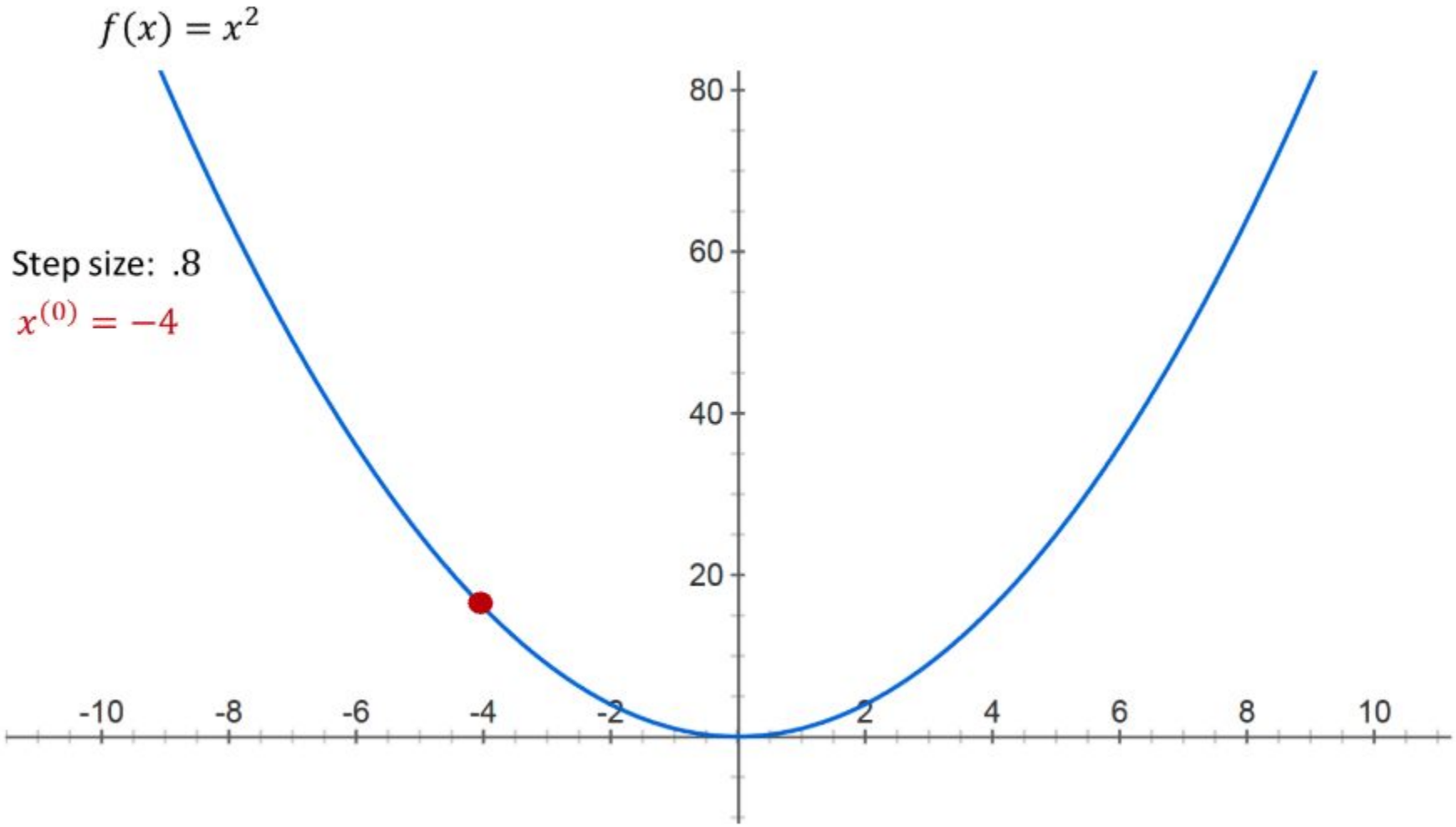


generated by chatgpt

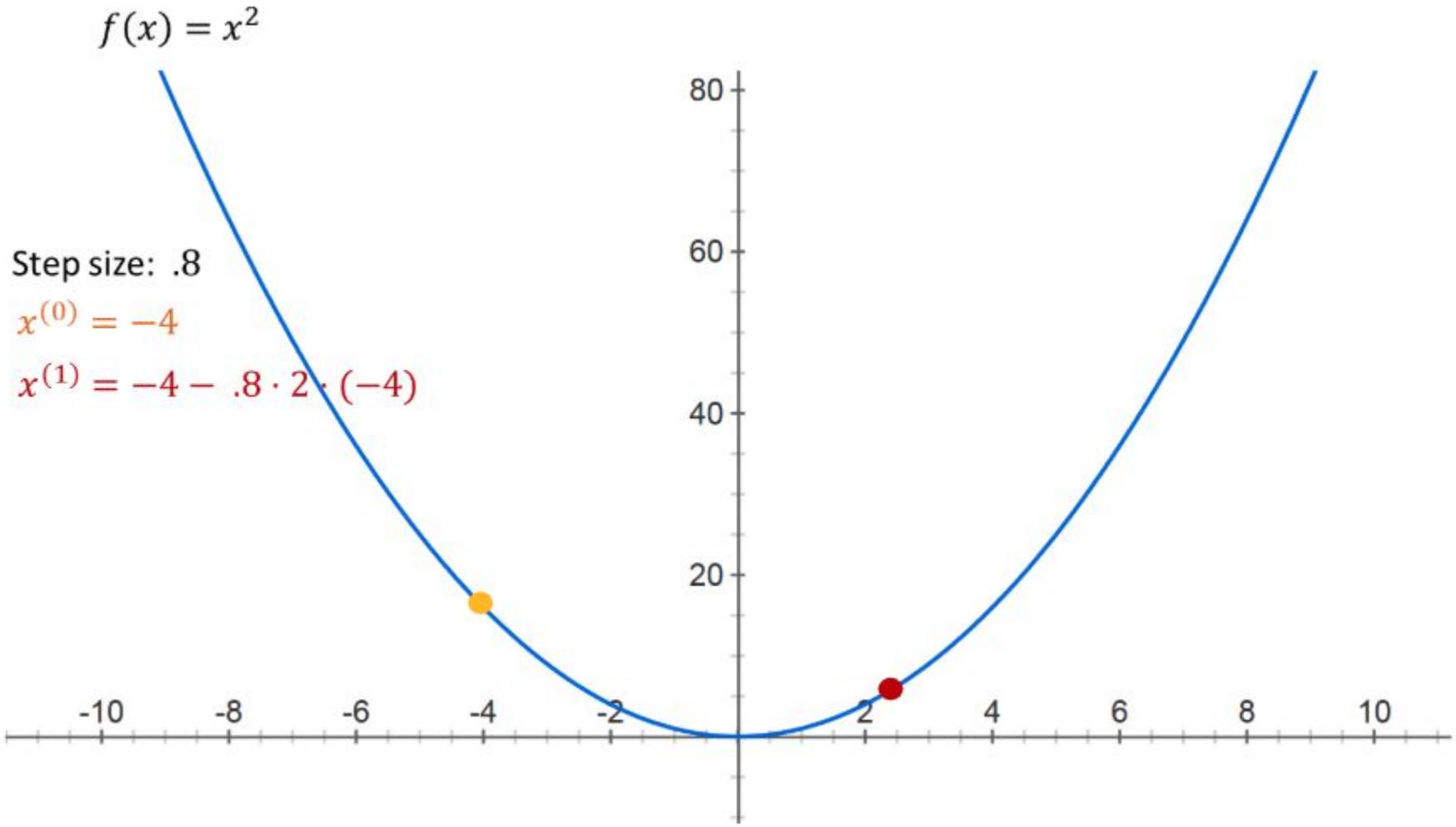
This is the **main reason** why we **subtract** the derivative (gradient) from the old weight in gradient descent.

$$w_i^{new} = w_i^{old} - \eta \cdot \frac{\partial Loss}{\partial w_i}$$

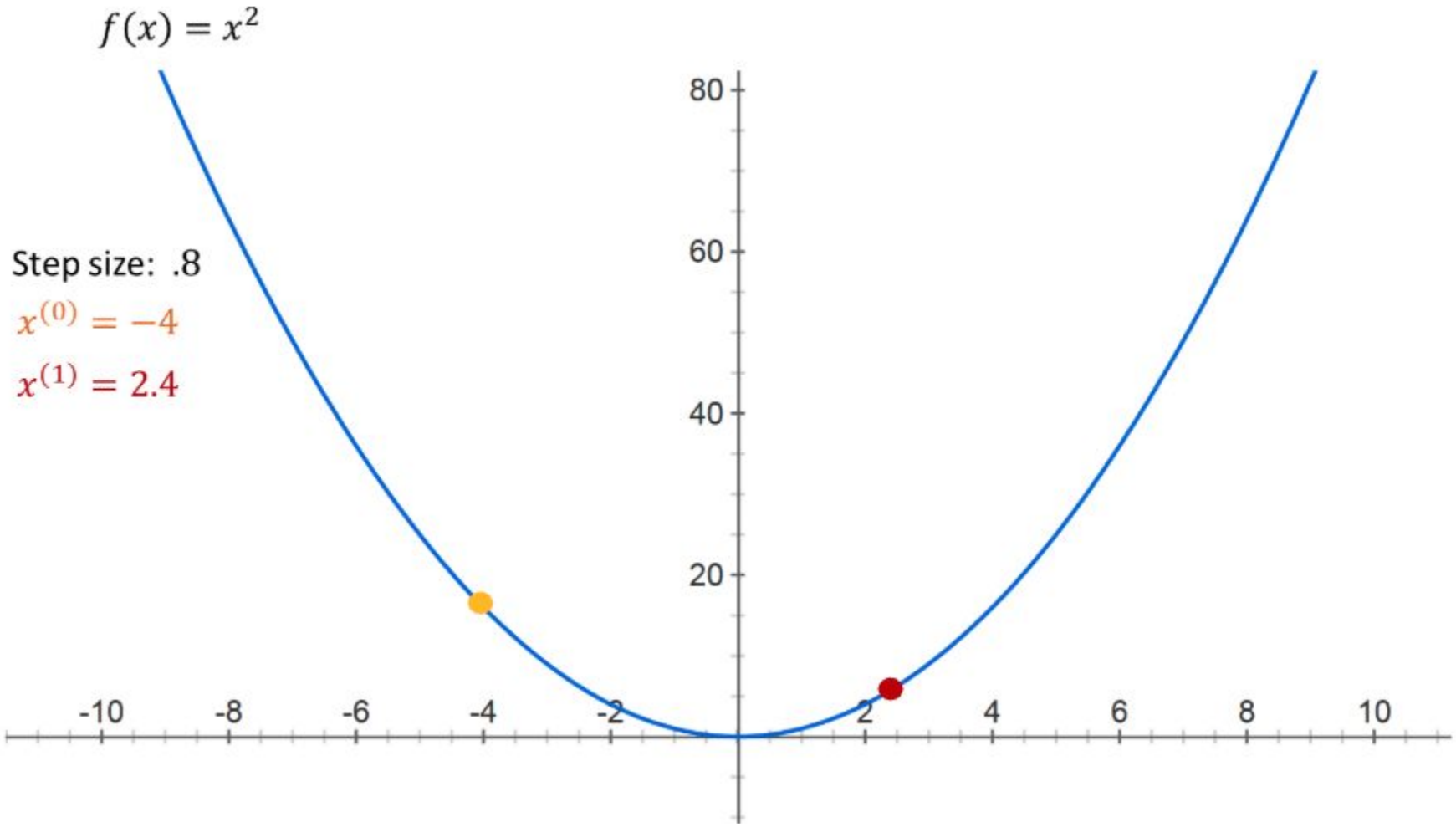
How Weights Change in Gradient Descent



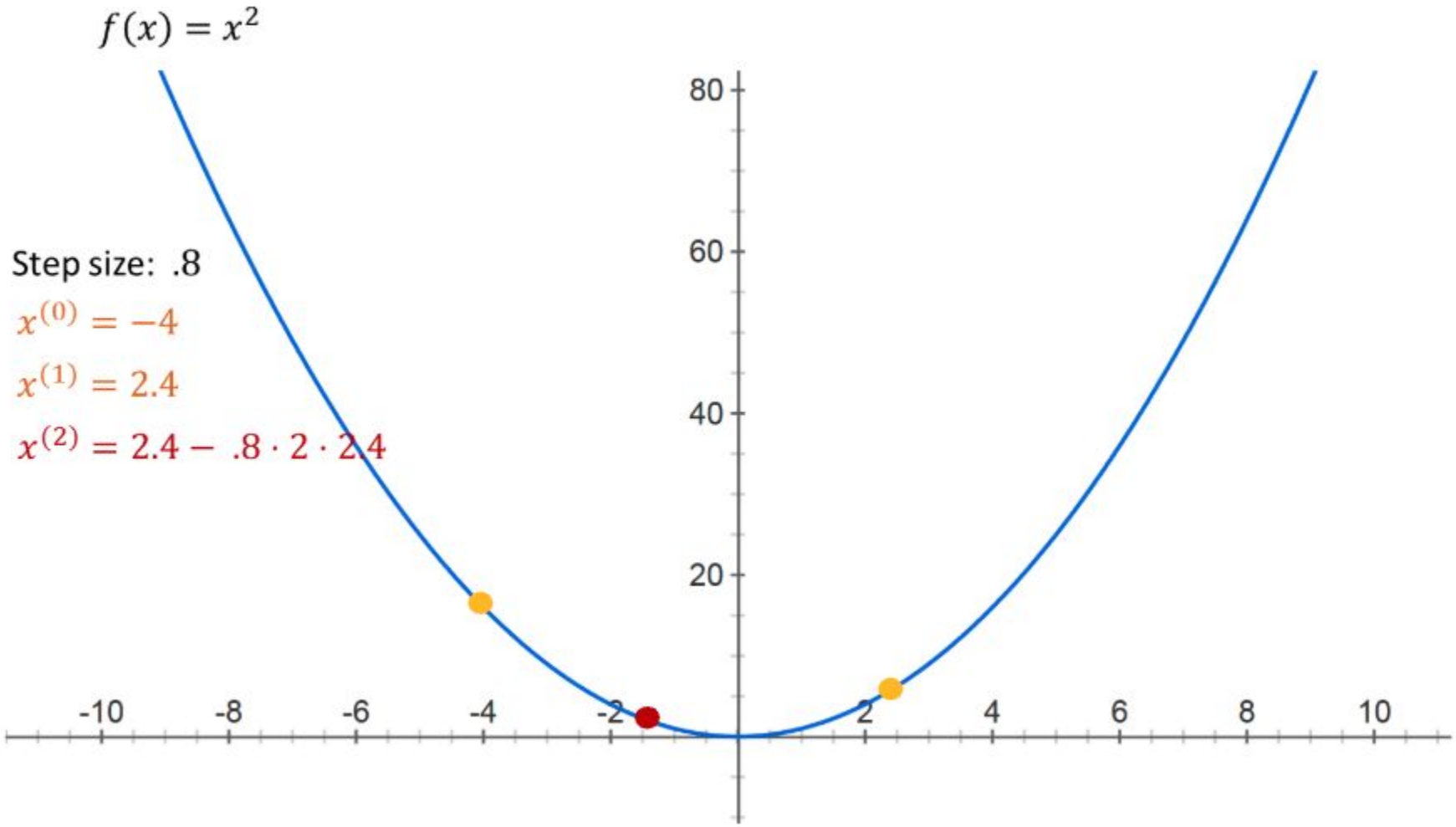
How Weights Change in Gradient Descent



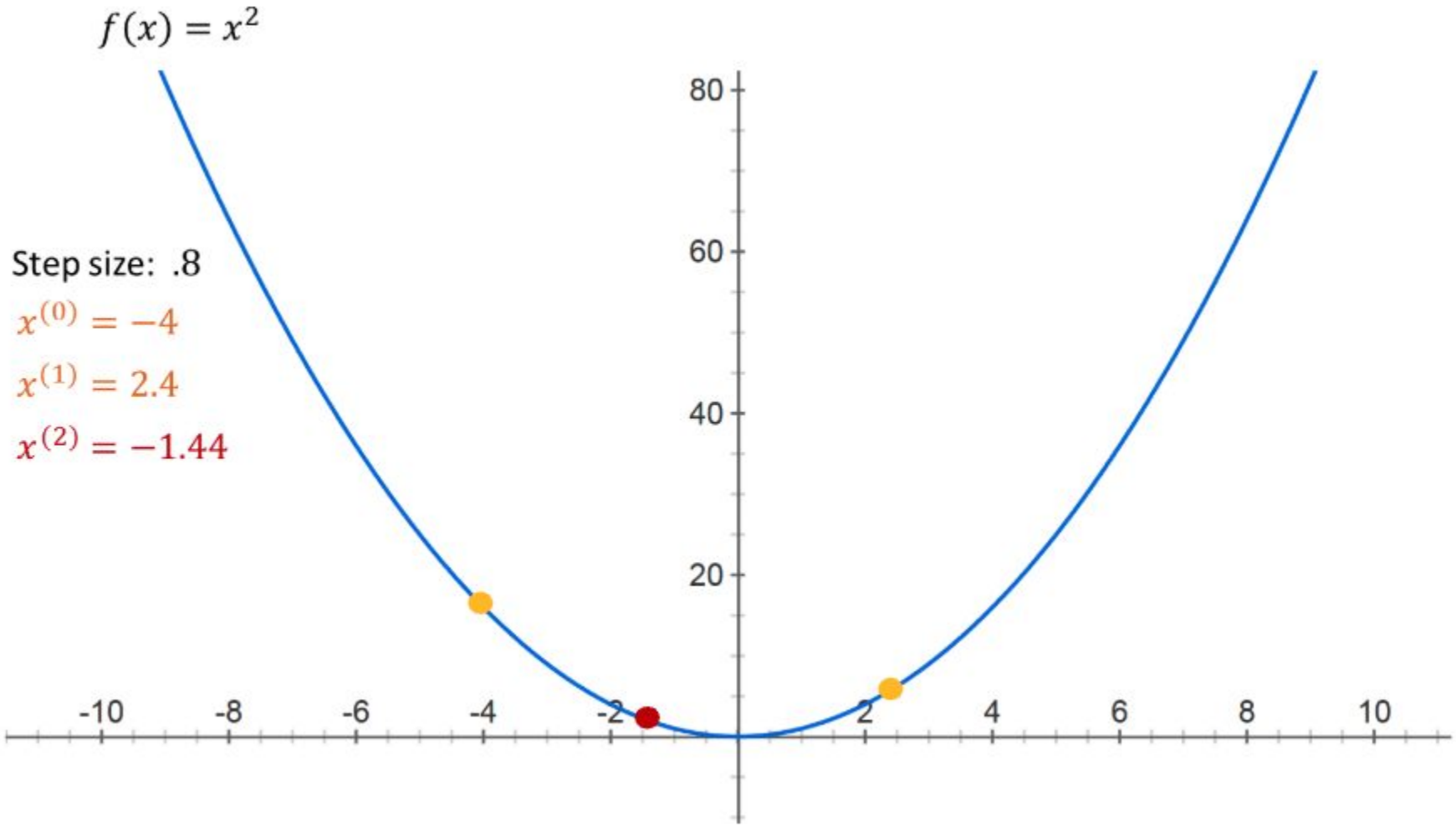
How Weights Change in Gradient Descent



How Weights Change in Gradient Descent

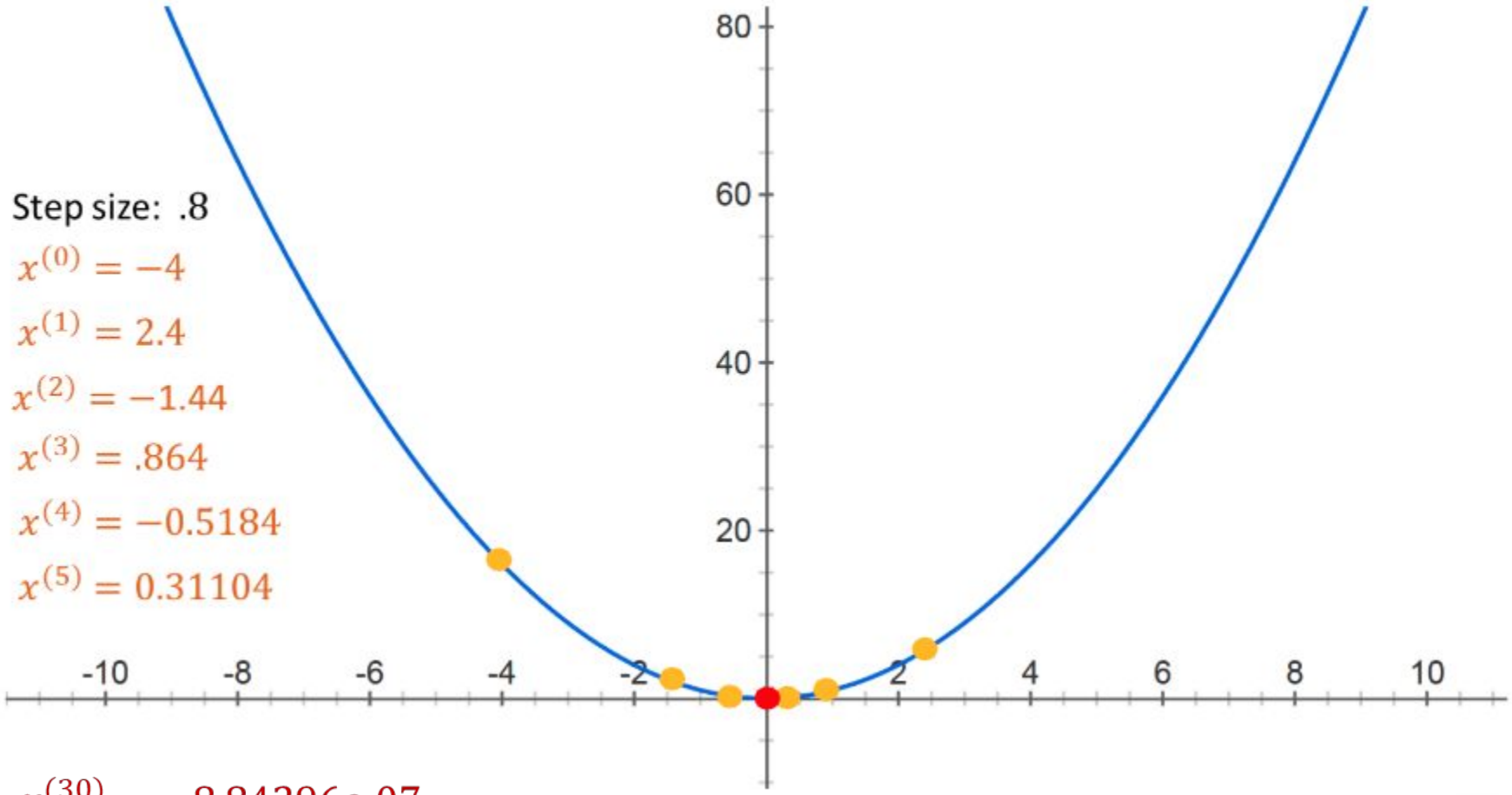


How Weights Change in Gradient Descent



How Weights Change in Gradient Descent

$f(x) = x^2$



Step size: .8

$x^{(0)} = -4$

$x^{(1)} = 2.4$

$x^{(2)} = -1.44$

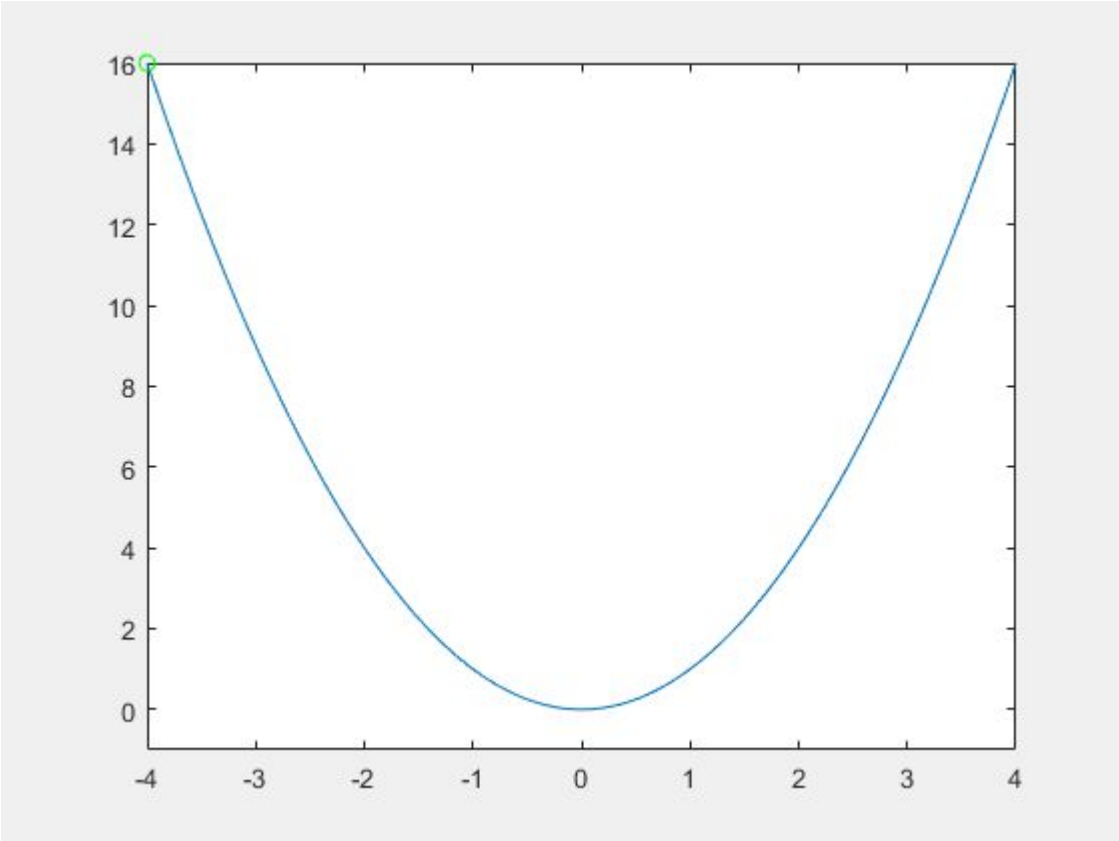
$x^{(3)} = .864$

$x^{(4)} = -0.5184$

$x^{(5)} = 0.31104$

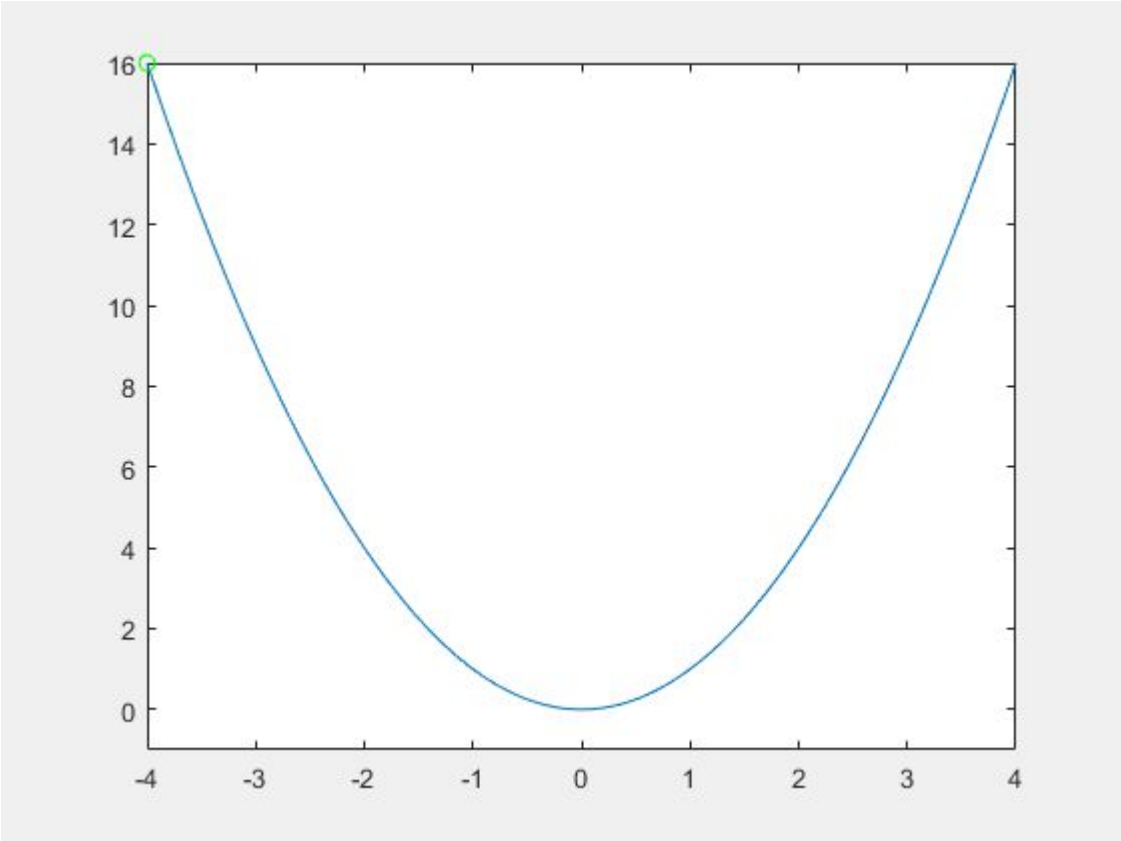
$x^{(30)} = -8.84296e\text{-}07$

Gradient Descent



Step size: .9

Gradient Descent



Step size: .2

Gradient Descent Summary

- The network has one single set of weights (shared by all samples → input data)
- At the start → weights are randomly initialized once
- One training round (called an epoch):
  - Send all samples through the network → **forward propagation**
  - Calculate loss for all samples
  - Compute gradients for all samples→ **backpropagation**
  - Calculate gradients with respect to loss for all samples and then **average** all gradients
  - Update the weights once using this average gradient

❗ **Problem: Using all samples each time makes training very slow on large datasets.**

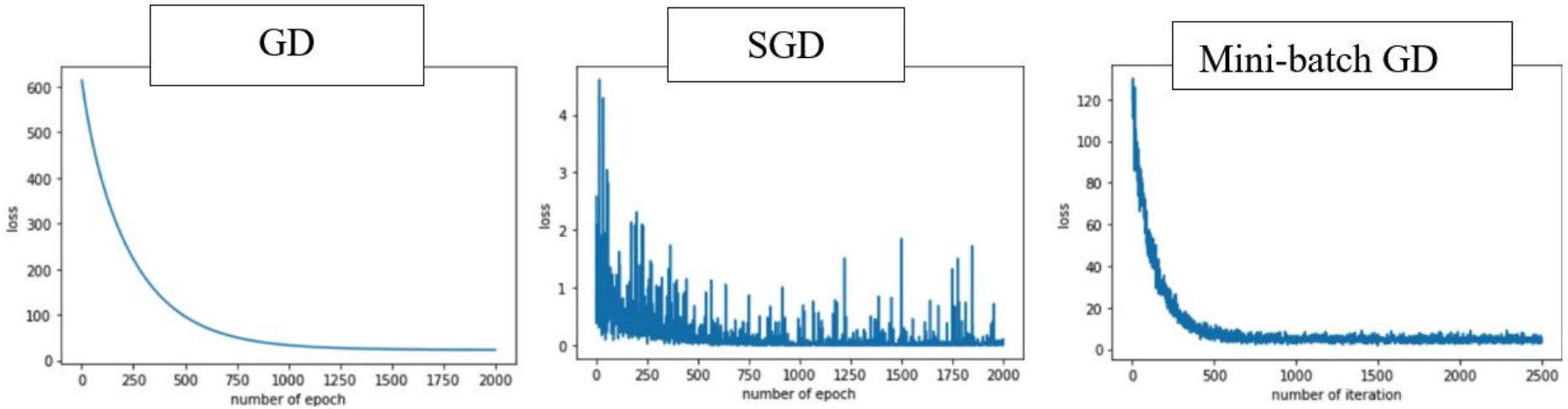
Solution

Stochastic Gradient Descent (SGD)

- A faster version of Gradient Descent
- Uses one sample at a time instead of all samples together
  - Pick one sample
  - Do forward & backward → compute error (gradient)
  - Update weights immediately
  - Repeat for all samples → this completes **1 epoch**

Mini-batch SGD

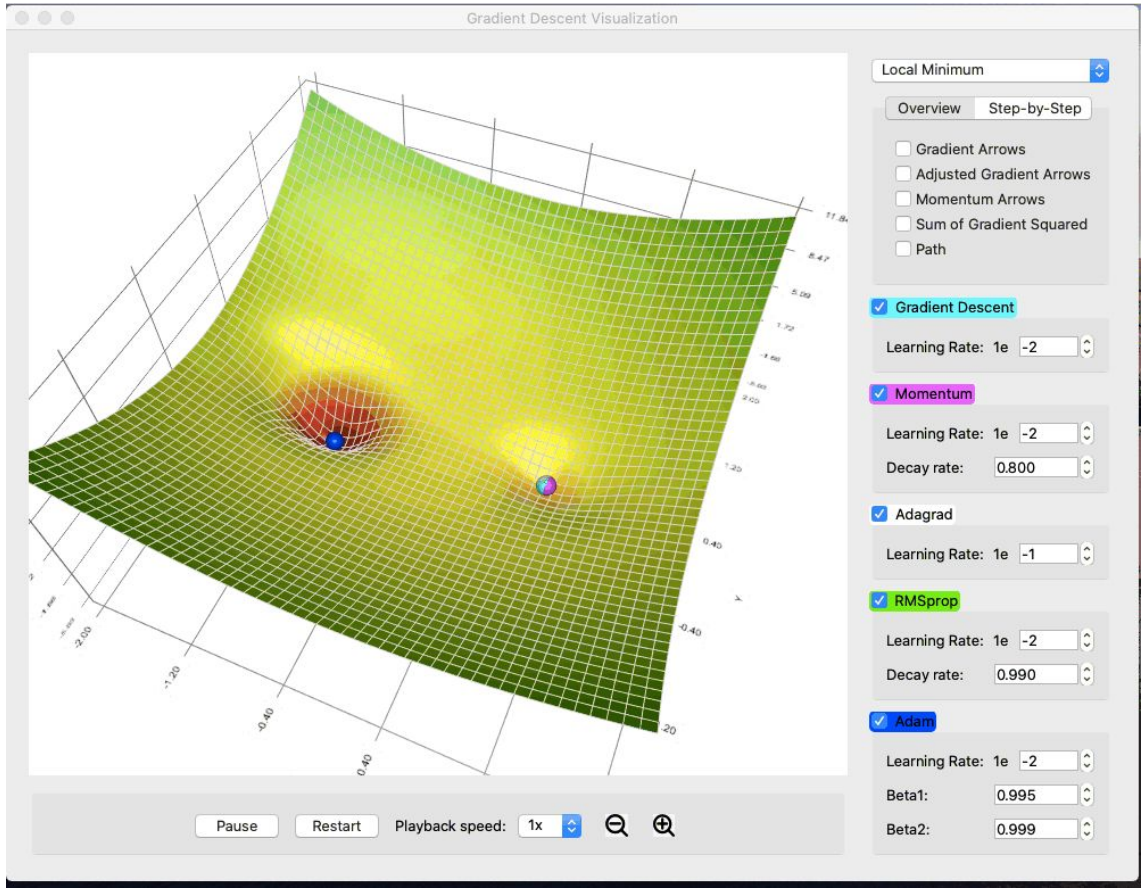
- Instead of 1 sample or all samples, take a **small group** (like 32 or 64).
- Compute average error for that group.
- Update weights.
- Repeat with the next mini-batch until the whole dataset is used → **1 epoch.**



https://medium.com/data-science/deep-learning-optimizers-436171c9e23f

Variants of Stochastic Gradient Descent (SGD)

- SGD with Momentum
  - remembers past direction → makes learning smoother
- Adagrad
  - Instead of one learning rate for everything, it gives each weight its own learning rate.
  - Weights that need bigger updates get them, while stable ones move less.
- RMSprop
  - Adagrad slows down too much over time.
  - RMSprop fixes this by keeping learning rates from becoming too small.
- Adam
  - Combines the ideas of Momentum and RMSprop.
  - Learns fast, adapts well, and works great for deep learning.



ADAM is the most popular choice this days.

https://github.com/lilipads/gradient_descent_viz

Training

After choosing how the model learns using optimizers,

- ❏ we must decide **what data to learn from**
- ❏ **and how to check its learning**.

Training: Dataset Splitting & Evaluation

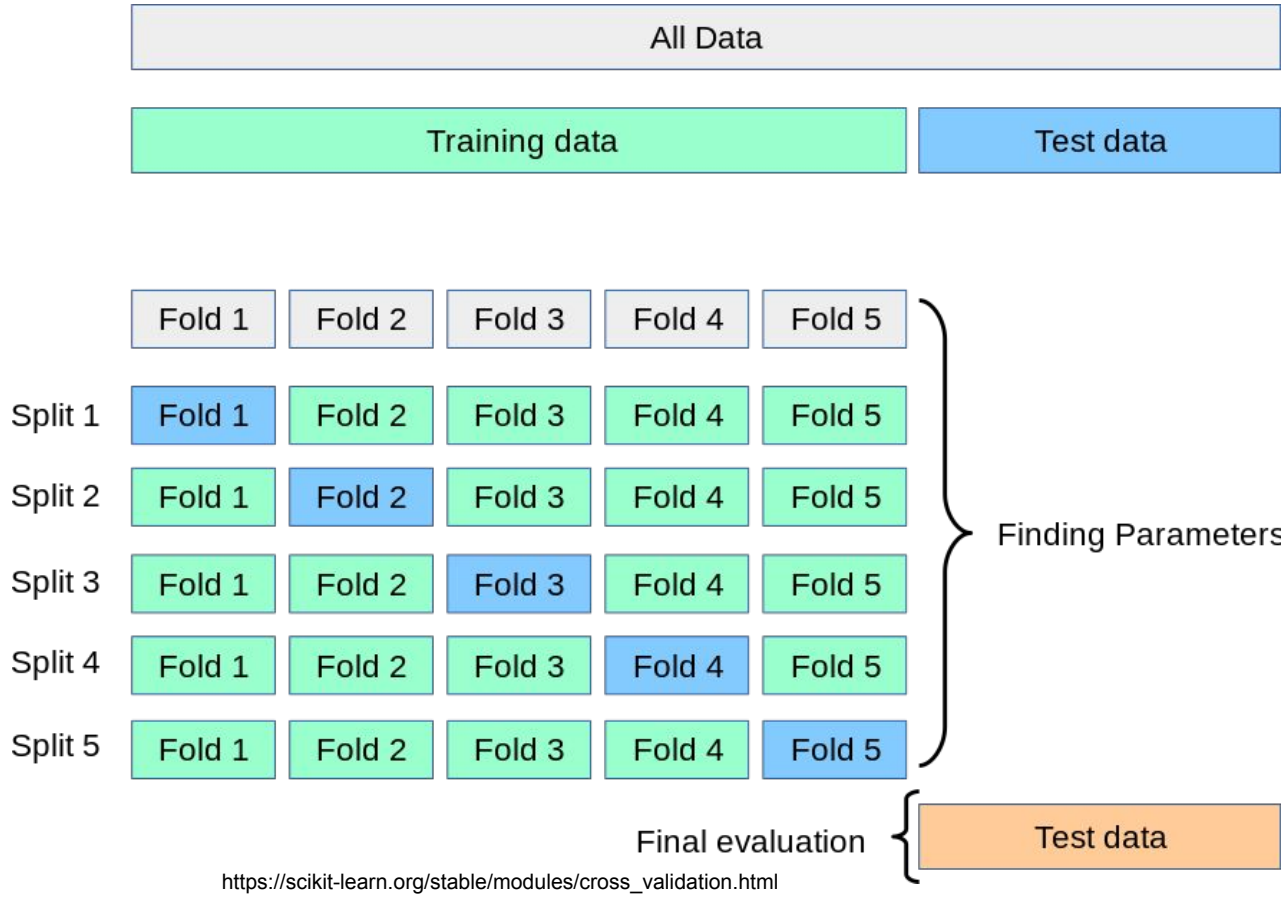
Dataset Splitting

- **Library**: `scikit-learn` (`train_test_split, KFold`)

- **Why**:
  - **Train set → model learns**
  - **Validation set → tune hyperparameters**
  - **Test set → unbiased evaluation**

Cross Validation

- Split data into k folds.
- Train on k-1 folds, validate on remaining fold. Repeat for all folds.



https://scikit-learn.org/stable/modules/cross_validation.html

Why: more reliable performance estimate, reduces overfitting risk.

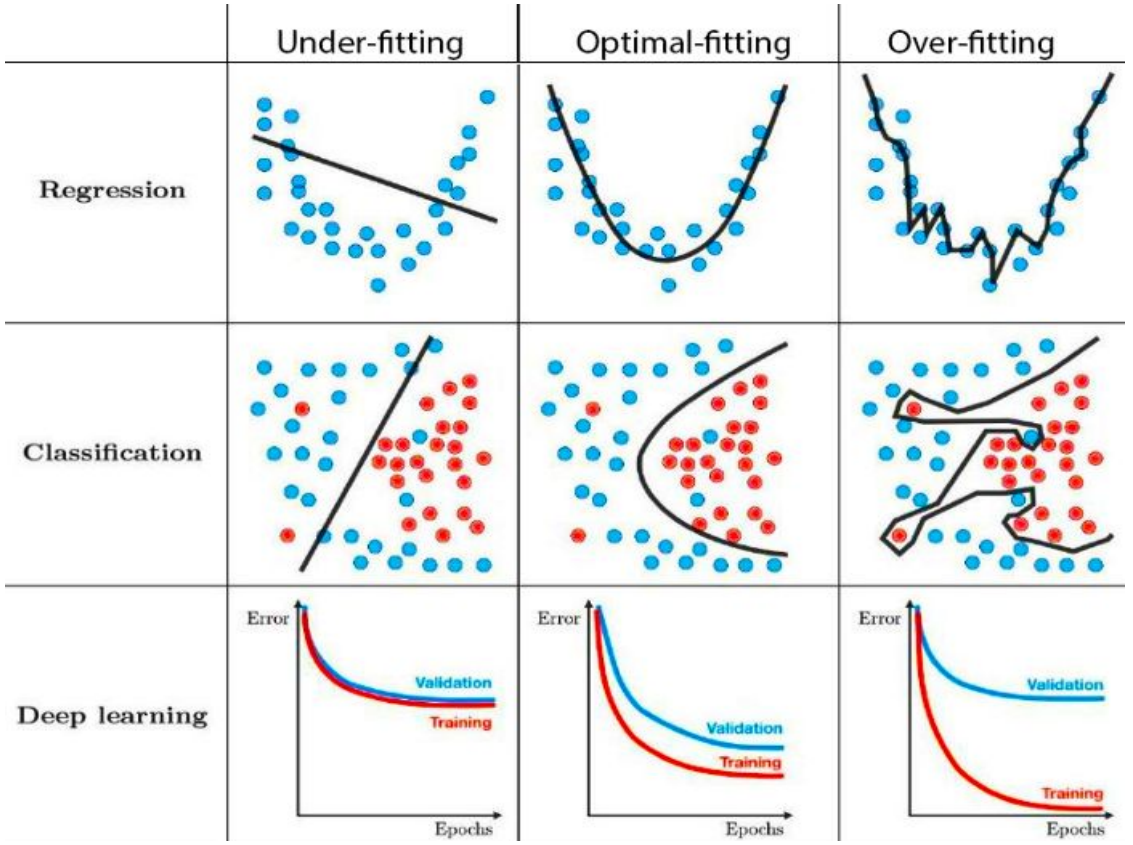
Training Challenges: Overfitting/Underfitting

What is Overfitting?

- Model learns the training data *too well*, including noise and random details.
- Performs well on training data but poorly on new, unseen data.

What is Underfitting?

- Model is **too simple** to learn the patterns in the data.
- Performs poorly on both training and new data → hasn't learned enough.



https://towardsdatascience.com/techniques-for-handling-underfitting-and-overfitting-in-machine-learning-348daa2380b9/

Why Overfitting/Underfitting happen?

Overfitting — model memorizes training data

- Model too complex (many parameters, layers, neurons)
- Small dataset (not enough examples)
- Trains too long without stopping

Underfitting — model fails to learn

- Model too simple (few parameters, shallow)
- Trains too little (stopped early)
- Features not informative

Solution

- Dropout
- L1/L2 regularization
- Early Stopping
- Data Augmentation
- Weight Initialization

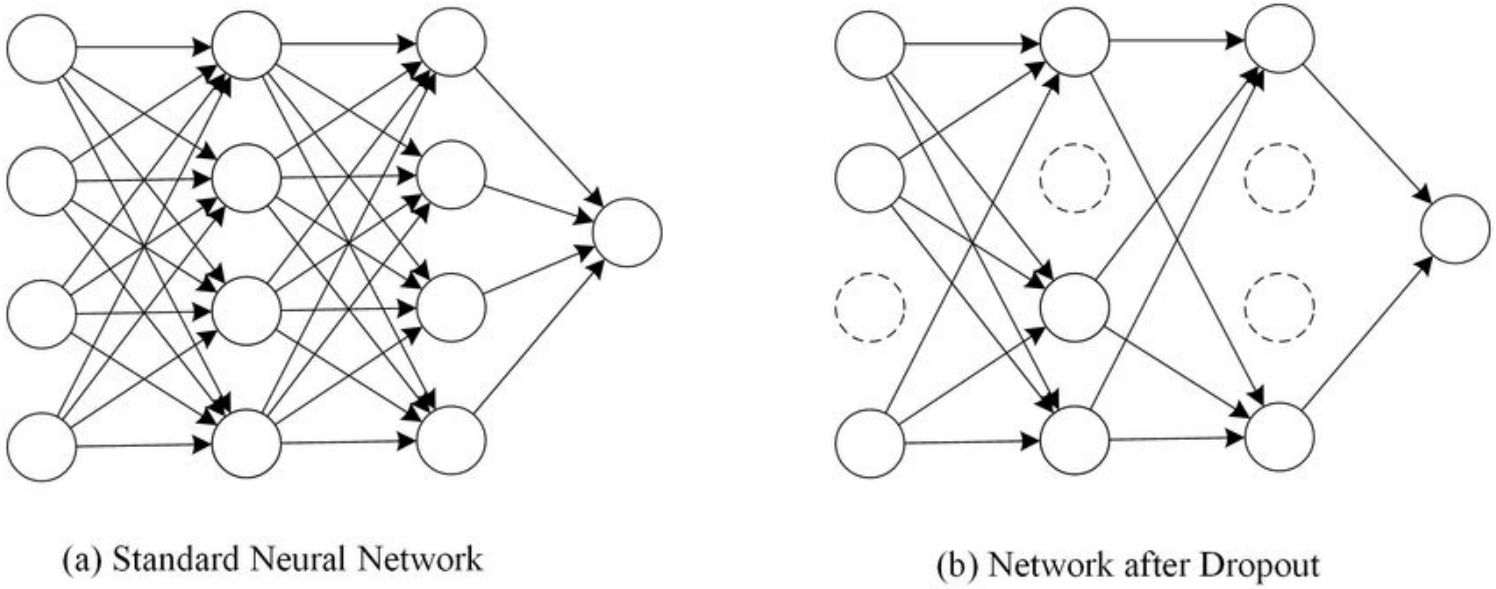
Solution: Dropout

What is Dropout?

- Regularization technique to reduce overfitting.
- Randomly **turns off** some neurons during training

Why Dropout helps?

- Forces the network to learn more robust patterns instead of memorizing.
- Makes models less overfitted and more generalizable



(a) Standard Neural Network

(b) Network after Dropout

https://www.researchgate.net/figure/Dropout-neural-network-model-a-is-a-standard-neural-network-b-is-the-same-network_fig3_309206911

Solution: Regularization

Regularization = Adding a penalty to the loss function to reduce overfitting

- Stops the model from depending too much on any one weight.
- Makes the model simpler (with smaller or fewer weights).

L1 Regularization (Lasso)

$$\text{Loss} = \text{Prediction Error} + \lambda \sum |w|$$

- Adds **absolute value** of weights to loss
- Pushes many weights to zero → removes unimportant features

L2 Regularization (Ridge)

$$\text{Loss} = \text{Prediction Error} + \lambda \sum w^2$$

- Adds **squared values of weights** to the loss
- Makes all weights smaller and smoother → none becomes too large

Solution: Early Stopping

What it is?

- Stop training when validation performance starts to drop

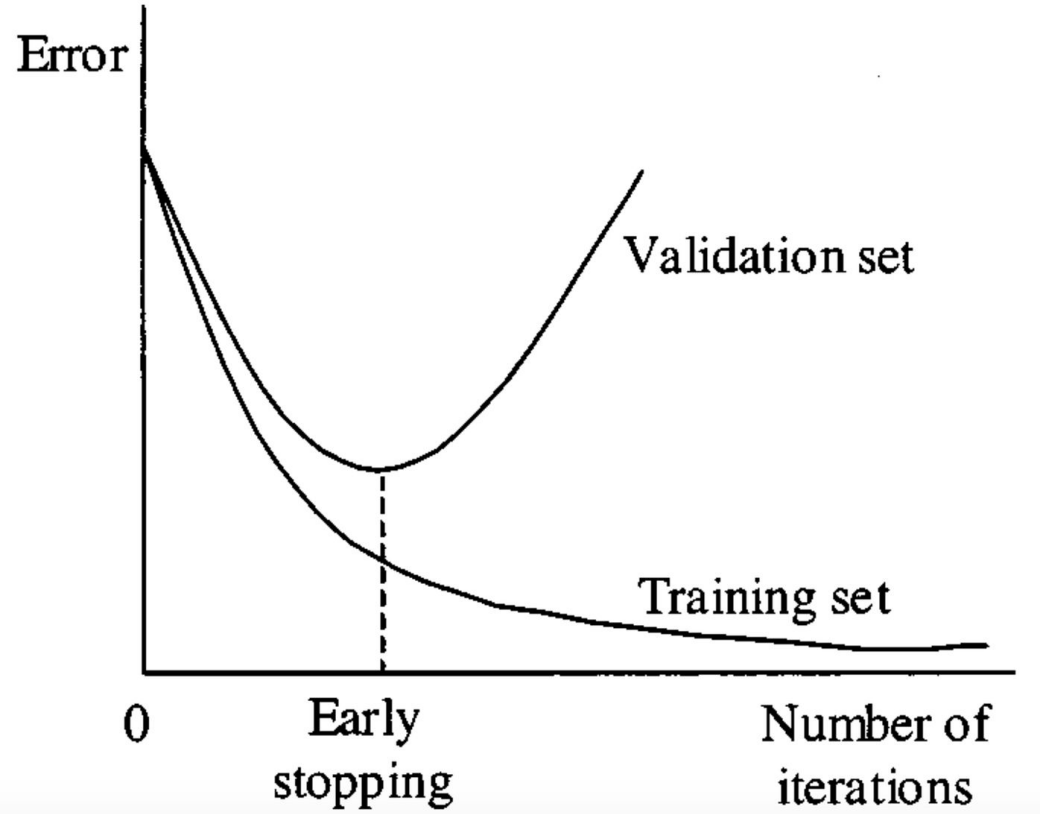
Why it helps:

- Prevents overfitting on training data
- Helps model generalize better to new data

Note: Requires **splitting** data into three sets:

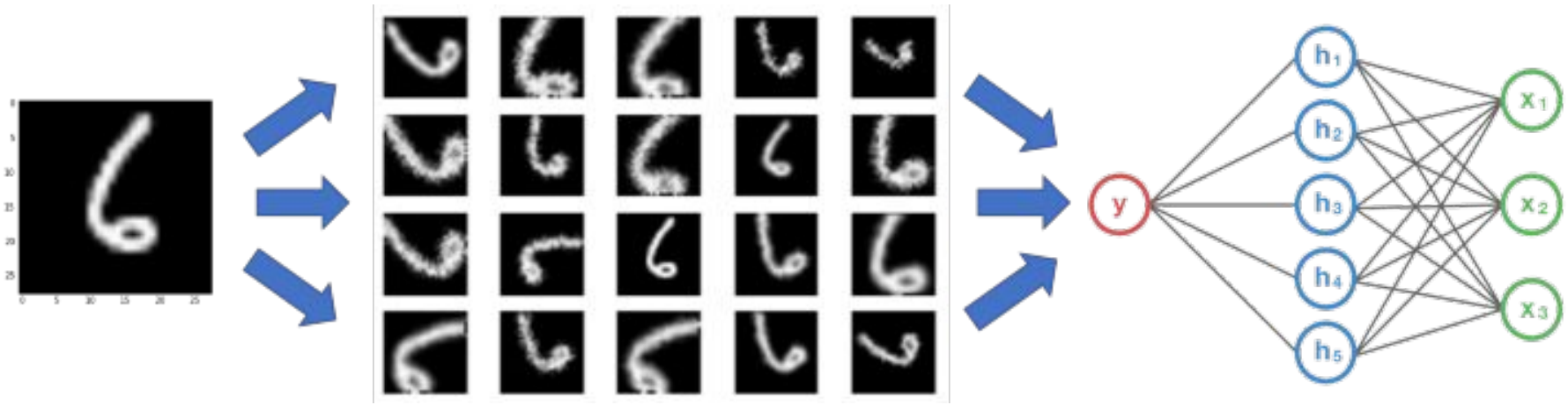
- **Train, Validation, and Test**.

Takeaway: Stop at the right time → less overfitting



https://www.researchgate.net/figure/Early-stopping-based-on-cross-validation_fig1_3302948

Solution: Data Augmentation



https://medium.com/data-science/rethinking-data-augmentations-a-causal-perspective-e0b7810579a7

What is Data Augmentation?

- Artificially increase the size of a training dataset.
- Creates new training examples by making small changes to existing data (like flipping, rotating, or cropping images).

Why it helps?

- Gives the model more varied data to learn from.
- Prevents the model from memorizing training data.
- Improves generalization to unseen data.

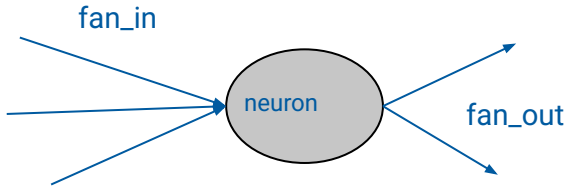
Takeaway: Makes training data richer → avoid overfitting.

Weight Initialization

- Weight should be small but should not be very small
- Weight should not be same, rather should have good variance.

Technique:

1. Uniform Distribution $W_{ij} \sim U\left(-\frac{1}{\sqrt{\text{fan\_in}}}, \frac{1}{\sqrt{\text{fan\_out}}}\right)$



2. Xavier/Gorat Distribution

   a. **Xavier Normal** $W_{ij} \sim \mathcal{N}\left(0, \frac{2}{\text{fan\_in} + \text{fan\_out}}\right)$

   b. **Xavier Uniform** $W_{ij} \sim U\left(-\frac{\sqrt{6}}{\sqrt{\text{fan\_in} + \text{fan\_out}}}, \frac{\sqrt{6}}{\sqrt{\text{fan\_in} + \text{fan\_out}}}\right)$

   } good with sigmoid activation function

3. He init

   a. **He uniform** $W_{ij} \sim U\left(-\frac{\sqrt{6}}{\sqrt{\text{fan\_in}}}, \frac{\sqrt{6}}{\sqrt{\text{fan\_in}}}\right)$

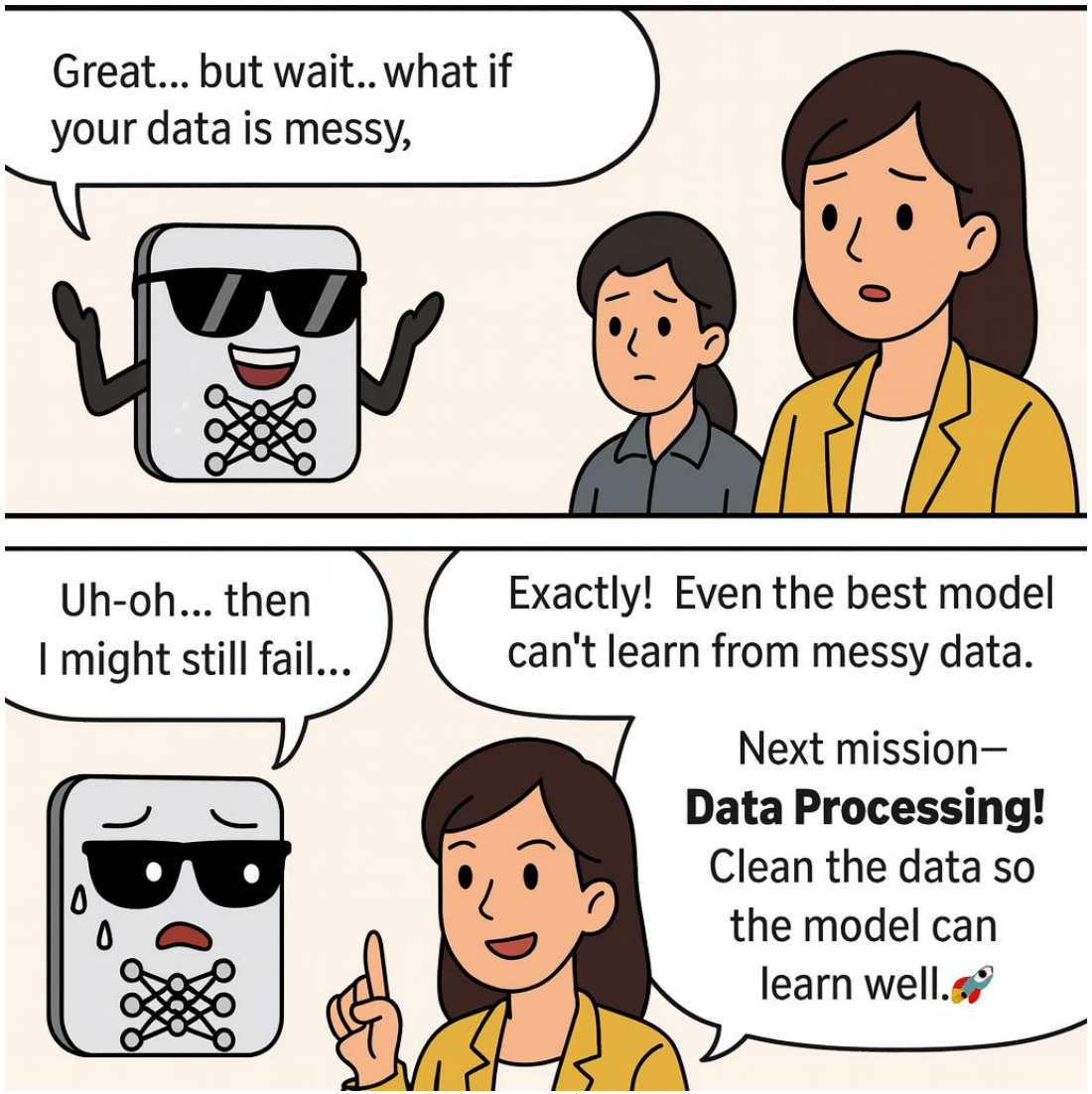
   b. **He Normal** $W_{ij} \sim \mathcal{N}\left(0, \frac{2}{\text{fan\_in}}\right)$

   } good with RELU activation function

https://www.youtube.com/watch?v=tMjdQLylyGI&list=PLZoTAELRMXVPGU70ZGsckrMdr0FteeRUi&index=15

Regularized… but Still Failing?

Next Mission: Data Processing!



generated by chatgpt

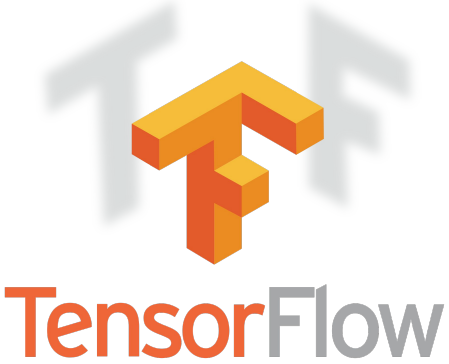
Data Preprocessing

- **Libraries**:
  - **pandas → data cleaning, handling missing values (fillna, dropna)**
  - **scikit-learn → normalization (MinMaxScaler)** →scales values to range [0,1].
    → **standardization** (**StandardScaler**)→rescales data to mean=0, std=1.

- **Why**: Models converge faster & avoid bias from different scales.

Hyperparameter Tuning Tools

- **Goal:** Find best values for learning rate, batch size, dropout, layers, etc.
- **Grid Search:** Tries all combinations (slow but thorough)
- **Random Search:** Tries random combinations (faster, often good enough)
- **Optuna:** Smart search — automates tuning, works with **PyTorch**, **TensorFlow**, etc.



Deep Learning Models & Transfer Learning

Types of Models

- **Fully Connected (Dense)** → each neuron connects to all in next layer; general tasks.
- **CNN (Convolutional Neural Networks)** → captures spatial patterns; best for images.
- **RNN (Recurrent Neural Networks)** → sequence learning; used for text, time series.
- **Transformers** → parallel processing, attention mechanism; state-of-the-art in NLP & vision.

Frameworks

- **TensorFlow** (Google) → scalable, production-ready.
- **PyTorch** (Meta) → flexible, research-friendly.
- **Keras** → high-level API, fast prototyping (runs on TensorFlow).

Transfer Learning

- Start with a **pretrained model** (e.g., ResNet, BERT).
- Fine-tune on your dataset.
- **Why it helps**:
  - Much **faster training** (because most features are already learned)
  - Works well even with **less data**
  - Often gives **better performance** than training from scratch

TensorFlow / PyTorch

TensorFlow:
Built by Google for large-scale deployment

Strong tools:

- **TensorFlow Serving** → tool to send trained models to servers.
- **TensorFlow Lite** → tool to shrink models to run on **phones, IoT devices**.
- **TensorFlow.js** → run in the **browser**.
- **Direct TPU support** → runs very well on Google's special AI hardware.

Great for mobile, web, production pipelines

PyTorch:

- PyTorch started as a **research-first framework** (built by Facebook AI).
- Focus: easy debugging, Pythonic, simple, flexible experiments → researchers loved it.
- But it didn't originally have strong deployment tools.
- TensorFlow was earlier for **production pipelines**, so companies adopted it.

Today: This is changing! PyTorch has **TorchServe, TorchScript, ONNX** for deployment. It's catching up fast. But TensorFlow still has more tools integrated for production at giant scale.

Thank You Any Questions?

Time for exercise

DRESDEN concept SCIENCE AND INNOVATION CAMPUS

HZDR