Gradient Decent

CT-466 | Week 4 - Lecture 9

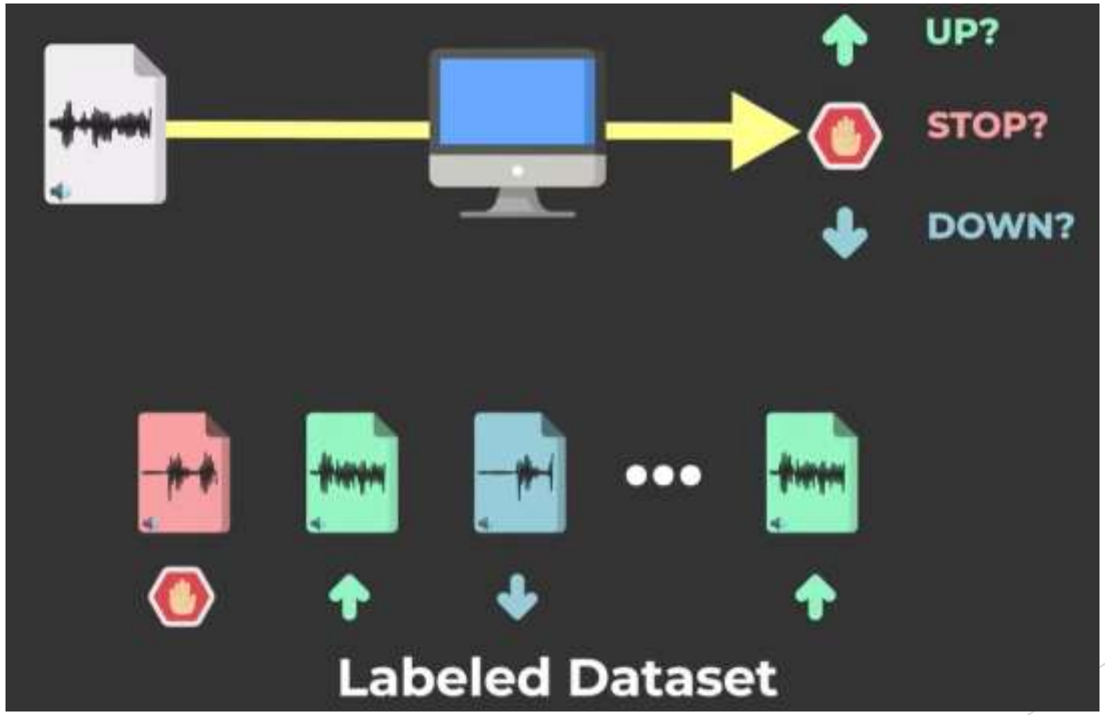
Instructor: Mehar Fatima Shaikh

Gradient descent

- Gradient descent is the process by which machines learn complex tasks, such as generating new faces, playing advanced strategy games like Dota, and recognizing speech commands. At its core, gradient descent is an optimization algorithm used to minimize the cost function of a model by adjusting its parameters (weights and biases).

Gradient descent

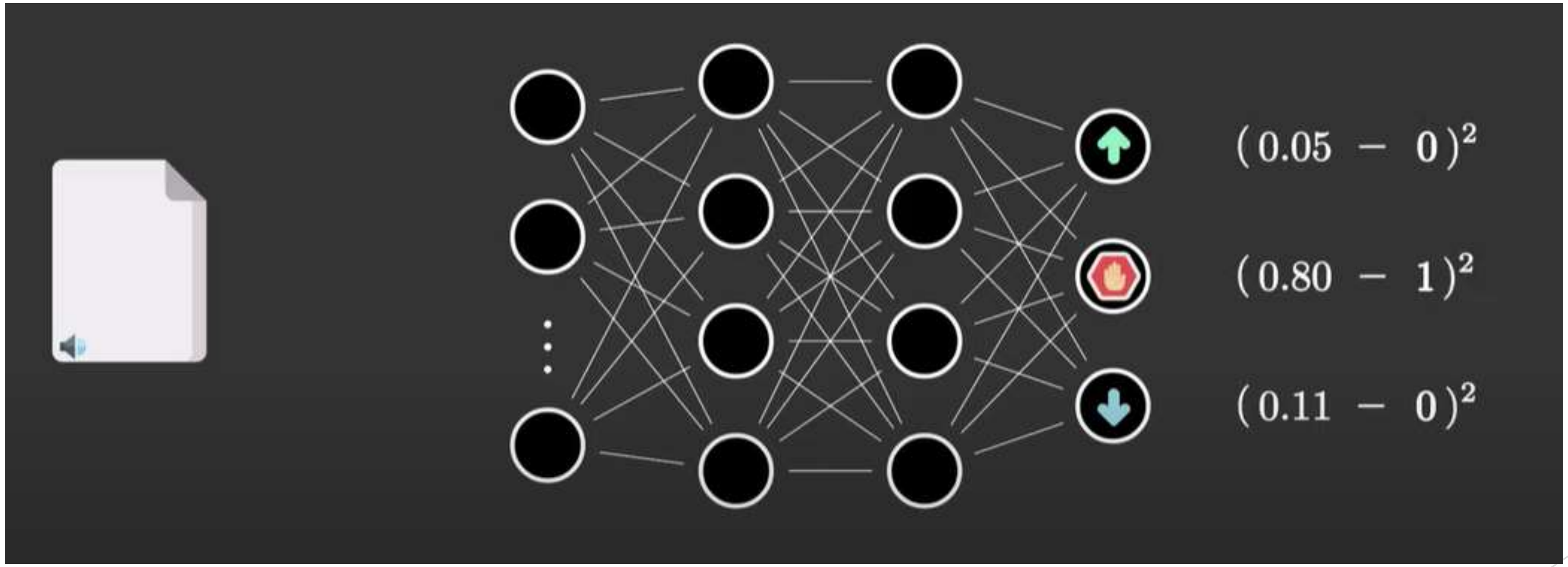
- Suppose we want to train a neural network to recognize three spoken commands from an audio dataset. The input is the audio data, and the output is a vector of size 3. Each element of this vector represents the probability that the audio corresponds to one of the three commands. The weights of the neural network are initially unknown and must be learned.



Gradient descent

- To measure how well the neural network performs, we define a cost function. For each training example, the network's prediction is compared to the ideal (labeled) output. We calculate the difference, square it, and then sum it. This gives us the cost for a single training example. By summing over all training examples, we obtain the overall cost function.
- Mathematically, if y is the true label and ŷ is the predicted output, then the cost for one example can be written as:
- $\textbf{\textit{Cost}} = (y - \hat{y})^2$
- And the overall cost function J(θ) for all training examples is:
- $J(\theta) = \Sigma\, (y_i - \hat{y}_i)^2$

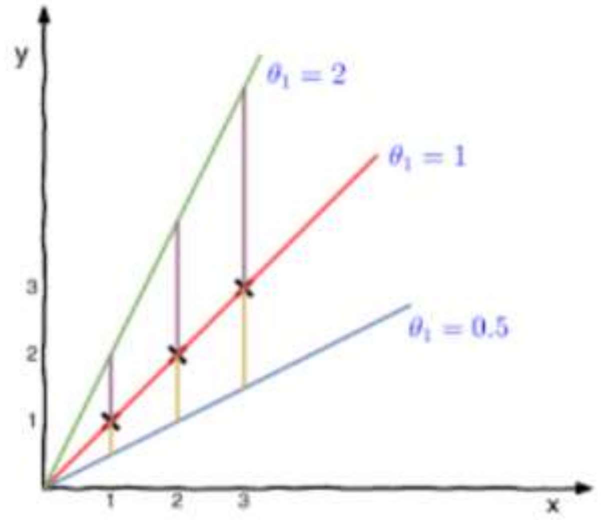
Gradient descent

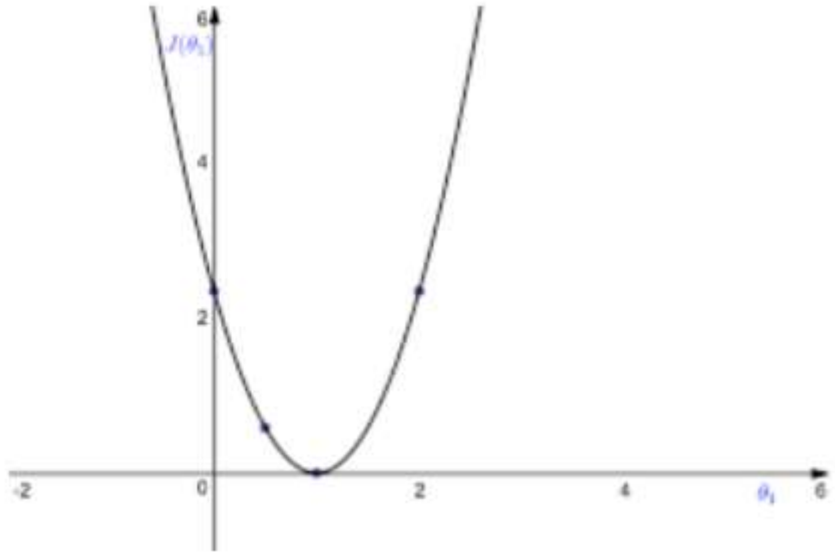


Cost Function

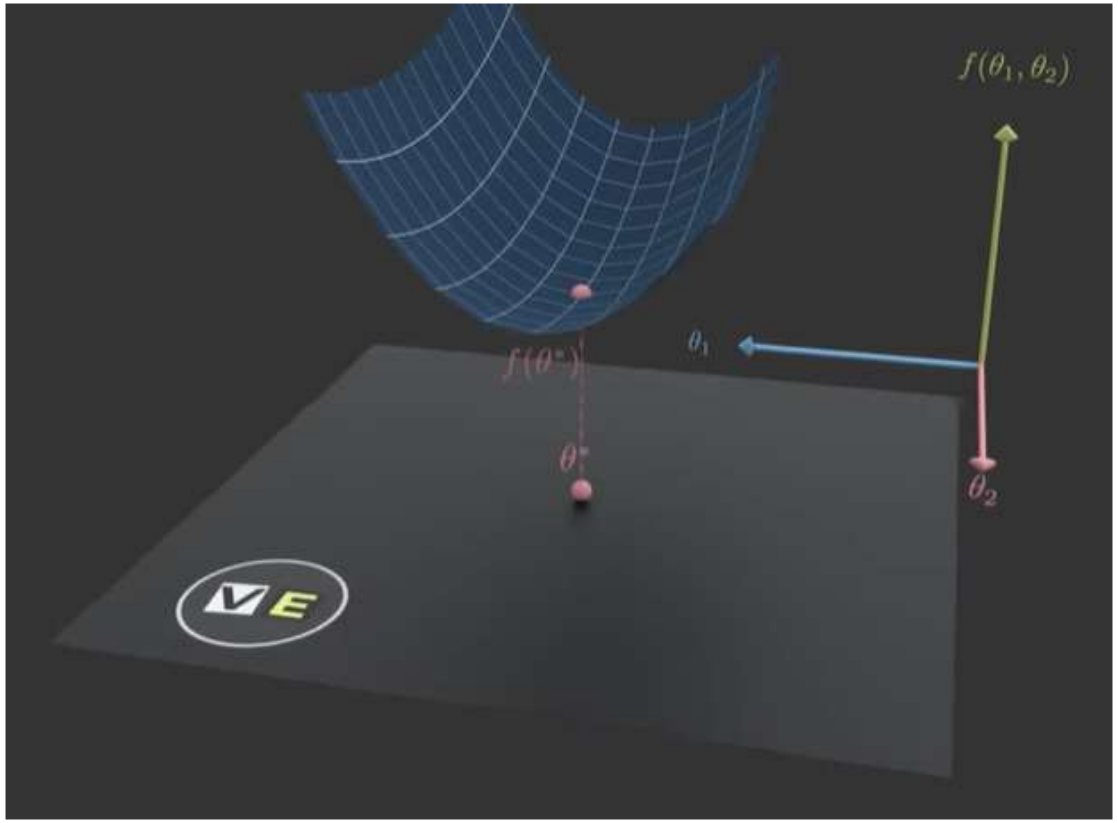
- A loss function in a neural network evaluates how different the predicted output is from the actual target value.
- It measures the error and guides the model to improve its predictions using backpropagation and gradient descent.
- Suppose $\theta_0 = 0$

θ_1	x	Actual y	Predicted y	$J(\theta_1)$
0.5	1	1	0.5	**0.58**
	2	2	1	
	3	3	1.5	
1	1	1	1	**0**
	2	2	2	
	3	3	3	
2	1	1	2	**2.34**
	2	2	4	
	3	3	6	

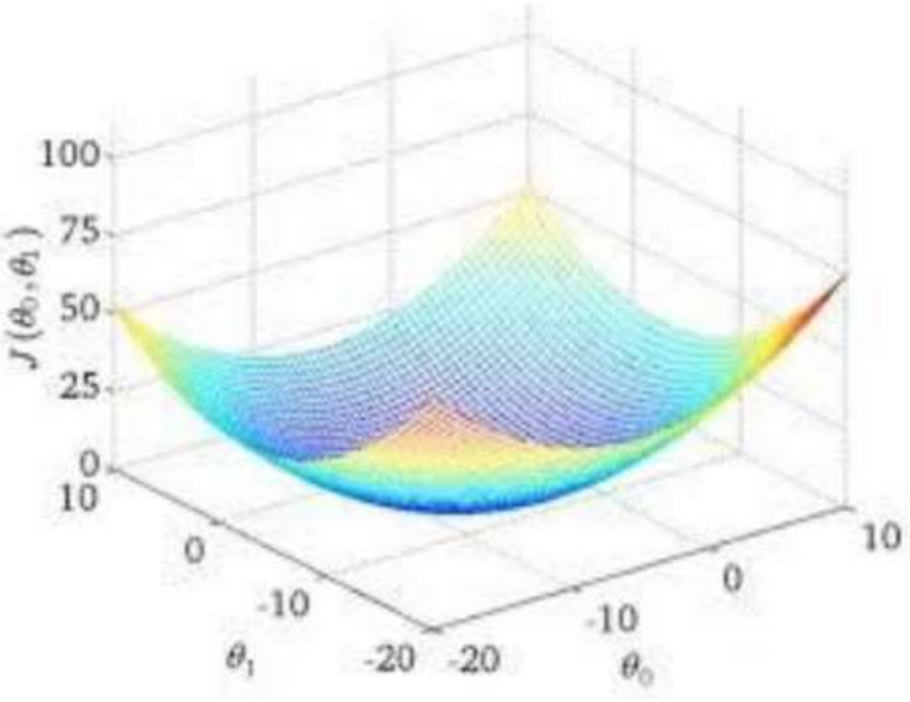


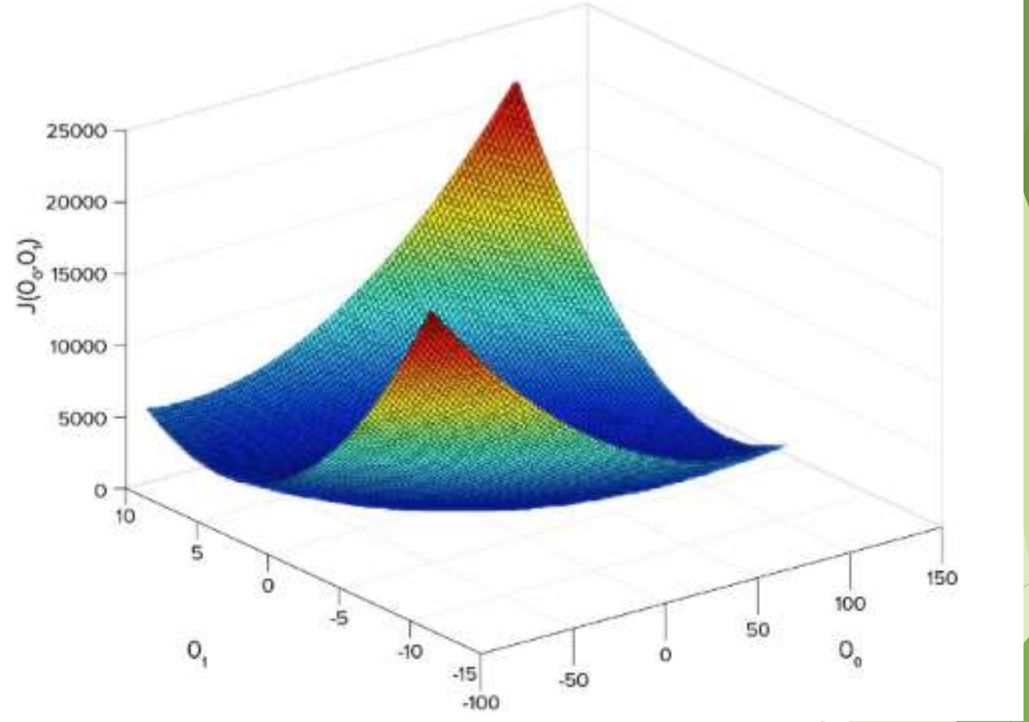


Cost Functions

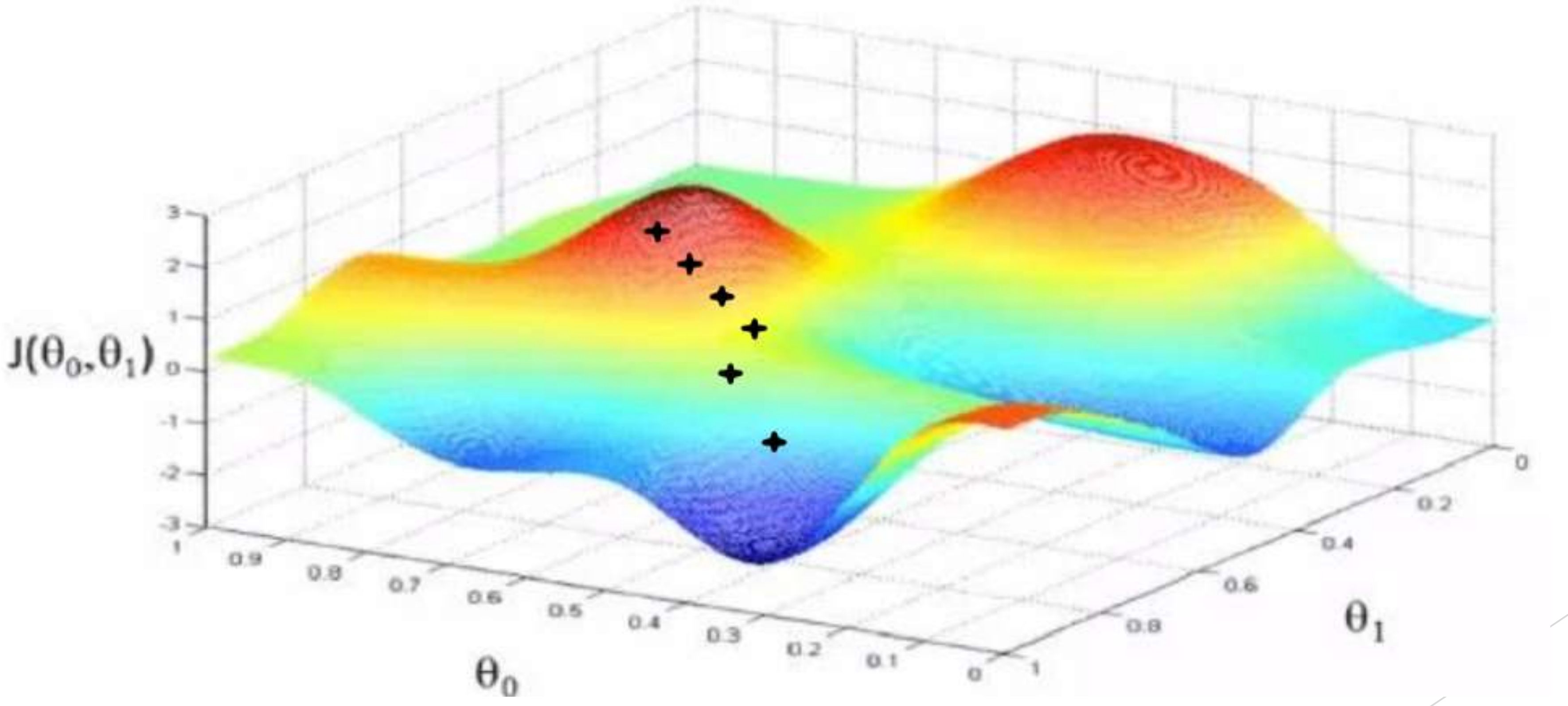


Cost function



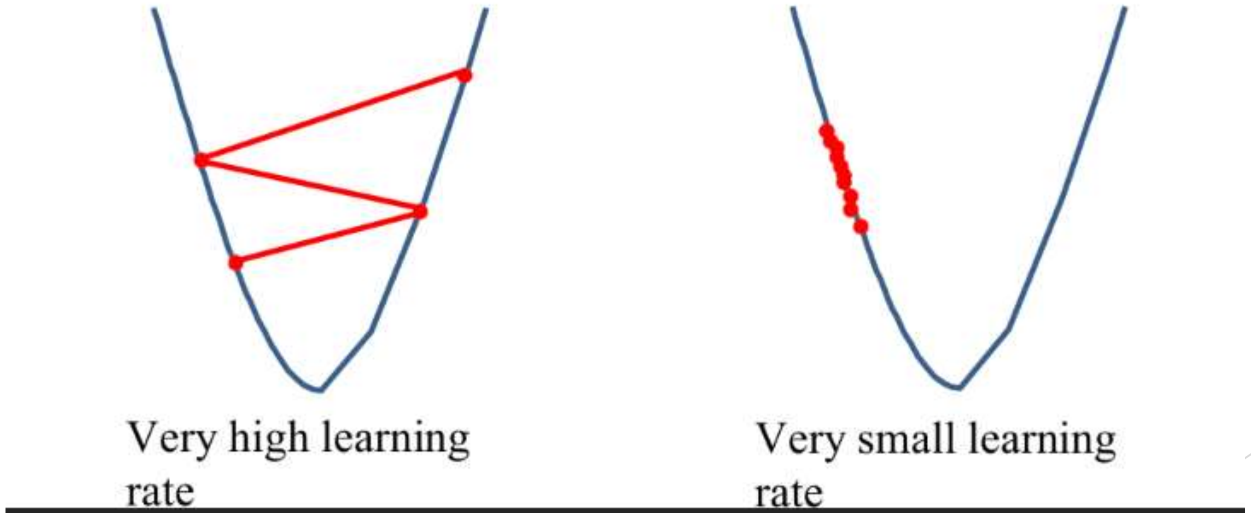


Cost function



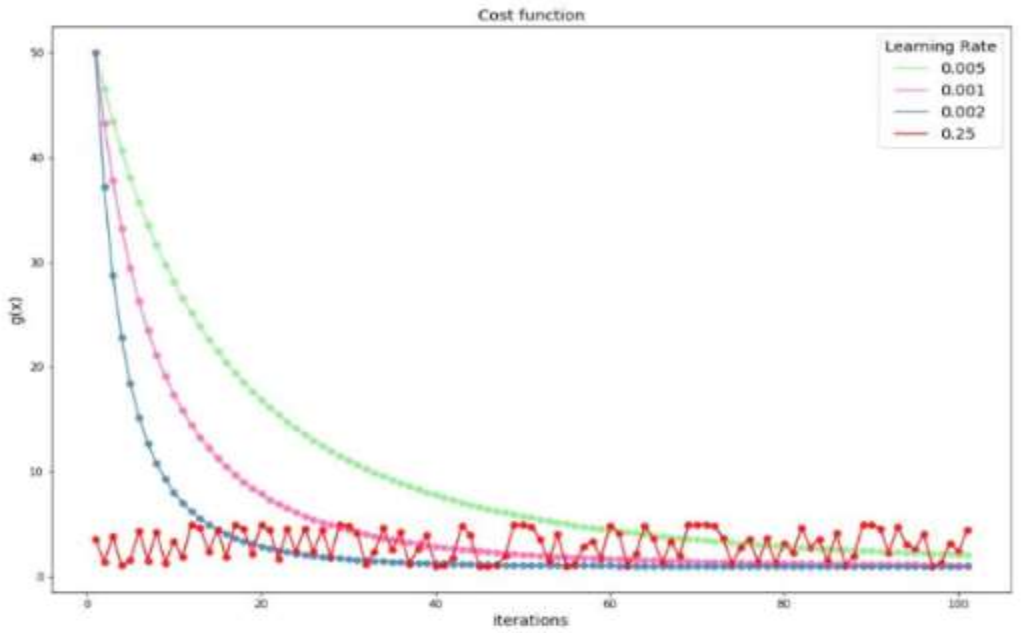
Effect of Learning Rate on Cost

- **What is Learning Rate?**
- The learning rate αcontrols the **step size** in **gradient descent**.
- Update rule:
- $w_{new} = w_{old} - \alpha \cdot \nabla J(w)$
- where $\nabla J(w)$is the gradient of the cost function.



Effect of Learning Rate on Cost

- **Small Learning Rate (α too small)**
  - Cost decreases **slowly**.
  - Convergence is guaranteed but **training is very slow**.
  - May get stuck in **local minima or plateaus**.
- **Large Learning Rate (α too large)**
  - Cost decreases **fast at first** but may **overshoot the minimum**.
  - Training can oscillate or even diverge (cost increases instead of decreasing).
  - Network may fail to converge.
- **Optimal Learning Rate**
  - Strikes a balance:
    - Fast convergence.
    - Smooth decrease in cost.
    - Avoids oscillations.



Global Minimum

- The absolute lowest point of the function across the entire domain.
- For a cost function $J(\theta)$,the global minimum is where:
- $J(\theta^*) \leq J(\theta)\ \forall\theta$
- In ANN $\longrightarrow$ the point where the cost (error) is the smallest possible.
- Example: The bottom of the deepest valley in a mountain range.

Local Minimum

- A point where the function has a lower value than all nearby points, but it may not be the lowest overall.
- Mathematically:
- $J(\theta^*) \leq J(\theta)\ \forall \theta\ \textit{in a small neighborhood around}\ \theta^*$
- In ANN → training may get stuck in a local minimum instead of reaching the global one.
- Example: A smaller valley inside the mountain range, but not the deepest one.

Global Maximum

- The highest point of the function over the entire domain.
- For a reward function $R(\theta)$:
- $R(\theta^*) \geq R(\theta)\ \forall\theta$
- Example: The tallest mountain peak in the range.

Local Maximum

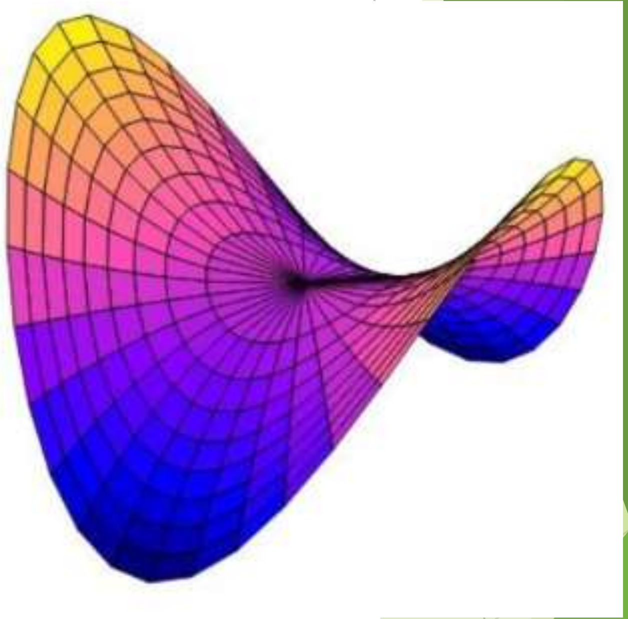
- A peak that is higher than surrounding points but not the highest overall.
- Formally:
- $R(\theta^*) \geq R(\theta)\ \forall\theta\ in\ a\ neighborhood\ of\ \theta^*$
- Example: A smaller hilltop next to the tallest peak.

In Neural Networks

- Loss functions often have many local minima and saddle points due to non-linearity and high-dimensional weight spaces.
- Global minimum is ideal (lowest error).
- But in practice:
  - Getting stuck in local minima is less of a problem in deep learning (many local minima are good enough).
  - Saddle points (flat regions) are more problematic, as they slow down gradient descent.

The Monkey Saddle

- A hyperbolic paraboloid is a saddle-shaped surface.
  Its general equation is:
- $z = \frac{x^2}{a^2} - \frac{y^2}{b^2}$
- Along the x-axis, it curves upward (like a parabola).
- Along the y-axis, it curves downward (like an inverted parabola).
- The result is a saddle point at the origin.
- **Geometric Properties**
  - Doubly ruled surface → can be generated by two families of straight lines.
  - Saddle point at the center → neither a maximum nor a minimum.

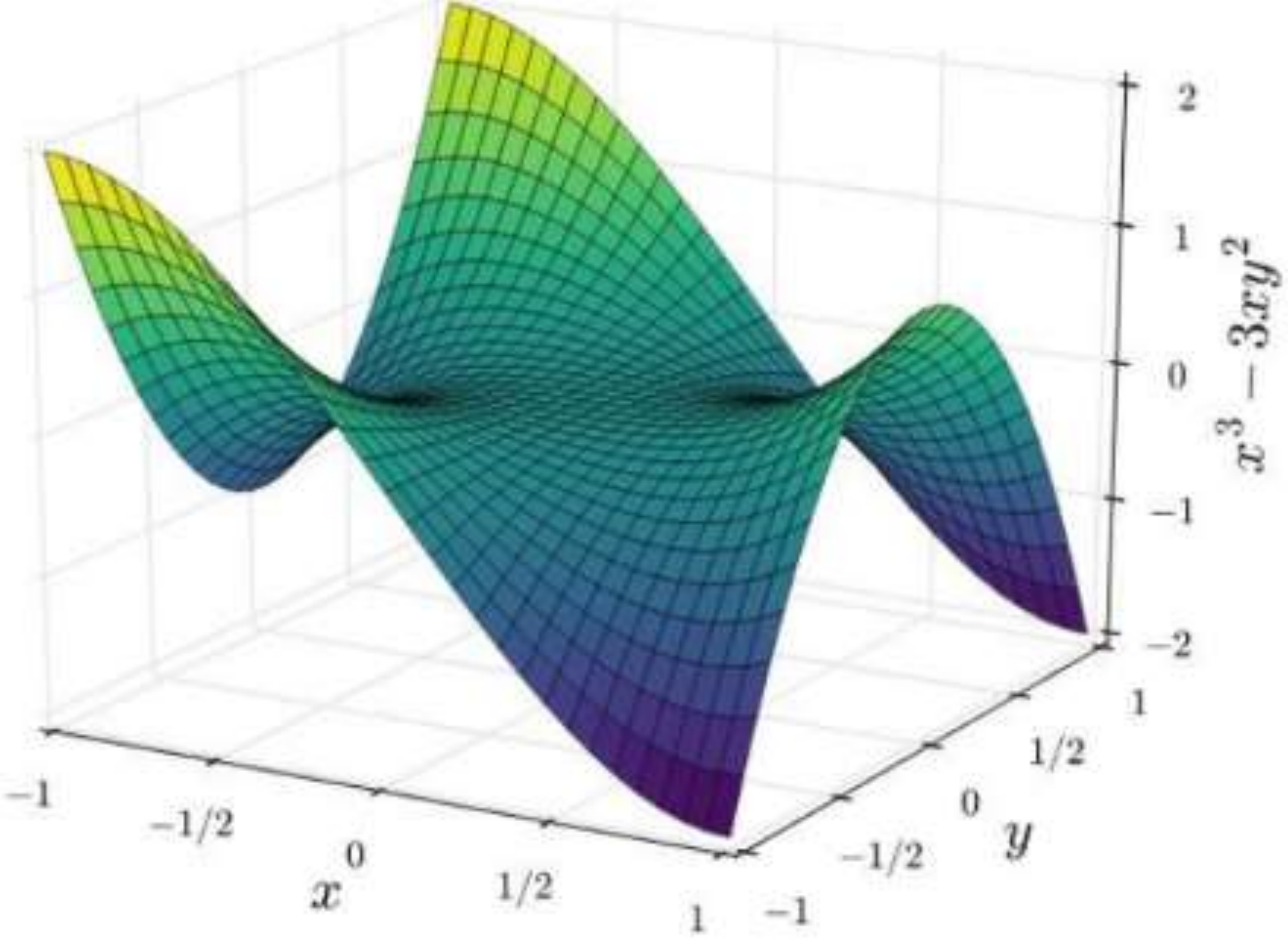


Connection to ANN / Optimization

- In machine learning and neural networks:
- Loss functions often have non-convex surfaces like this.
- The saddle point represents a location where the gradient = 0, but it’s not an optimum (neither min nor max).
- Gradient descent can get “stuck” around saddle points, making training difficult.
- This explains why optimization in deep learning is harder than in convex problems (like linear regression).

Monkey Saddle

- The Monkey Saddle is defined by the e
- $z = x^3 - 3xy^2$
- It's called the Monkey Saddle because two legs and a tail (three "directions" t
- **Geometric Properties**
- At the origin (0, 0, 0) ,the surface has a
- Unlike the usual saddle (which curves Saddle curves in three directions.
- This makes it more complex and symm
- In terms of partial derivatives:
- $\frac{\partial z}{\partial x} = 3x^2 - 3y^2$
- $\frac{\partial z}{\partial y} = -6xy$
- At (0,0), both are zero → critical point
- But it's not a minimum or maximum, i



The Monkey Saddle

Connection to Neural Networks / Optimization

- In deep learning optimization landscapes, we encounter saddle points where gradients vanish.
- The Monkey Saddle is a more complex version: multiple directions of “escape.”
- It illustrates why gradient descent may slow down, gradients are near zero, but the point isn’t optimal.

Vanilla Gradient Descent

- “Vanilla” means the **simplest/basic version** (without modifications).
- It is an iterative optimization algorithm used to minimize a **cost function** $J(\theta)$by updating parameters in the direction of the **negative gradient**.
- For parameters θ(e.g., weights in a neural network):
- $\theta := \theta - \alpha \cdot \nabla J(\theta)$
- Where:
- α= learning rate (step size).
- $\nabla J(\theta)$ =gradient of the cost function w.r.t. parameters.

Vanilla Gradient Descent **Steps**

- Initialize parameters θ randomly.
- Compute the gradient of cost function $J(\theta)$.
- Update parameters using the rule above.
- Repeat until convergence (cost stabilizes or falls below threshold).

Types of Vanilla Gradient Descent

- Batch Gradient Descent: Uses the entire dataset to compute gradient each step.
- Stochastic Gradient Descent (SGD): Uses one sample at a time.

Batch Gradient Descent:

- In batch gradient descent, at each iteration we compute the gradient of the cost function using all training examples:
- $\nabla J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla J_i(\theta)$
- where $m$is the number of training examples.
- Accurate gradient.
- Very slow when m is large (big datasets).

Stochastic Gradient Descent (SGD)

- Instead of using all examples, SGD updates parameters using just one randomly chosen training example at each iteration:
- $\theta := \theta - \alpha \cdot \nabla J_i(\theta)$
- where i is randomly sampled from the dataset.
- Much faster → immediate updates after each example.
- Can escape local minima due to noise.
- Noisy updates → path to the minimum “zig-zags” instead of smooth.

Limitations of Vanilla Gradient

- Sensitive to learning rate (too large → divergence, too small → slow).
- Can get stuck in local minima or saddle points.
- Inefficient for very large datasets (Batch GD).

Adaptive Gradient Descent

- In vanilla gradient descent, we use a single global learning rate α for all parameters.
- But in adaptive methods, the learning rate is adjusted independently for each parameter (or feature).
- In problems with sparse data (e.g., text represented as one-hot vectors, or high-dimensional image pixels):
  - Some features occur frequently (dense features).
  - Others occur rarely (sparse features).
- If we use the same learning rate for all, we risk:
  - Over-updating frequent features.
  - Under-updating rare features.
- Adaptive methods solve this by giving:
- Smaller learning rates for frequent features.
- Larger learning rates for rare features.

Momentum Gradient Descent

- Problem with Vanilla Gradient Descent
- In vanilla gradient descent, updates depend only on the current gradient.
- This can cause:
    - Slow convergence in ravines (narrow, steep regions).
    - Oscillations when gradients alternate directions (e.g., zig-zagging in cost valleys).
- Momentum Idea
- Momentum adds memory of past gradients to the update.
- Think of it like a ball rolling down a hill:
    - It builds speed (momentum) in the right direction.
    - Ignores small oscillations.
    - Accelerates towards the minimum.

Momentum Gradient Descent

- We maintain a velocity term v:
- $v_t = \beta v_{t-1} + (1 - \beta)\nabla J(\theta_t)$
- $\theta_{t+1} = \theta_t - \alpha v_t$
- Where:
- v_t =velocity (exponentially smoothed past gradients).
- β= momentum coefficient (e.g., 0.9).
- α= learning rate.
- $\nabla J(\theta_t)$ =gradient at step t.
- **Effects**
- Damps oscillations in directions where gradient signs keep flipping.
- Accelerates convergence in consistent gradient directions.
- Works especially well in ravine-like cost surfaces (steep in one direction, shallow in another).