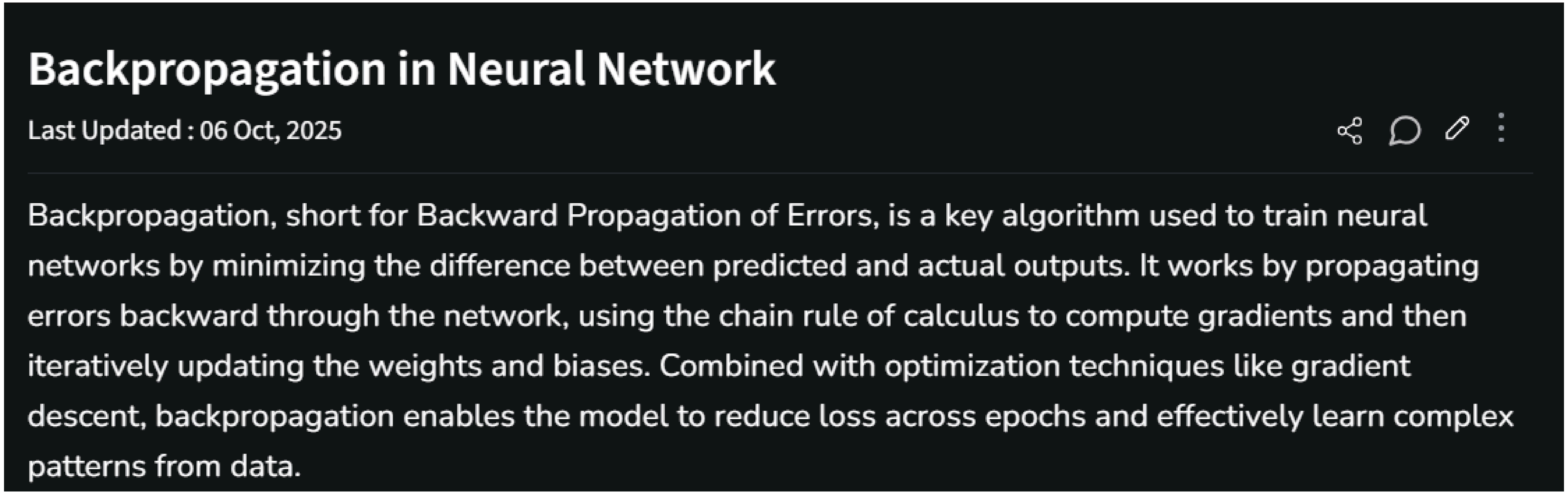
ARTIFICIAL NEURAL NETWORKS

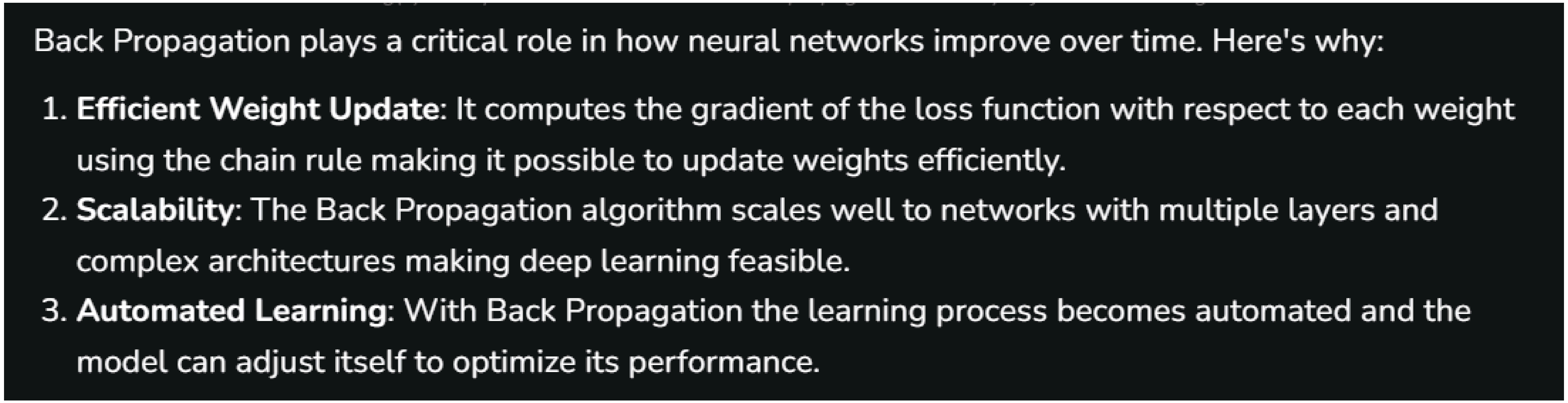
Architecting Intelligence



Backpropagation in Neural Network

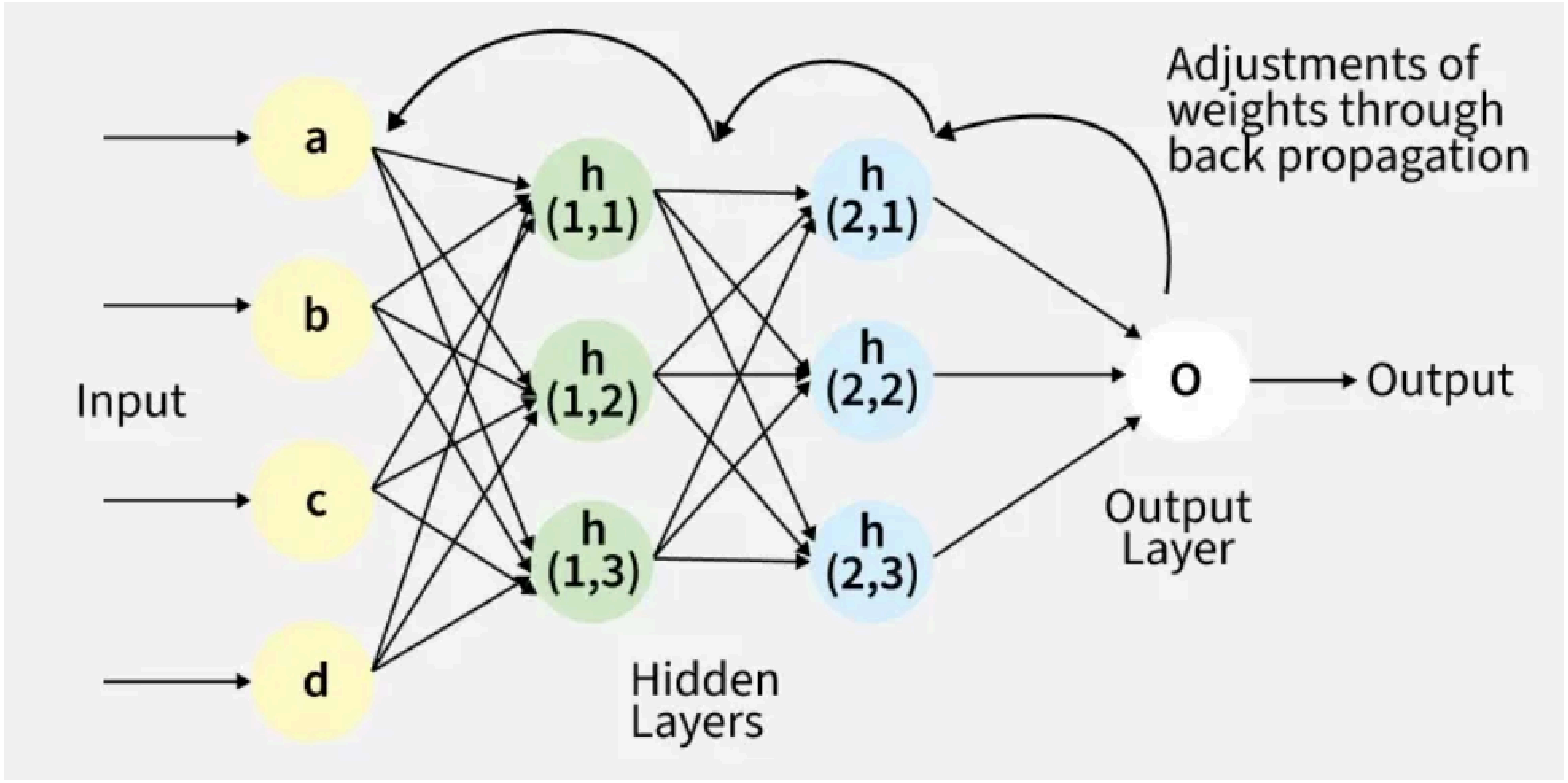
Last Updated : 06 Oct, 2025

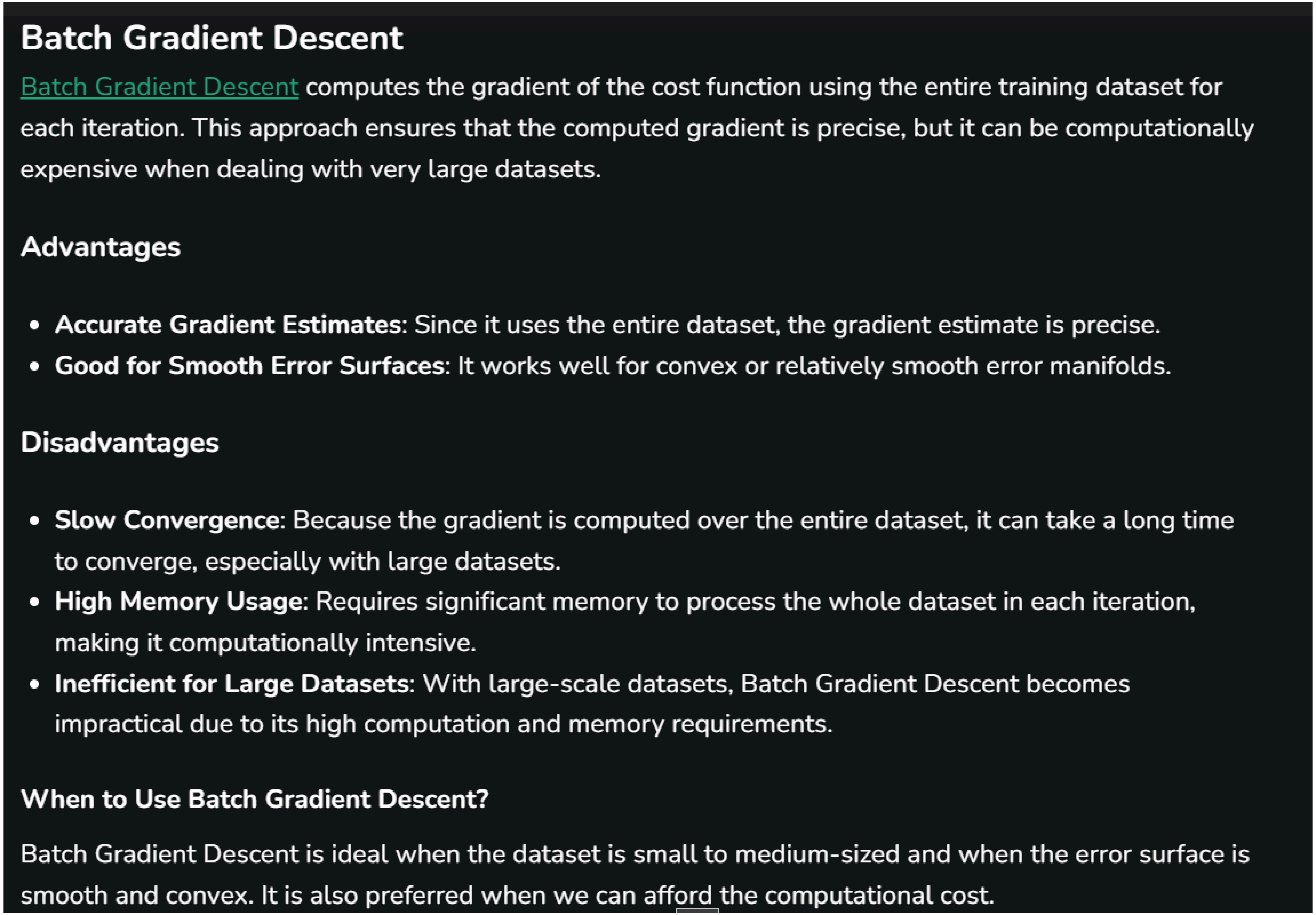
Backpropagation, short for Backward Propagation of Errors, is a key algorithm used to train neural networks by minimizing the difference between predicted and actual outputs. It works by propagating errors backward through the network, using the chain rule of calculus to compute gradients and then iteratively updating the weights and biases. Combined with optimization techniques like gradient descent, backpropagation enables the model to reduce loss across epochs and effectively learn complex patterns from data.



Back Propagation plays a critical role in how neural networks improve over time. Here's why:

1. **Efficient Weight Update**: It computes the gradient of the loss function with respect to each weight using the chain rule making it possible to update weights efficiently.
2. **Scalability**: The Back Propagation algorithm scales well to networks with multiple layers and complex architectures making deep learning feasible.
3. **Automated Learning**: With Back Propagation the learning process becomes automated and the model can adjust itself to optimize its performance.


Adjustments of weights through back propagation
a
b
c
d
Input
h (1,1)
h (1,2)
h (1,3)
h (2,1)
h (2,2)
h (2,3)
Hidden Layers
O
Output Layer
Output



Batch Gradient Descent

Batch Gradient Descent computes the gradient of the cost function using the entire training dataset for each iteration. This approach ensures that the computed gradient is precise, but it can be computationally expensive when dealing with very large datasets.

Advantages

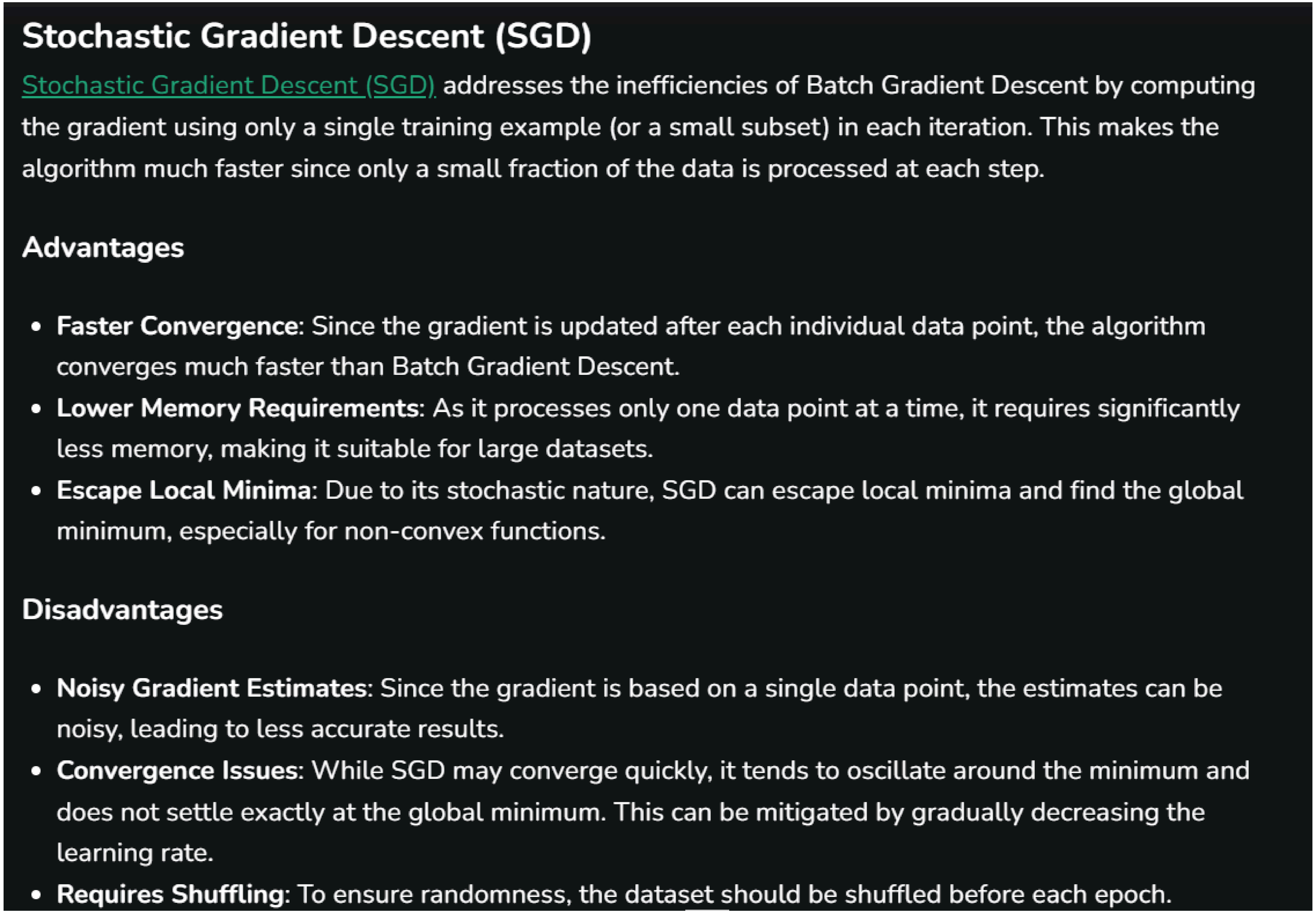
- **Accurate Gradient Estimates**: Since it uses the entire dataset, the gradient estimate is precise.
- **Good for Smooth Error Surfaces**: It works well for convex or relatively smooth error manifolds.

Disadvantages

- **Slow Convergence**: Because the gradient is computed over the entire dataset, it can take a long time to converge, especially with large datasets.
- **High Memory Usage**: Requires significant memory to process the whole dataset in each iteration, making it computationally intensive.
- **Inefficient for Large Datasets**: With large-scale datasets, Batch Gradient Descent becomes impractical due to its high computation and memory requirements.

When to Use Batch Gradient Descent?

Batch Gradient Descent is ideal when the dataset is small to medium-sized and when the error surface is smooth and convex. It is also preferred when we can afford the computational cost.



Stochastic Gradient Descent (SGD)

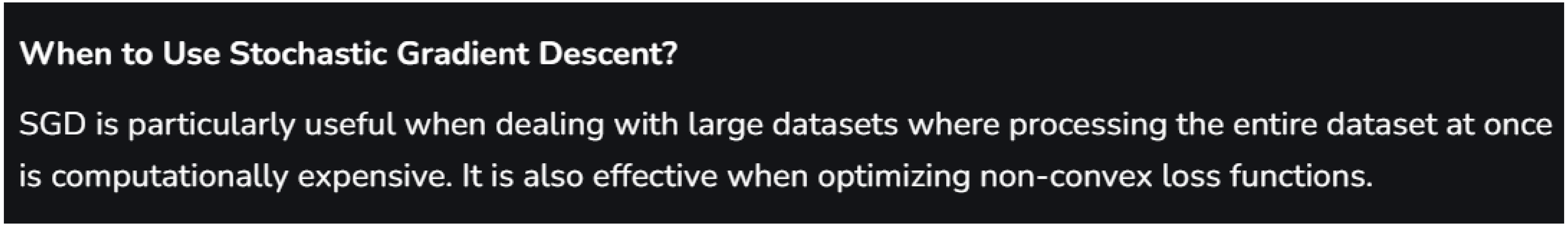
Stochastic Gradient Descent (SGD) addresses the inefficiencies of Batch Gradient Descent by computing the gradient using only a single training example (or a small subset) in each iteration. This makes the algorithm much faster since only a small fraction of the data is processed at each step.

Advantages

- **Faster Convergence**: Since the gradient is updated after each individual data point, the algorithm converges much faster than Batch Gradient Descent.
- **Lower Memory Requirements**: As it processes only one data point at a time, it requires significantly less memory, making it suitable for large datasets.
- **Escape Local Minima**: Due to its stochastic nature, SGD can escape local minima and find the global minimum, especially for non-convex functions.

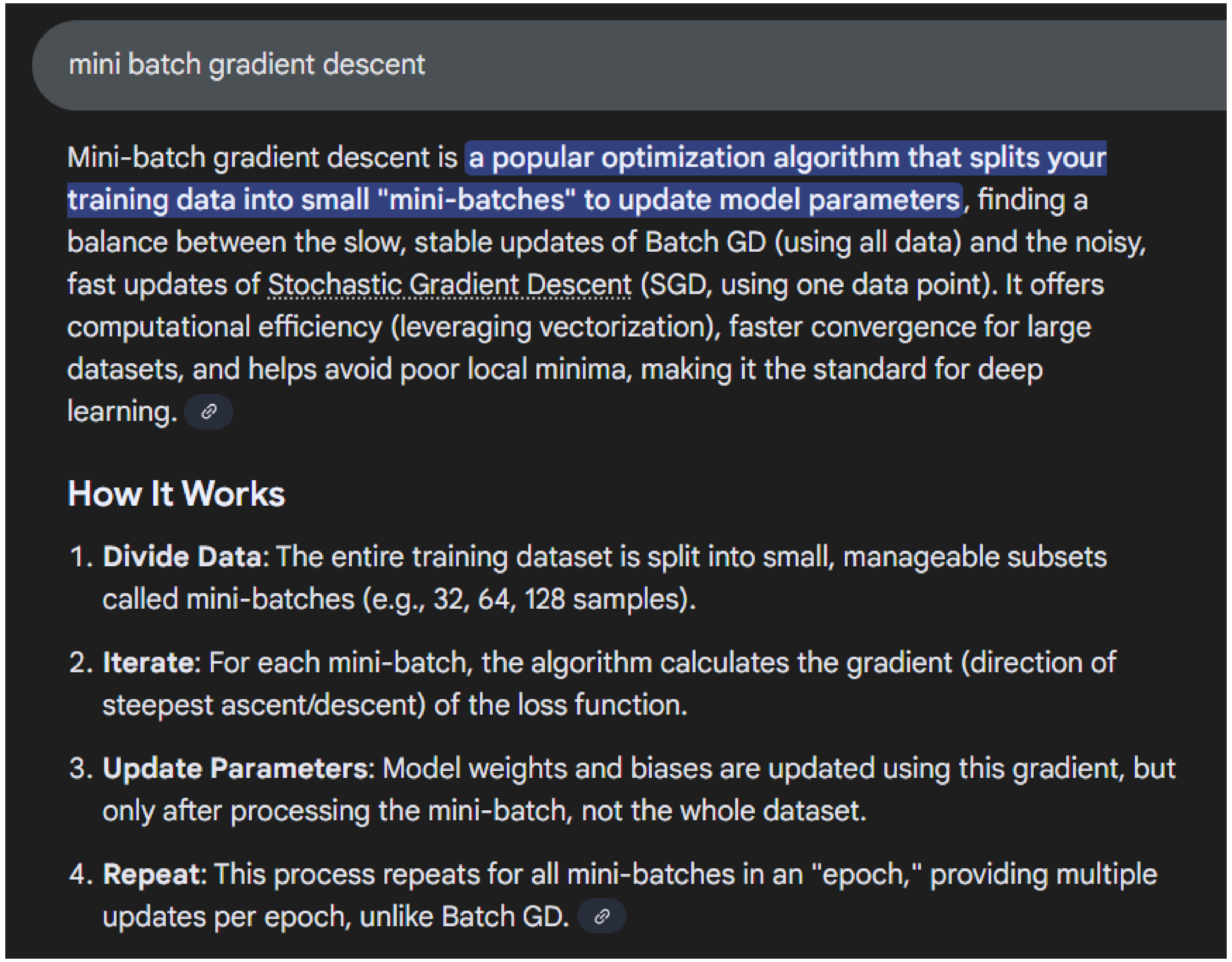
Disadvantages

- **Noisy Gradient Estimates**: Since the gradient is based on a single data point, the estimates can be noisy, leading to less accurate results.
- **Convergence Issues**: While SGD may converge quickly, it tends to oscillate around the minimum and does not settle exactly at the global minimum. This can be mitigated by gradually decreasing the learning rate.
- **Requires Shuffling**: To ensure randomness, the dataset should be shuffled before each epoch.



When to Use Stochastic Gradient Descent?

SGD is particularly useful when dealing with large datasets where processing the entire dataset at once is computationally expensive. It is also effective when optimizing non-convex loss functions.



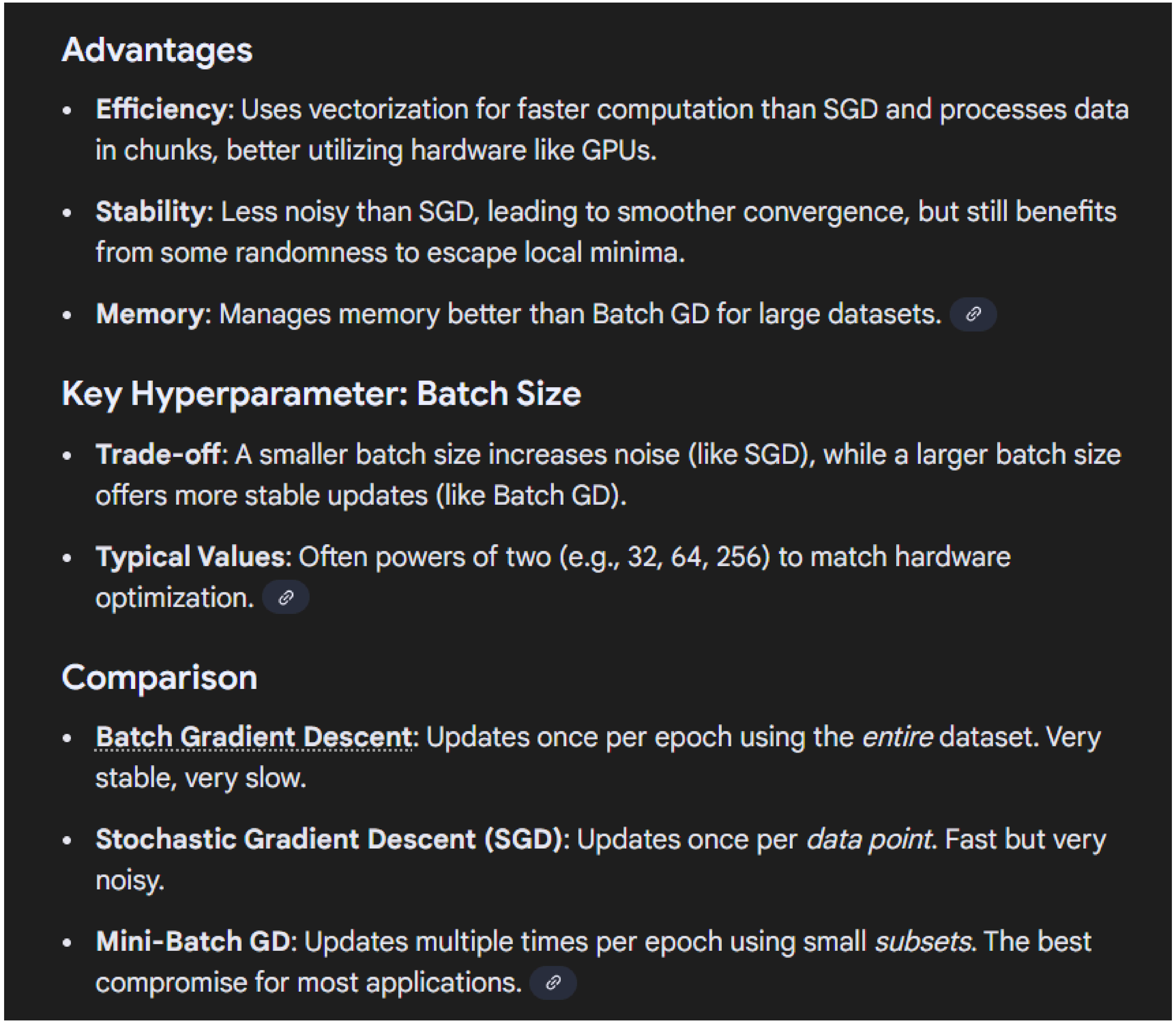
mini batch gradient descent

Mini-batch gradient descent is a popular optimization algorithm that splits your training data into small "mini-batches" to update model parameters, finding a balance between the slow, stable updates of Batch GD (using all data) and the noisy, fast updates of Stochastic Gradient Descent (SGD, using one data point). It offers computational efficiency (leveraging vectorization), faster convergence for large datasets, and helps avoid poor local minima, making it the standard for deep learning.

How It Works

1. **Divide Data**: The entire training dataset is split into small, manageable subsets called mini-batches (e.g., 32, 64, 128 samples).
2. **Iterate**: For each mini-batch, the algorithm calculates the gradient (direction of steepest ascent/descent) of the loss function.
3. **Update Parameters:** Model weights and biases are updated using this gradient, but only after processing the mini-batch, not the whole dataset.
4. **Repeat:** This process repeats for all mini-batches in an "epoch," providing multiple updates per epoch, unlike Batch GD.

We moslty use mini batch gradient descent



Advantages

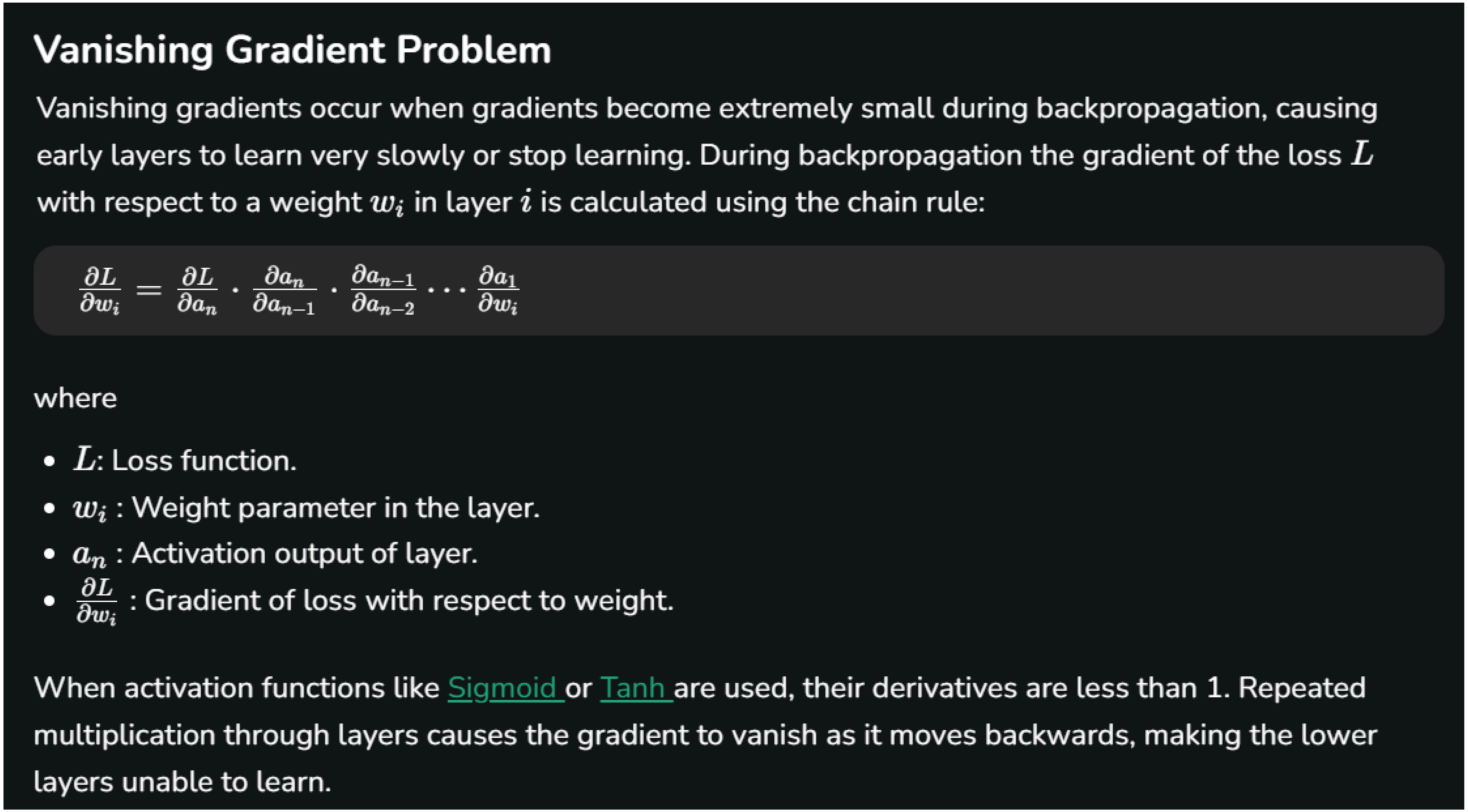
- **Efficiency:** Uses vectorization for faster computation than SGD and processes data in chunks, better utilizing hardware like GPUs.
- **Stability:** Less noisy than SGD, leading to smoother convergence, but still benefits from some randomness to escape local minima.
- **Memory**: Manages memory better than Batch GD for large datasets.

Key Hyperparameter: Batch Size

- **Trade-off**: A smaller batch size increases noise (like SGD), while a larger batch size offers more stable updates (like Batch GD).
- **Typical Values**: Often powers of two (e.g., 32, 64, 256) to match hardware optimization.

Comparison

- **Batch Gradient Descent**: Updates once per epoch using the *entire* dataset. Very stable, very slow.
- **Stochastic Gradient Descent (SGD)**: Updates once per *data point*. Fast but very noisy.
- **Mini-Batch GD**: Updates multiple times per epoch using small *subsets*. The best compromise for most applications.



Vanishing Gradient Problem

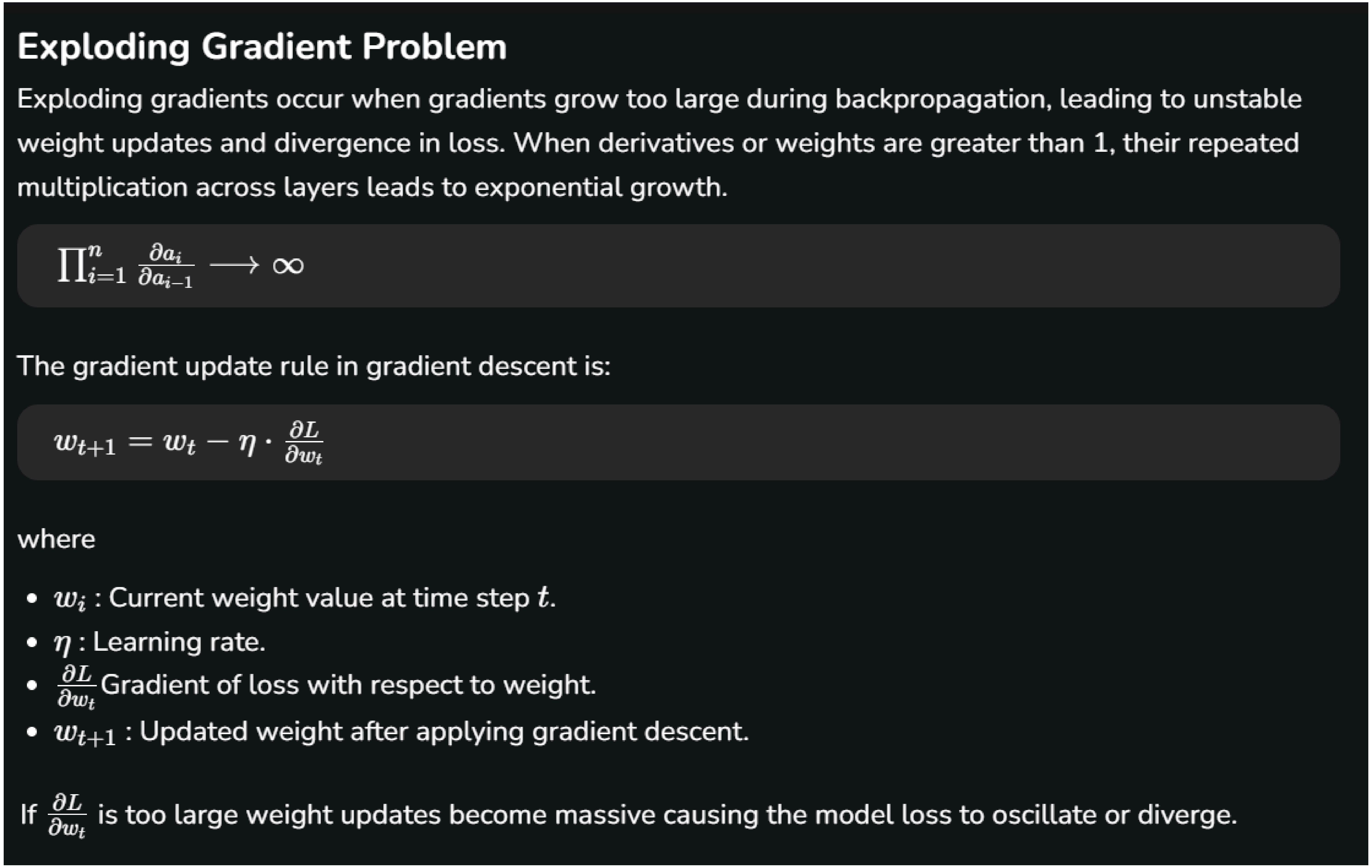
Vanishing gradients occur when gradients become extremely small during backpropagation, causing early layers to learn very slowly or stop learning. During backpropagation the gradient of the loss L with respect to a weight w_i in layer i is calculated using the chain rule:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial a_n} \cdot \frac{\partial a_n}{\partial a_{n-1}} \cdot \frac{\partial a_{n-1}}{\partial a_{n-2}} \cdots \frac{\partial a_1}{\partial w_i}$$

where

- L: Loss function.
- w_i : Weight parameter in the layer.
- a_n : Activation output of layer.
- $\frac{\partial L}{\partial w_i}$: Gradient of loss with respect to weight.

When activation functions like Sigmoid or Tanh are used, their derivatives are less than 1. Repeated multiplication through layers causes the gradient to vanish as it moves backwards, making the lower layers unable to learn.



Exploding Gradient Problem

Exploding gradients occur when gradients grow too large during backpropagation, leading to unstable weight updates and divergence in loss. When derivatives or weights are greater than 1, their repeated multiplication across layers leads to exponential growth.

$$\prod_{i=1}^{n} \frac{\partial a_i}{\partial a_{i-1}} \longrightarrow \infty$$

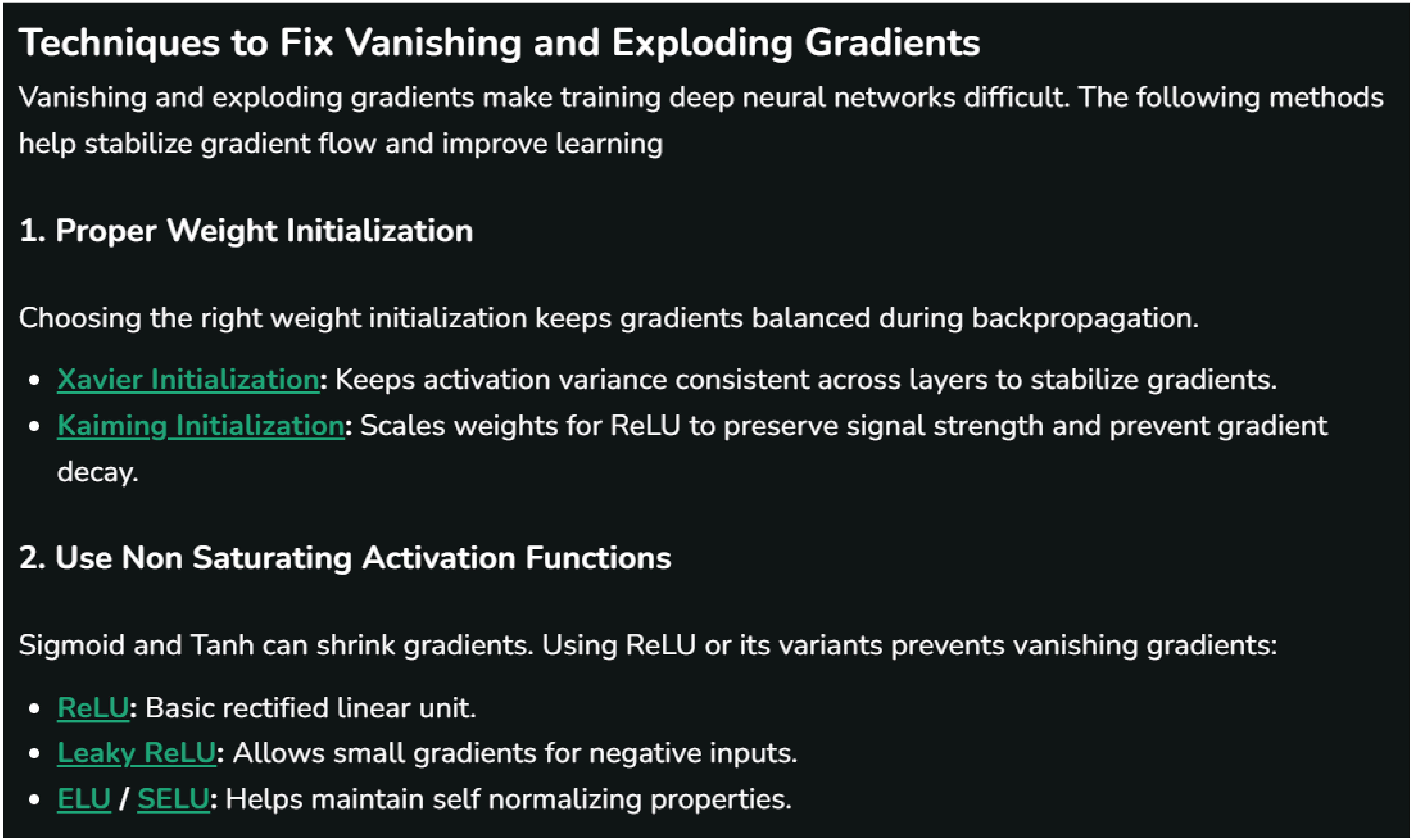
The gradient update rule in gradient descent is:

$$w_{t+1} = w_t - \eta \cdot \frac{\partial L}{\partial w_t}$$

where

- w_i : Current weight value at time step t.
- η : Learning rate.
- $\frac{\partial L}{\partial w_t}$ Gradient of loss with respect to weight.
- w_{t+1} : Updated weight after applying gradient descent.

If $\frac{\partial L}{\partial w_t}$ is too large weight updates become massive causing the model loss to oscillate or diverge.



Techniques to Fix Vanishing and Exploding Gradients

Vanishing and exploding gradients make training deep neural networks difficult. The following methods help stabilize gradient flow and improve learning

1. Proper Weight Initialization

Choosing the right weight initialization keeps gradients balanced during backpropagation.

- Xavier Initialization: Keeps activation variance consistent across layers to stabilize gradients.
- Kaiming Initialization: Scales weights for ReLU to preserve signal strength and prevent gradient decay.

2. Use Non Saturating Activation Functions

Sigmoid and Tanh can shrink gradients. Using ReLU or its variants prevents vanishing gradients:

- ReLU: Basic rectified linear unit.
- Leaky ReLU: Allows small gradients for negative inputs.
- ELU / SELU: Helps maintain self normalizing properties.

3. Apply Batch Normalization

Normalizes layer inputs to have zero mean and unit variance, stabilizing gradients and accelerating convergence.

4. Gradient Clipping

Limits gradients to a maximum threshold to prevent them from exploding and destabilizing training.