**1. Introduction to Optimization**

**What it is:**  
Optimization in machine learning means finding the **best values for the model’s parameters**—weights and biases—so that predictions are as accurate as possible. Think of it like tuning a musical instrument; the better you tune it, the closer the sound is to perfect.

**The main tool:** **Gradient Descent (GD)**

* GD is like rolling a ball down a hill, where the hill is the **loss function** (error).
* The goal: reach the **lowest point** (minimum error).
* The ball’s movement is guided by the **gradient**, which tells us the slope and direction to go down.

**Analogy:** Imagine you are blindfolded on a mountain. You feel the slope with your feet and step downhill. Step by step, you try to reach the valley. That’s Gradient Descent.

**2. Gradient Descent (Batch GD)**

**How it works:**

* Uses **all the training data** to compute the gradient once per epoch.
* Updates the weights only **after seeing the whole dataset**.

**Problems:**

1. **Slow:** Because it waits for the whole dataset before updating.
2. **Sensitive to learning rate:** Too large → overshoot; too small → slow progress.
3. **Analogy:** Like “snailing down a hill” because each step is very careful and slow.

**Example:** You have 50,000 images; Batch GD computes the gradient using all 50,000 before making one update.

**3. Stochastic Gradient Descent (SGD) & Mini-Batch GD**

**SGD:**

* Updates weights **after every single training example**.
* Pros: Faster, can escape small dips (local minima).
* Cons: Noisy updates → the path “zigzags” a lot.

**Mini-Batch GD:**

* Compromise between Batch and SGD.
* Split dataset into **small batches**, update after each batch.
* Example: 10,000 samples, batch size 1,000 → 10 updates per epoch.

**Why it’s better:**

* **Faster** than Batch GD.
* **Works well** with GPUs/parallel computing.
* Most people call Mini-Batch GD just “SGD” in practice.

**Summary:**

| **Type** | **Batch Size** | **Update Frequency** | **Speed** |
| --- | --- | --- | --- |
| Batch GD | all data | once per epoch | slow |
| Mini-Batch | 32–1024 | per batch | fast |
| True SGD | 1 | per example | noisy |

**4. Local vs Global Minimum**

**Problem:**

* Loss functions are not always smooth; they have **many dips**.
* **Local minimum:** small dip that is not the lowest.
* **Global minimum:** the deepest dip → best solution.

**Gradient Descent behavior:**

* Batch GD → often gets stuck in nearest local minimum.
* Mini-Batch GD → more random jumps, may escape small dips.

**Learning rate role:**

* **Too small:** stuck in local minima.
* **Too large:** may overshoot or oscillate.

**Analogy:** Think of a hilly landscape:

* Batch GD = careful walker → stops in first small valley.
* Mini-Batch GD = bouncy runner → can jump out of small valleys, maybe reach the deepest one.

**5. Momentum**

**Idea:**

* Momentum helps the ball **keep moving forward** instead of getting stuck in small dips.
* Uses **past updates** to influence current update.

**Formula:**

v = α\*v\_prev - η\*gradient

weight = weight + v

α = momentum coefficient (usually 0.9)

η = learning rate

Effect:

Faster convergence

Smoother path

Better chance to reach global minimum

Analogy:

A ball rolling down a hill keeps some “speed.” Small bumps (local minima) are ignored, and it continues toward the deepest valley.

6. Learning Rate (η)

Definition: Step size for weight updates.

Balance:

Too small: very slow learning.

Too large: unstable, may diverge.

Learning Rate Schedules: Adjust η over time.

Piecewise constant: start high (0.1), reduce stepwise (0.01 → 0.001).

Exponential decay: η decreases smoothly with epochs.

Benefit:

Train fast at the beginning.

Fine-tune at the end with smaller steps.

7. Visualizing Learning Rate

Low η: slow, takes long to reach minimum.

High η: oscillates or diverges.

Good η: fast, stable convergence.

Tip: Try multiple values, plot loss curve, choose the one with fast and smooth decline.

8. Adaptive Learning Rates

Automatically adjust learning rate for each parameter.

AdaGrad (2011):

Each weight has its own learning rate.

Decreases over time → works well for sparse data.

Problem: can shrink too much → stop training early.

RMSProp:

Fixes AdaGrad by using moving average of squared gradients.

Keeps learning rates from shrinking too much.

More stable and efficient.

**9. Adam Optimizer (2014)**

**Adam = Adaptive Moment Estimation**

* Combines **momentum + RMSProp**.
* Maintains **history of updates** (momentum) and **adaptive learning rates per weight** (RMSProp).

**Benefits:**

* Fast, stable, reliable.
* Works well in almost all deep learning tasks.
* Default in TensorFlow, PyTorch, Keras.

**Summary Table:**

| **Optimizer** | **Key Idea** | **Pros** |
| --- | --- | --- |
| Batch GD | Full dataset update | Simple, accurate |
| SGD | Single sample update | Faster, escapes local minima |
| Mini-Batch | Small batch update | Standard, fast |
| Momentum | Uses past gradients | Smoother, faster |
| AdaGrad | Adaptive per weight | Sparse data, early training |
| RMSProp | Fix AdaGrad | Stable, efficient |
| Adam | Momentum + RMSProp | Fast, reliable, default |

