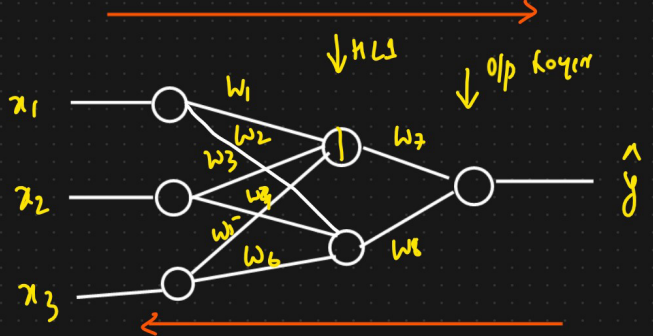
# Optimizers ⟶ used for reducing cost function or reach Global minima.

1. Gradient Descent
2. Stochastic Gradient Descent (SGD)
3. Mini batch SGD
4. SGD With Momentum
5. Adagrad and RMSPROP
6. Adam Optimizers

Epoch → completed 1 iteration i.e.: 1 Forward P. 1 Backward P.

Best of both worlds but bit noisy

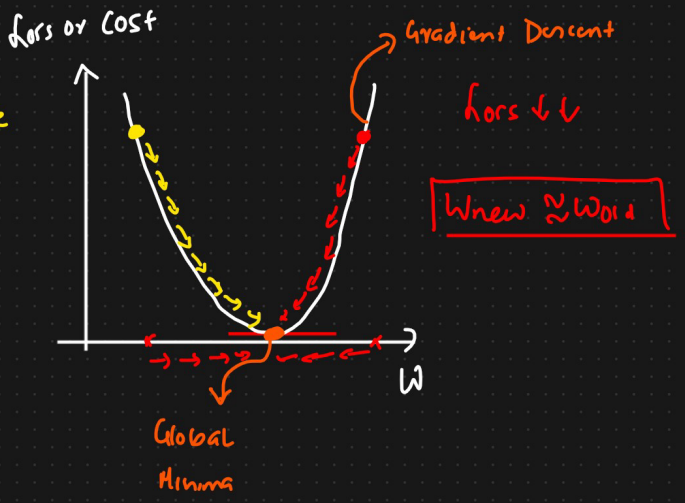
## ① Gradient Descent Optimizer




Backward Propogation

Loss = [ ] ↓↓↓

Optimizers ↑




Learning Rate

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

MSE

$$\text{Loss fn} = (y - \hat{y})^2 \qquad \text{Cost fn} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Dataset = 1000 Data points

## Epochs, Iteration

1 Epoch = 1 Iteration

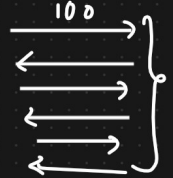
10 Iteration

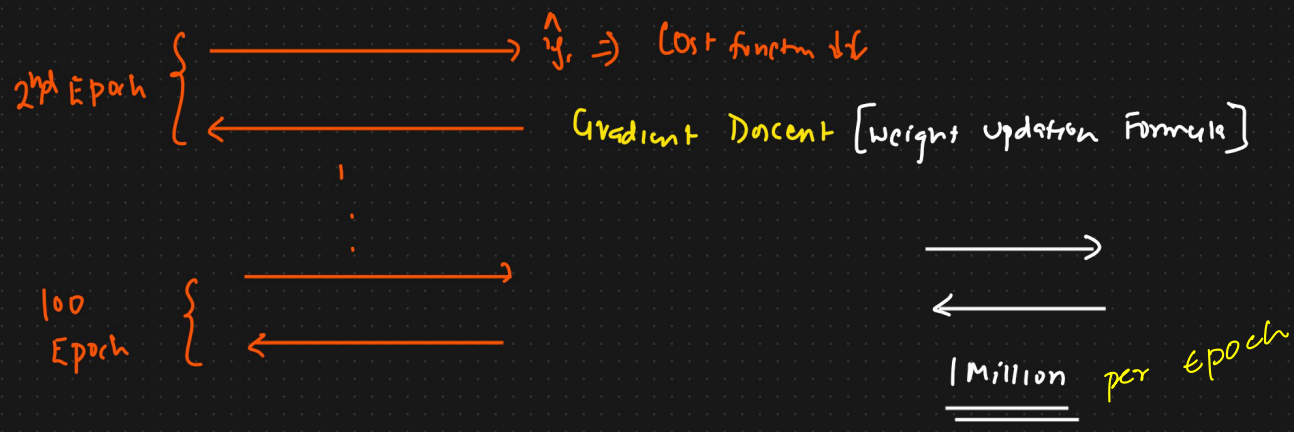
100/10 = 100

1 Epoch { 1000 datapoint ⟶ $\hat{y}_i$ ⇒ Cost function ↓↓

Weights will get updated ⟵ }

Gradient Descent optimizer [weight updation]






## Advantages

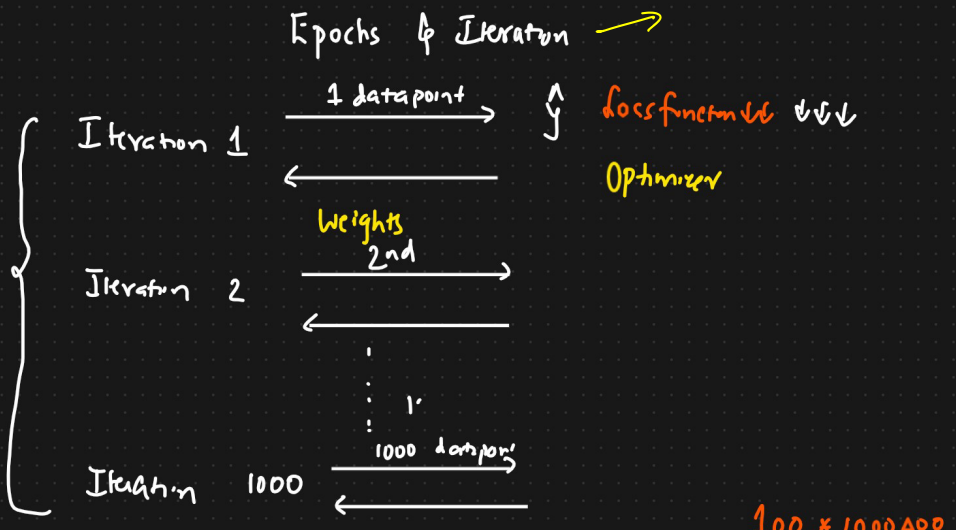
① Convergance will happen.
- Accurate Gradients
- Good for small Datasets

## Disadvantage

① Huge Resource RAM, GPU
⇓
Resource Intensive.

## ② Stochastic Gradient Descent (SGD)

1000 datapoint



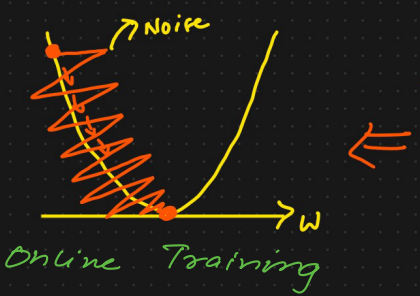

100 × 1000 = 100000

Epochs

1000000 datapoint

## Advantage

Ⓐ Solve Resource Issue




## Disadvantage


Ⓑ Time Complexity ↑↑

Ⓒ Convergence will also take more time.

Ⓓ Noise gets Introduced

→ may skip Global minima.

# ③ Mini Batch SGD

Batch sizes

No. of iterations = $\frac{100000}{1000}$ = 100 iterations

Epoch, Iteration, Batch_size

Datapoints = 100000

batch_size = 1000

MSE

Cost fn = $\sum_{i=1}^{1000} (y_i - \hat{y}_i)^2$ ↓↓

Epoch 1

Iteration 1 → 1000 →

← Optimizer ⇒ Mini Batch SGD

Change the weights

Iteration 2 → 1000 →

←

Iteration 3 →

←

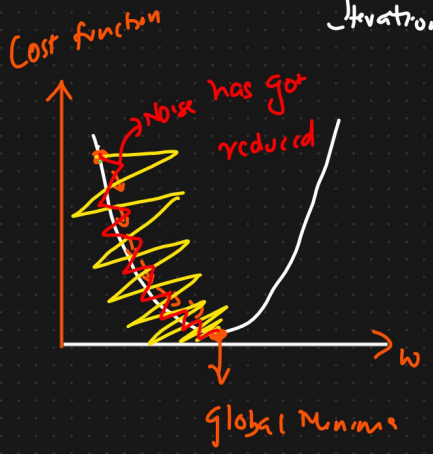
⋮

Iteration 100 →

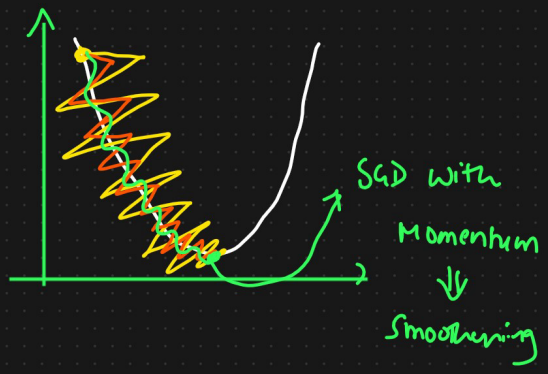
←

↓↓

| | |
|---|---|
| 8gb | 1 |
| 16gb | 5000 |




Batch Size




## Advantages

① Convergence speed will increase =

② Noise will be less when compared to SGD

③ Efficient Resource Usage (RAM)

## Disadvantage

① Noise still exists

In every epoch 1000 iterations of 1000 DP

# Conclusion

Optimizers = Reduce cost function

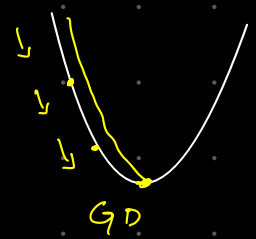
## Working

### 🔪 1. Batch Gradient Descent (GD)

You taste **all 100 cakes at once**, take notes, average out all your feedback, and then **make one big adjustment** to the recipe.

- 🎯 One big decision after thorough review
- ⏳ Takes time (you must finish all cakes)
- 📉 Very precise, but slow

📌 **1 update per round (epoch)**
📌 Uses all 100 cakes each time




### ☕ 2. Stochastic Gradient Descent (SGD)

You're in a hurry — you **taste one cake**, make a quick adjustment. Then you taste the next cake, make another tweak.

- 🎲 Fast but very chaotic
- 👎 Some tweaks may help, others might hurt
- 🎯 But over time, you may stumble upon the perfect recipe

📌 **100 updates per round (epoch)**
📌 Uses 1 cake at a time, very noisy

### 🍽️ 3. Mini-Batch Gradient Descent (batch size = 10)

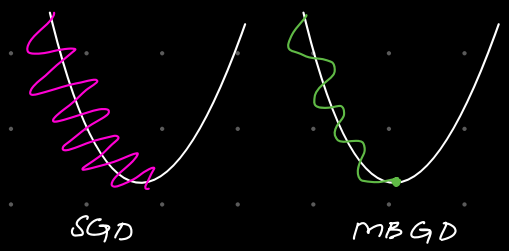
You group the 100 cakes into **10 plates** (batches) of 10 cakes each. You taste a plate, make a small adjustment, and repeat.

- ⏱️ Balanced speed
- ✅ Not too noisy, not too slow
- 🎯 Gets better with practice

📌 **10 updates per round (epoch)**
📌 Each update uses 10 cakes

### 🔁 Summary (Cake Edition):

| Method | How You Taste | How You Adjust | Feeling |
|---|---|---|---|
| GD | Taste all 100 | One big tweak | Slow but sure |
| SGD | Taste 1 cake | Constant tweaking | Fast but shaky |
| Mini-Batch | Taste 10 at a time | Small steady tweaks | Best of both worlds |



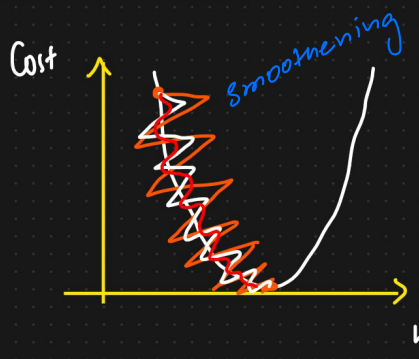

## Explanation:

### ✅ Gradient Descent Variants Explained

| Type | How it Works | Speed | Stability | Used When | Pros | Cons |
|---|---|---|---|---|---|---|
| **Batch Gradient Descent** | Uses the **entire dataset** to compute gradient before updating weights. | ❌ Slow (per update) | ✅ Very stable | Small datasets | + Accurate gradients<br>+ Converges smoothly | – Very slow on large data<br>– High memory usage |
| **Stochastic GD (SGD)** | Uses **1 data point** at a time to update weights. | ✅ Fast (per update) | ❌ Very noisy | Large datasets, online learning | + Fast updates<br>+ Good for huge/streaming data<br>+ Can escape local minima | – Noisy convergence<br>– May overshoot or bounce around |
| **Mini-Batch SGD** | Uses a **small subset** of data (e.g., 32, 64) for each update. | ⚖️ Balanced | ✅ Smooth + efficient | Most common in deep learning | + Fast and stable<br>+ Efficient on GPUs<br>+ Generalization-friendly | – Needs tuning of batch size<br>– Still a bit noisy |

we need Momentum based SGD to smoothen noise (exponential weighted average)

## ④ SGD With Momentum




$$W_{new} = W_{old} - \eta \frac{\partial h}{\partial W_{old}}$$ (learning rate)

$$b_{new} = b_{old} - \eta \frac{\partial h}{\partial b_{old}}$$

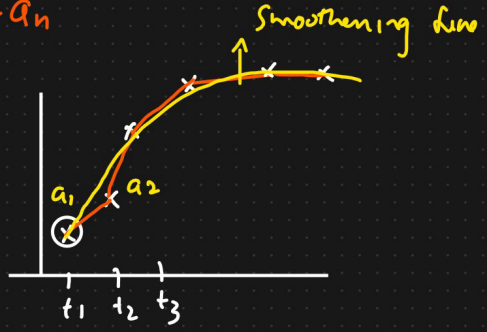
$$W_t = W_{t-1} - \eta \frac{\partial h}{\partial W_{t-1}}$$

Weight Updation formula

Exponential Weight Average {Smoothening} ⇒ ARIMA, SARIMAX

| Time = | $t_1$ | $t_2$ | $t_3$ | $t_4$ | - - - | $t_n$ |
|---|---|---|---|---|---|---|
| Values = | $a_1$ | $a_2$ | $a_3$ | $a_4$ | - - - - | $a_n$ |

Time Series




$$V_{t_1} = a_1$$

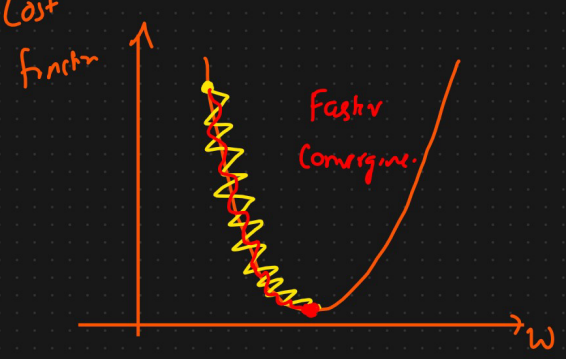
$$V_{t_2} = \beta * V_{t_1} + (1-\beta) * a_2$$

$\beta = 0$ to $1$

$$= 0.95 * a_1 + (0.05)\, a_2$$

$$V_{t_3} = \beta * V_{t_2} + (1-\beta) * a_3$$

$$= 0.95 \left[ 0.95 * a_1 + (0.05)\, a_2 \right] + (0.05) * a_3$$




Advantage

① Reduces the noise

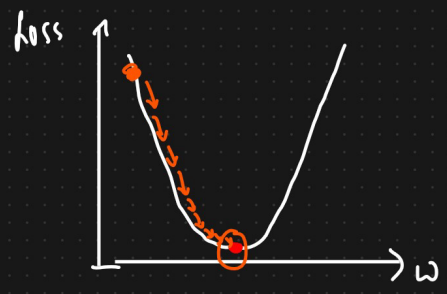
② Quick Convergence

## ⑤ Adagrad : Adaptive Gradient Descent

$\eta$ = fixed ⇒ Dynamic learning Rate

$$w_t = w_{t-1} - \eta \frac{\partial h}{\partial w_{t-1}}$$

→ Learning Rate = 0.001




As the convergence happen the learning rate should change

$0.00001 \approx 0$

$$w_t = w_{t-1} - \eta' \frac{\partial h}{\partial w_{t-1}}$$

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

$\eta' \downarrow\downarrow$ $\alpha_t \uparrow\uparrow$

Epsilon ≈ small value

$\alpha_t \uparrow\uparrow$

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial h}{\partial w_t}\right)^2$$

$w_t \approx w_{t-1}$

$t=1 \rightarrow t=2 \rightarrow t=3$

$\eta = 0.01$ $\eta = 0.005$ $\eta = 0.003$

### Disadvantage

① $\eta'$ → Possibility to become a very small value ≈ 0

## ⑥ Adadelta And RMSPROP

Exponential Weight Average

$\beta = 0.95$ $\quad S_{dw_t} = 0$

$$S_{dw_t} = \beta * S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial h}{\partial w_{t-1}}\right)^2$$

$$\eta' = \frac{\eta}{\sqrt{S_{dw_t} + \epsilon}}$$

[Dynamic LR + Smoothening EWA]

$$w_t = w_{t-1} - \eta' \frac{\partial h}{\partial w_{t-1}}$$

## ⑦ Adam Optimizer

SGD with Momentum + RMSPROP [Dynamic LR + Smoothening]

$$w_t = w_{t-1} - \eta' V_{dw}$$

Weight Updation

$$b_t = b_{t-1} - \eta' V_{db}$$

Bias Updation

$$\eta' = \frac{\eta}{\sqrt{S_{dw_t} + \epsilon}}$$

$S_{dw_t} = 0$

EWA

$$S_{dw_t} = \beta * S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial h}{\partial w_{t-1}}\right)^2$$

[EWA]

$$V_{dw_t} = \beta * V_{dw_{t-1}} + (1-\beta)\frac{\partial h}{\partial w_{t-1}}$$

$$V_{db_t} = \beta * V_{db_{t-1}} + (1-\beta)\frac{\partial h}{\partial b_{t-1}}$$

⇒ Momentum
↳ Smoothening

# Final Conclusion

## ✅ Gradient Descent Variants & Optimizers — Full Comparison

| Type | How it Works | Speed | Stability | Used When | Pros | Cons |
|---|---|---|---|---|---|---|
| **Batch Gradient Descent** | Uses the **entire dataset** to compute gradient before each weight update. | ❌ Slow (per update) | ✅ Very stable | Small datasets | + Accurate gradients<br>+ Smooth convergence | – Very slow on large data<br>– High memory usage |
| **Stochastic GD (SGD)** | Uses **1 data point** at a time to compute and apply updates. | ✅ Fast (per update) | ❌ Very noisy | Large datasets, online learning | + Very fast updates<br>+ Good for real-time/streaming<br>+ Can escape local minima | – Noisy updates<br>– May bounce or overshoot |
| **Mini-Batch SGD** | Uses a **small batch** (e.g., 32–128 samples) per update. | ⚖️ Balanced | ✅ Smooth + efficient | Deep learning, GPU training | + Combines speed & stability<br>+ Efficient on hardware<br>+ Improves generalization | – Needs batch size tuning<br>– Some noise still persists |
| **SGD with Momentum** | Like Mini-Batch SGD, but adds **momentum**: a fraction of past updates to current updates — helps **smooth curves** and accelerate convergence. | ✅ Faster than SGD | ✅ More stable | Deep learning, non-convex optimization | + Speeds up convergence<br>+ Reduces oscillation<br>+ Helps escape flat regions (plateaus) | – Needs momentum tuning<br>– Risk of overshooting |
| **AdaGrad** | Adapts learning rate **per parameter** based on past squared gradients — good for sparse data. | ⚡ Fast initially | ⚖️ Decent | Sparse features, NLP tasks | + No LR tuning needed<br>+ Great for sparse data like text (NLP) | – Learning rate decays too fast<br>– Stops learning early |
| **Adam Optimizer** | Combines **momentum + adaptive learning rate** per parameter using moving averages of gradients. | ✅ Fast & efficient | ✅ Very stable | Deep learning (default choice) | + Fast convergence<br>+ Works well out-of-the-box<br>+ Handles sparse & noisy gradients | – May overfit<br>– Sensitive to hyperparameters<br>– May not generalize best |

## 🧠 TL;DR:

| Situation | Best Choice |
|---|---|
| Small, simple dataset | **Batch GD** |
| Real-time or massive streaming data | **SGD** |
| General deep learning tasks | **Mini-Batch SGD / Adam** |
| Complex loss surfaces, faster training | **SGD with Momentum** |
| NLP or sparse data | **AdaGrad / Adam** |

Online Train

Less Noise

Imp. Adam optimizer = Best optimizer (Moving avg of gradients).
Combines Adaptive Learning Rate + Momentum