① every optimizer is a Research Paper

② Lot of optimizers made by PhD students in their 5 yrs of PhD

③ Going into Deep Math of optimizers won't be useful to us.

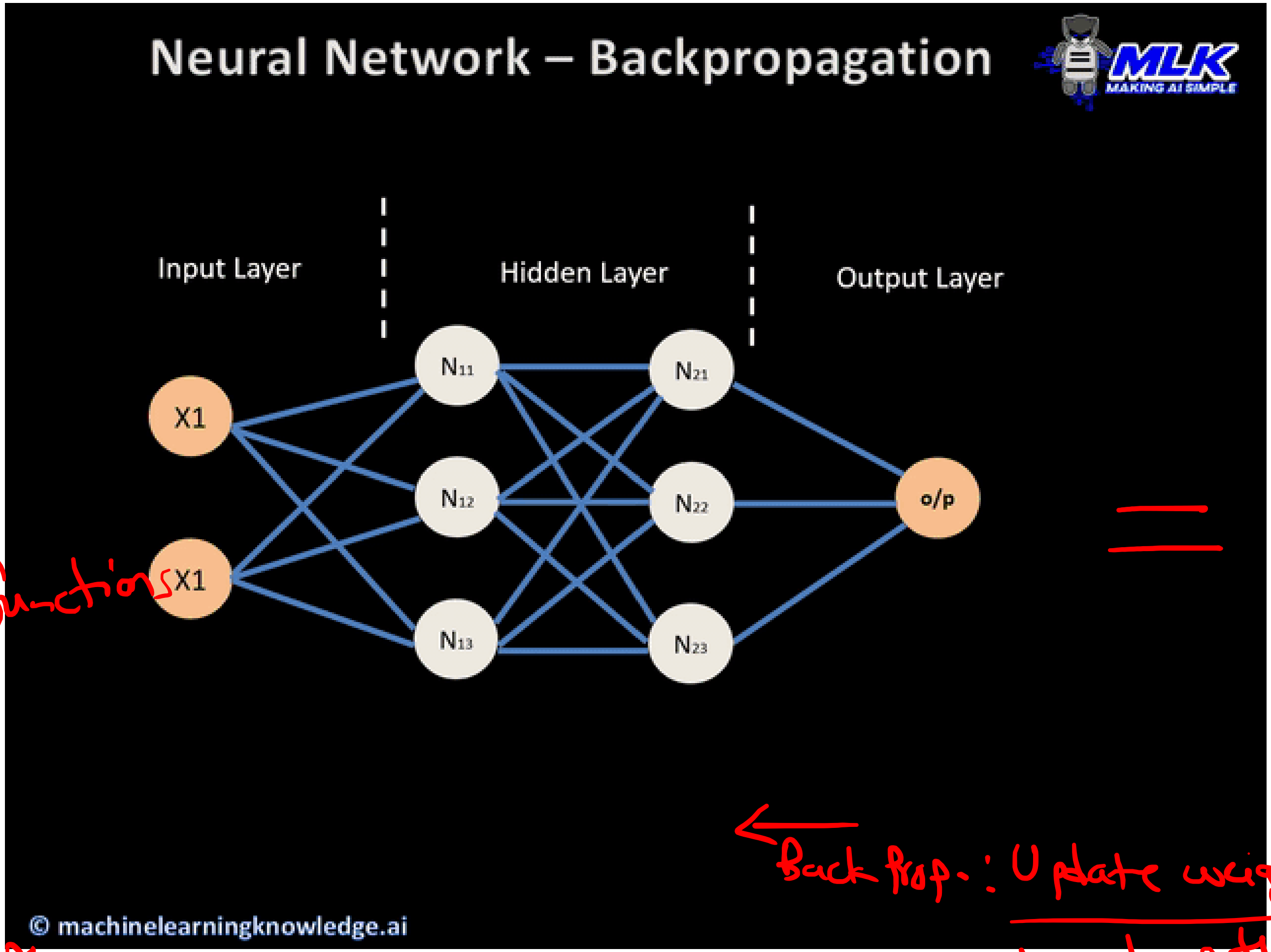
④ what is useful is to understand its overall working & to some extent its Mathematics?

# Optimizers

⑤ Optimizer is best friend of Loss functions

Opt. tk help of Loss func. to make a problem converge faster

↳ get a solution faster




← Back Prop.: Update weights.

who does that?

& on what basis? OPTIMIZERS

# Optimizers



- **What is Optimizer ?**

change | update

- It is very important to tweak the weights of the model during the training process, to make our predictions as correct and optimized as possible. But how exactly do you do that?
- How do you change the parameters of your model, by how much, and when?

Optimizer. → they tie together loss function & model parameters
↳ shape & mould your model into the most accurate form by futzing the weights.

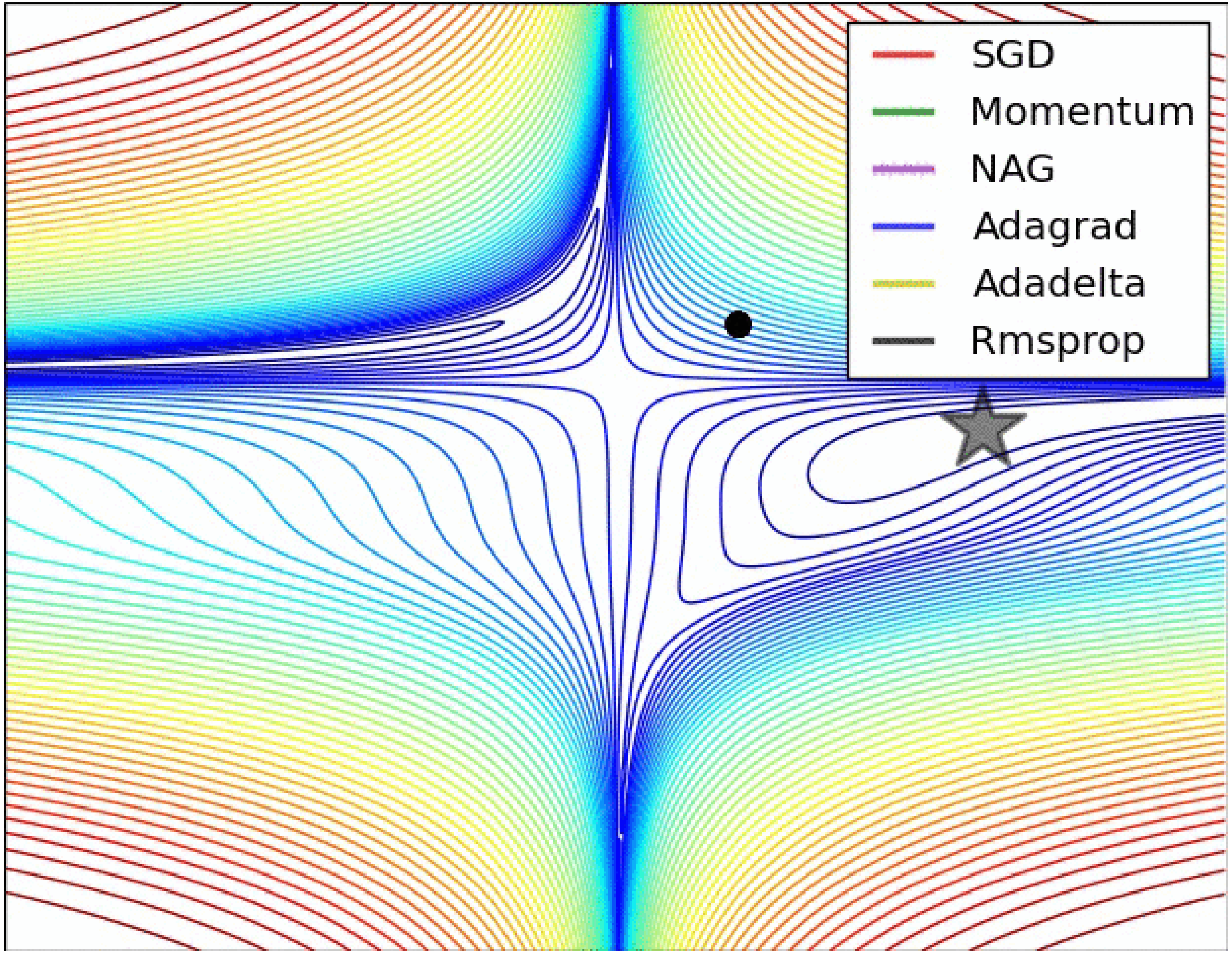
# Optimizers

- Below are list of example optimizers

- Adagrad
- Adadelta
- Adam
- Conjugate Gradients
- BFGS
- Momentum
- Nesterov Momentum
- Newton’s Method
- RMSProp
- SGD

# Optimizers

- Picking the right optimizer with the right parameters, can help you squeeze the last bit of accuracy out of your neural network model.




experimentation:

Look at the latest optimizer in the research paper, usually that performs the best.

# Adagrad Optimizer

- Adagrad (short for adaptive gradient) adaptively sets the learning rate ($\eta$) according to a parameter.

⭑ Divides the learning rate by sum of squares (SS) of all previous gradients.

→ When the SS past gradients has a high value, Adagrad divides the L.R. by a high value, ∴ the L.R. will become less.

→ $|||^{ly}$, when the SS past gradients has a low value, Adagrad divides the L.R. by a low value, ∴ the L.R. will become high.

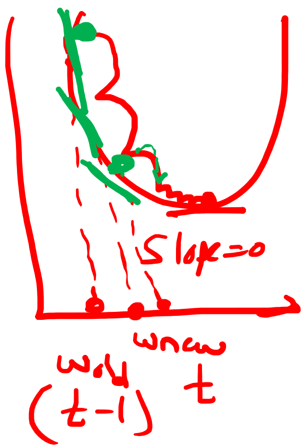
Concluding:

$$\boxed{L.R. \propto \frac{1}{\text{SS of all previous gradients of the parameter}}}$$

Gradient Descent, Minibatch GD, Stochastic GD $\Rightarrow$ In all of these optimizers, L.R is FIXED

# Adagrad Optimizer

$\eta = 0.001$




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

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

Adagrad:

$$w_t = w_{t-1} - \eta'_t \cdot \frac{\partial L}{\partial w_{t-1}}$$

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

$\epsilon$ ↳ a very small +ve value

$$\boxed{\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_i}\right)^2}$$

t=3
t=2
t=1

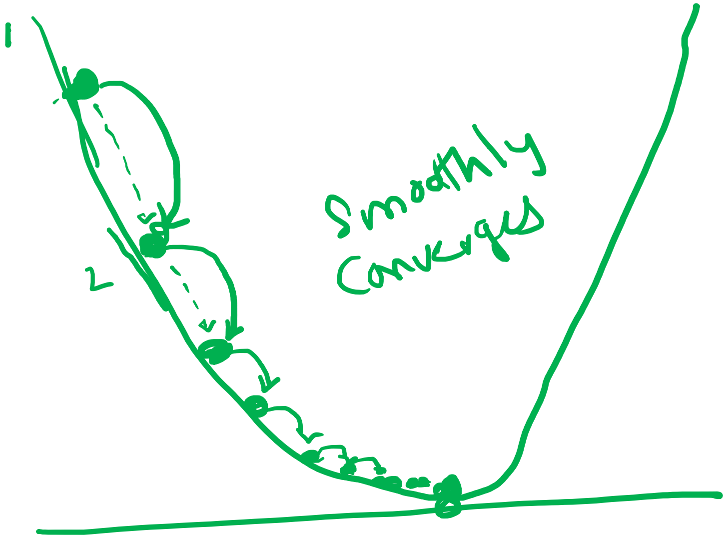
Conclusion: If $\alpha_t$ is a very high no., $\eta'_t$ is v. small

If $\alpha_t$ is a very small no, $\eta'_t$ is large.

If $\alpha_t \to 0$, $\epsilon$ will help us by ensuring the denominator doesn't become zero.

# Adagrad Optimizer

$\alpha_t \downarrow \; \eta'_t \uparrow$




$LR = \cancel{0.01}$

$\cancel{0.008}$

$0.00016$

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

$$\boxed{\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_i}\right)^2}$$

as we move down the slope, $\alpha_t \uparrow$

$SS \propto Slope$

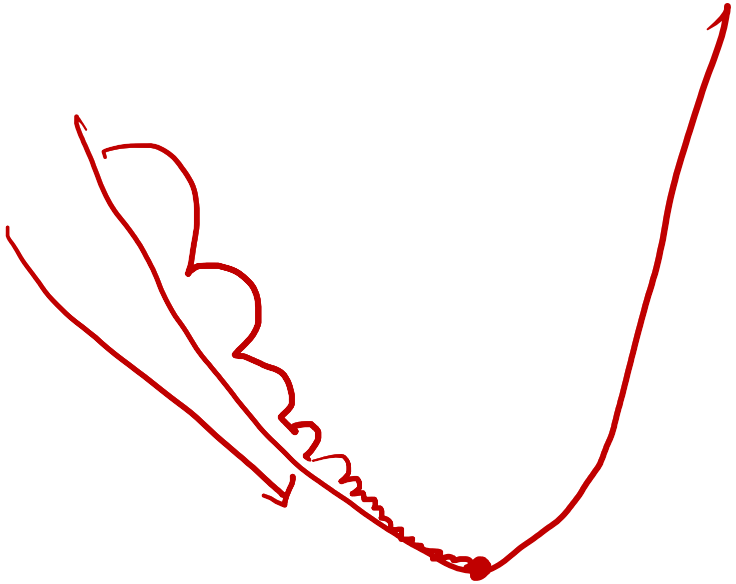
$SS = L.R. \times Slope$

$\text{Step size} = \eta \cdot \frac{\partial L}{\partial w}$

$$w_t = w_{t-1} - \eta'_t \cdot \frac{\partial L}{\partial w_{t-1}}$$

# Adagrad Optimizer Disadvantage

How to fix this?
RMS Prop
& Adadelta



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

$$\boxed{\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_i}\right)^2}$$

$\alpha \uparrow$

Sometimes, $\alpha_t$ becomes a v.v. high no., $\therefore$ $\eta'_t$ becomes v.v.v. small

$$w_t = w_{t-1} - \underbrace{\eta'_t \cdot \frac{\partial L}{\partial w_{t-1}}}_{\text{v.v.v.small}}$$

$\therefore$ $W_{old} \approx W_{new}$.
i.e. weights do not update.
& Training takes v.v.v.v. long Time.

# RMSProp and Adadelta

(Both are almost same. Two different research teams developed it.)

How do the help overcome disadvantage of Adagrad?

Soln: They try to control the $\alpha_t$.

We know that when $\alpha_t \uparrow\uparrow\uparrow$, L.R. $\downarrow\downarrow$.

But we don't want it to decrease to a very small number.

# RMSProp and Adadelta

$w_{Avg}$ = Weighted Average

$$\boxed{\gamma = 0.95}$$

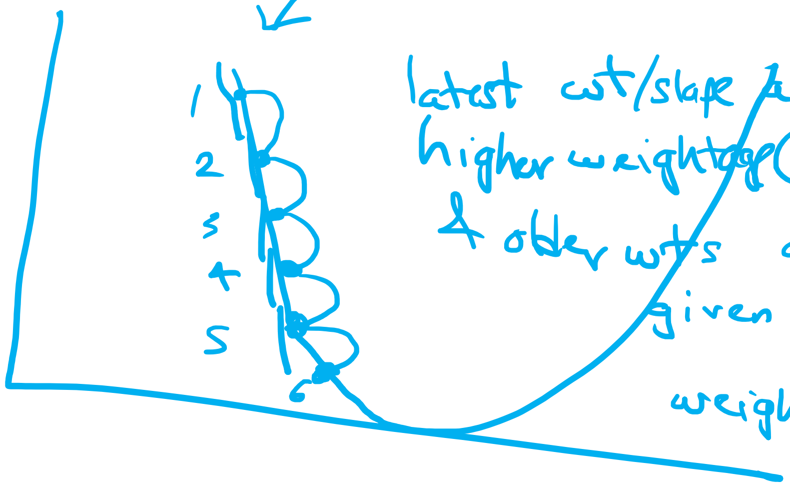
**Adagrad:** $w_{new\,(t)} = w_{old\,(t-1)} - \eta_t' \cdot \dfrac{\partial L}{\partial w_{old\,(t-1)}}$

$$\eta_t' = \frac{\eta}{\sqrt{w_{Avg} + \epsilon}}$$

↓ slowly & steadily

$\alpha$ ↑↑, now it won't explode

$$w_{Avg} = \underset{0.95}{\gamma} \cdot \underbrace{w_{Avg\,(t-1)}}_{6} + \underset{0.05}{(1-\gamma)} \cdot \underbrace{\left(\frac{\partial L}{\partial w_t}\right)^2}_{1,2,3,4,5}$$




latest wt/slope are given higher weightage (0.95) & older wts are given lesser weightage (0.05)

# RMSProp Optimizer

- Another adaptive learning rate optimization algorithm, Root Mean Square Prop (RMSProp) works by keeping an exponentially weighted average of the squares of past gradients. RMSProp then divides the learning rate by this average to speed up convergence.

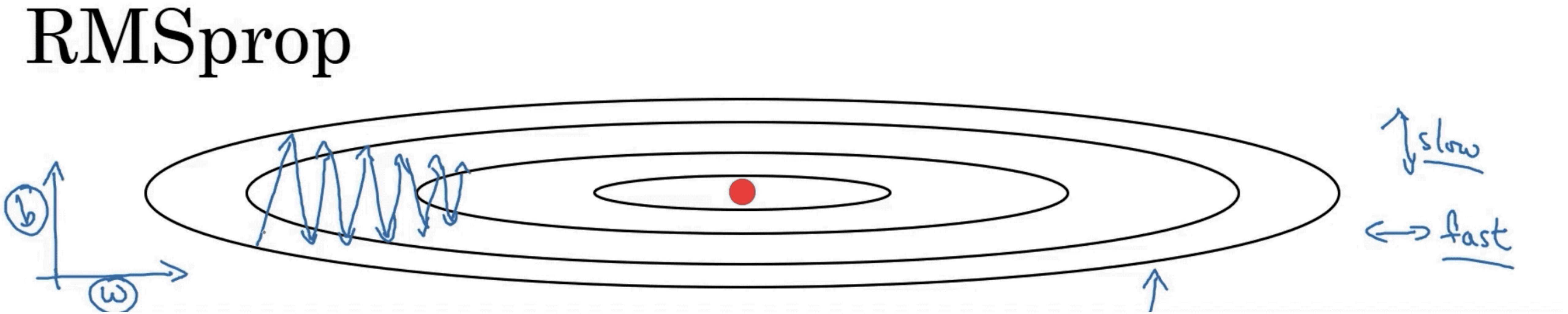
$$s_{dW} = \beta s_{dW} + (1-\beta)\left(\frac{\partial \mathcal{J}}{\partial W}\right)^2$$

$$W = W - \alpha \frac{\frac{\partial \mathcal{J}}{\partial W}}{\sqrt{s_{dW}^{corrected}} + \varepsilon}$$

**Note**

- $s$ - the exponentially weighted average of past squares of gradients
- $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer weight tensor
- $W$ - weight tensor
- $\beta$ - hyperparameter to be tuned
- $\alpha$ - the learning rate
- $\epsilon$ - very small value to avoid dividing by zero

# RMSProp Optimizer



# Adagrad Optimizer

$$g_t^i = \frac{\partial \mathcal{J}(w_t^i)}{\partial W}$$

$$W = W - \alpha \frac{\partial \mathcal{J}(w_t^i)}{\sqrt{\sum_{r=1}^{t} \left(g_r^i\right)^2 + \varepsilon}}$$

- Note

$g_t^i$ - the gradient of a parameter, :math: `Theta ` at an iteration t.
$\alpha$ - the learning rate
$\epsilon$ - very small value to avoid dividing by zero

# Adagrad Optimizer

```
def Adagrad(data):
    gradient_sums = np.zeros(theta.shape[0])
    for t in range(num_iterations):
        gradients = compute_gradients(data, weights)
        gradient_sums += gradients ** 2
        gradient_update = gradients / (np.sqrt(gradient_sums + epsilon))
        weights = weights - lr * gradient_update
    return weights
```

# Adadelta Optimizer

- Adadelta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address two drawbacks:
  - The continual decay of learning rates throughout training
  - The need for a manually selected global learning rate
- Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients.
- This way, Adadelta continues learning even when many updates have been done.
- Compared to Adagrad, in the original version of Adadelta you don't have to set an initial learning rate. In this version, initial learning rate can be set, as in most other Keras optimizers.

# Adadelta Optimizer

- AdaDelta belongs to the family of stochastic gradient descent algorithms, that provide adaptive techniques for hyperparameter tuning. Adadelta is probably short for ‘adaptive delta’, where delta here refers to the difference between the current weight and the newly updated weight.
- The main disadvantage in Adagrad is its accumulation of the squared gradients. During the training process, the accumulated sum keeps growing. As the accumulated sum increases, learning rate starts to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

# Adadelta Optimizer

- Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done.
- With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.
- Implementation is something like this,

$$v_t = \rho v_{t-1} + (1-\rho)\nabla_\theta^2 J(\theta)$$

$$\Delta\theta = \frac{\sqrt{w_t + \epsilon}}{\sqrt{v_t + \epsilon}}\nabla_\theta J(\theta)$$

$$\theta = \theta - \eta\Delta\theta$$

$$w_t = \rho w_{t-1} + (1-\rho)\Delta\theta^2$$

# Adadelta Optimizer

```
def Adadelta(weights, sqrs, deltas, rho, batch_size):
    eps_stable = 1e-5
    for weight, sqr, delta in zip(weights, sqrs, deltas):
        g = weight.grad / batch_size
        sqr[:] = rho * sqr + (1. - rho) * nd.square(g)
        cur_delta = nd.sqrt(delta + eps_stable) / nd.sqrt(sqr + eps_stable) * g
        delta[:] = rho * delta + (1. - rho) * cur_delta * cur_delta
        # update weight in place.
        weight[:] -= cur_delta
```

# Stochastic Gradient Descent

Compare only one point / row in the dataset.

& calculate the cost for each step.

# Stochastic Gradient Descent

# Stochastic Gradient Descent

**Pros** ① relatively fast as compared to older gradient desc. approaches.

② easy to learn for beginners (∵ math is not heavy).

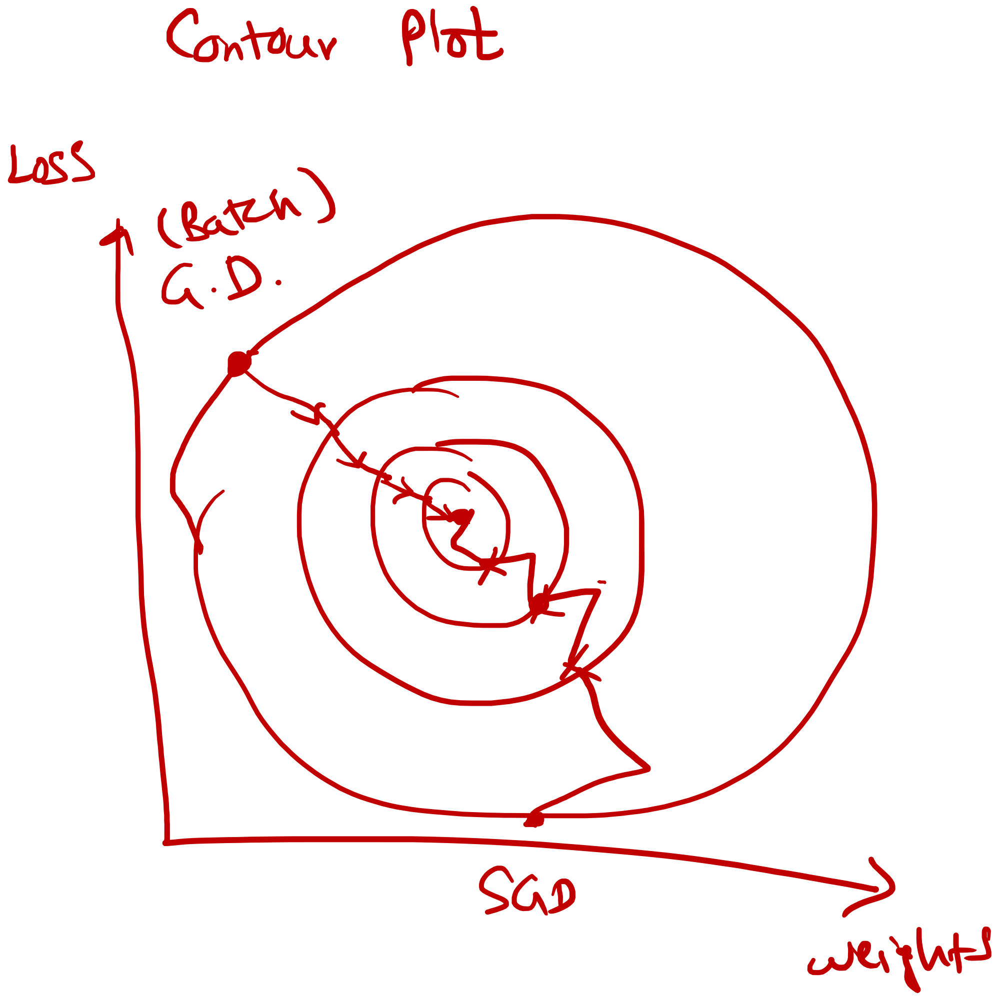
**Cons**

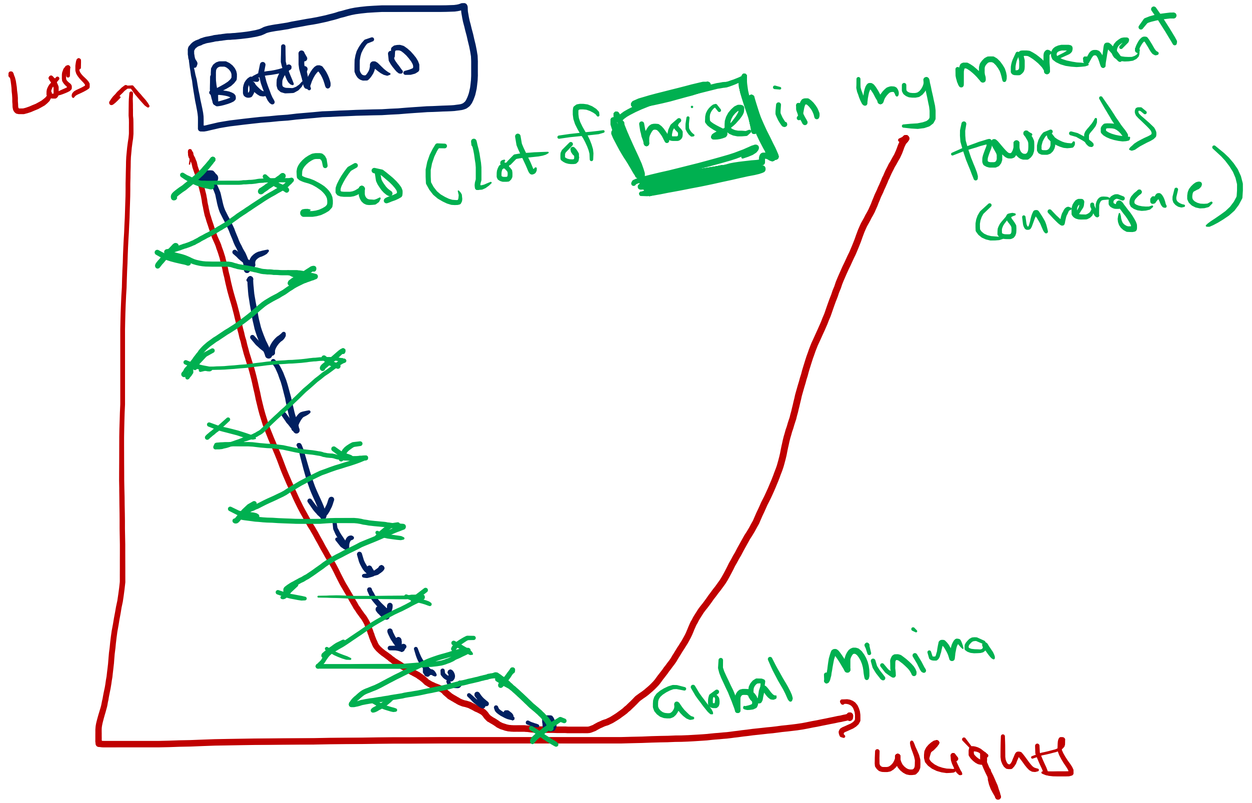
① Converges slowly than newer optimizers | algorithms.

② Can get stuck in local minima

③ Newer approaches outperform SGD in terms of optimizing the cost function.

# Stochastic Gradient Descent with Momentum



# Stochastic Gradient Descent with Momentum

Data point/rows:

| Time | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ .... $t_n$ |
|---|---|---|---|---|---|
| Data point | $b_1$ | $b_2$ | $b_3$ | $b_4$ | $b_5$ ---- $b_n$ |

$0 \le \gamma \le 1$

At time instant $t=1$,

Declare variable $\boxed{V_{t_1} = b_1}$ — (a) $\boxed{\gamma = 0.5}$

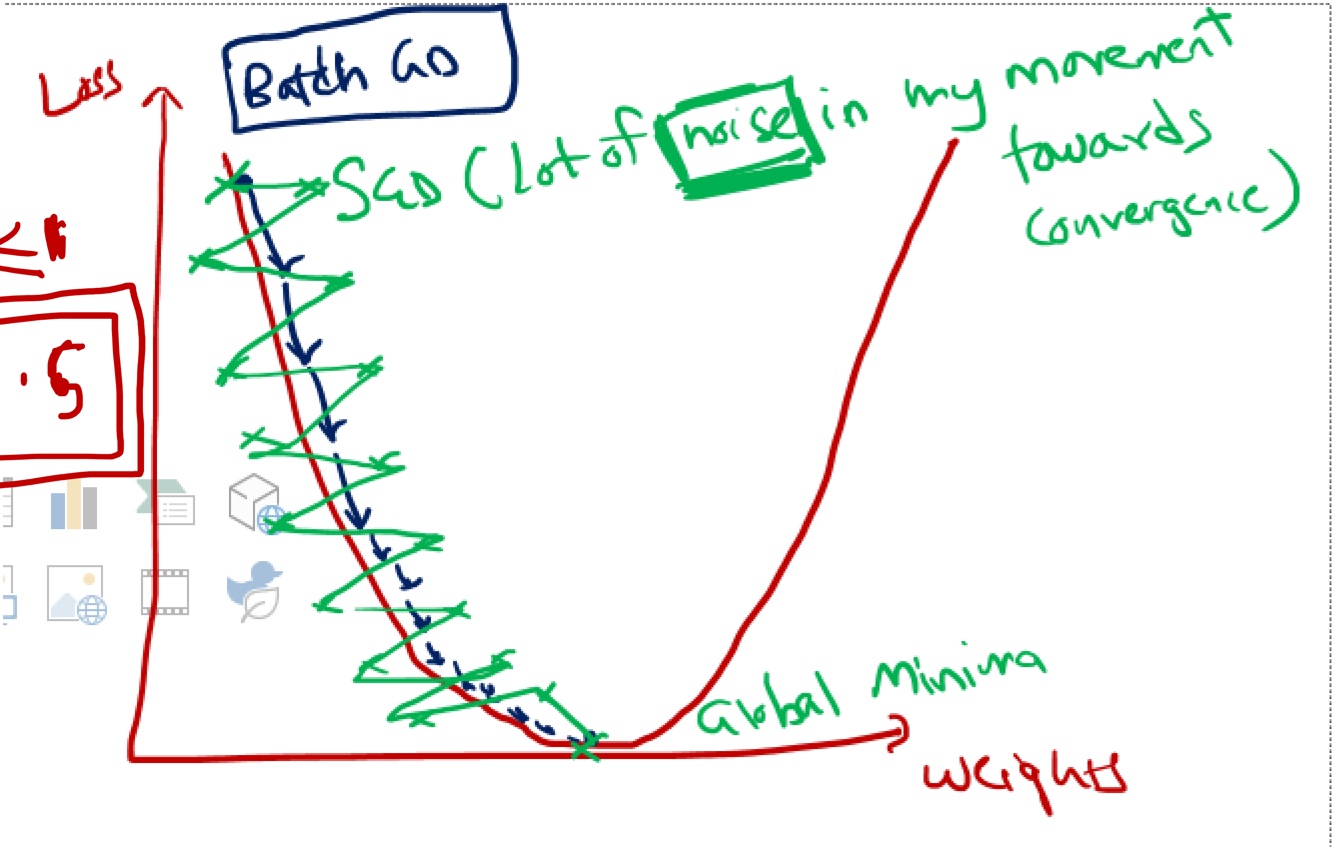
$t=2$ $V_{t_2} = \gamma \cdot V_{t_1} + b_2$ — (b)

$$\boxed{V_{t_2} = \underbrace{0.5 \times b_1}_{50\%} + \underbrace{b_2}_{100\%}}$$

$t=3$, $V_{t_3} = \gamma \cdot V_{t_2} + b_3$

$= \gamma(\gamma \cdot V_{t_1} + b_2) + b_3$

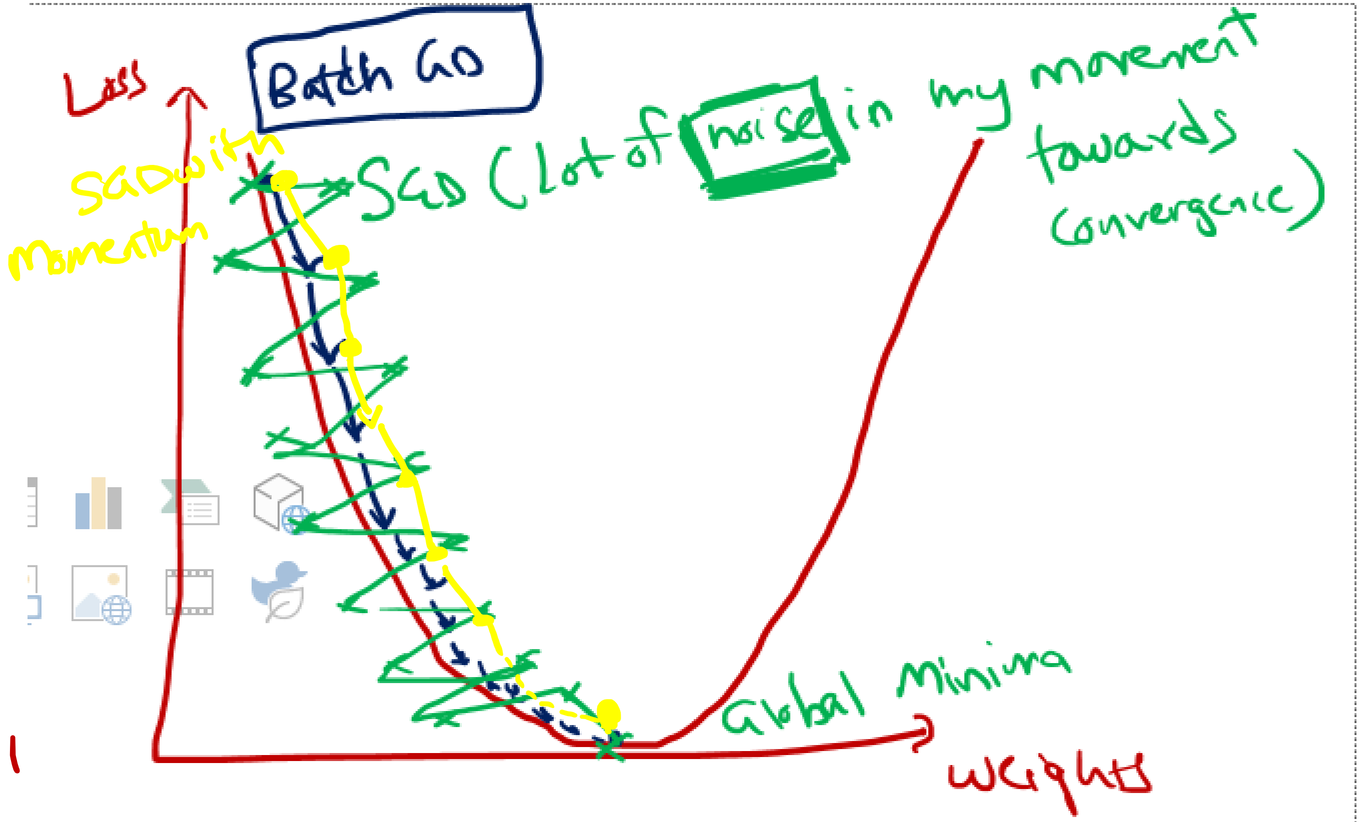
$$\boxed{V_{t_3} = \gamma^2 \cdot b_1 + \gamma b_2 + b_3} \text{ — (c)} \rightarrow V_{t_3} = \underbrace{0.25 \cdot b_1}_{25\%} + \underbrace{0.5\, b_2}_{50\%} + \underbrace{b_3}_{100\%}$$



# Stochastic Gradient Descent with Momentum

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

$$= w_{old} - \left[ \gamma \cdot V_{t-1} + \eta \times \frac{\partial L}{\partial w_{old}} \right]$$




$$\boxed{\gamma = 0.5}$$

$$V_{t-1} = 1 \times \left(\frac{\partial L}{\partial w_{old}}\right)_t + \gamma \left(\frac{\partial L}{w_{old}}\right)_{t-1} + \gamma^2 \left(\frac{\partial L}{\partial w_{old}}\right)_{t-2} + \gamma^3 \left(\frac{\partial L}{\partial w_{old}}\right)_{t-3} + \dots + \gamma^n \left(\frac{\partial L}{\partial w_{old}}\right)_{t-n}$$

highest imp.

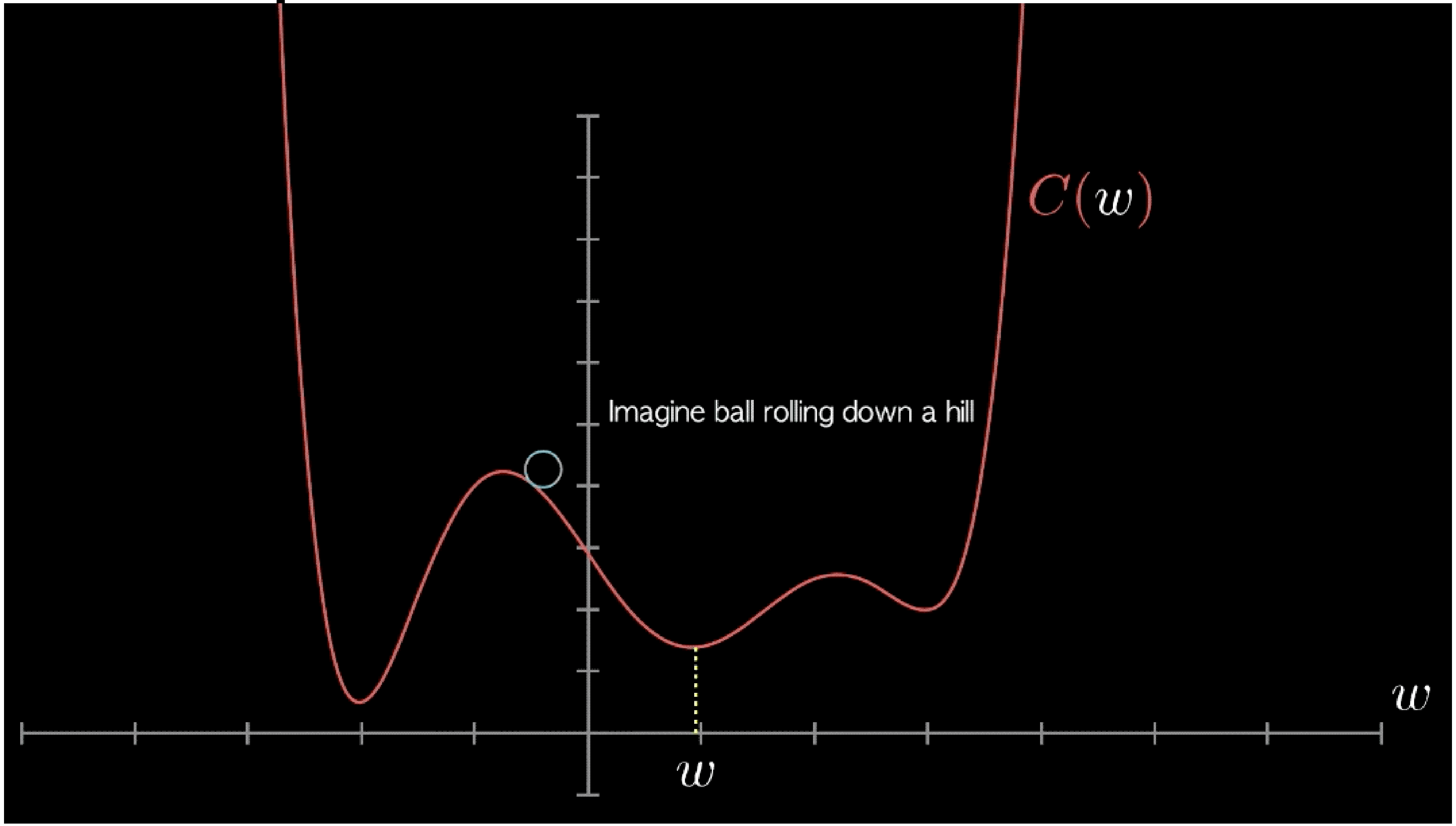
lowest imp.

bcz of $\gamma$ controlled.

# Momentum Optimizer

- Simply put, the momentum algorithm helps us progress faster in the neural network, negatively or positively, to the ball analogy. This helps us get to a local minimum faster.
- Motivation for momentum
- For each time we roll the ball down the hill (for each epoch), the ball rolls faster towards the local minima in the next iteration. This makes us more likely to reach a better local minima (or perhaps global minima) than we could have with SGD.

# Momentum Optimizer

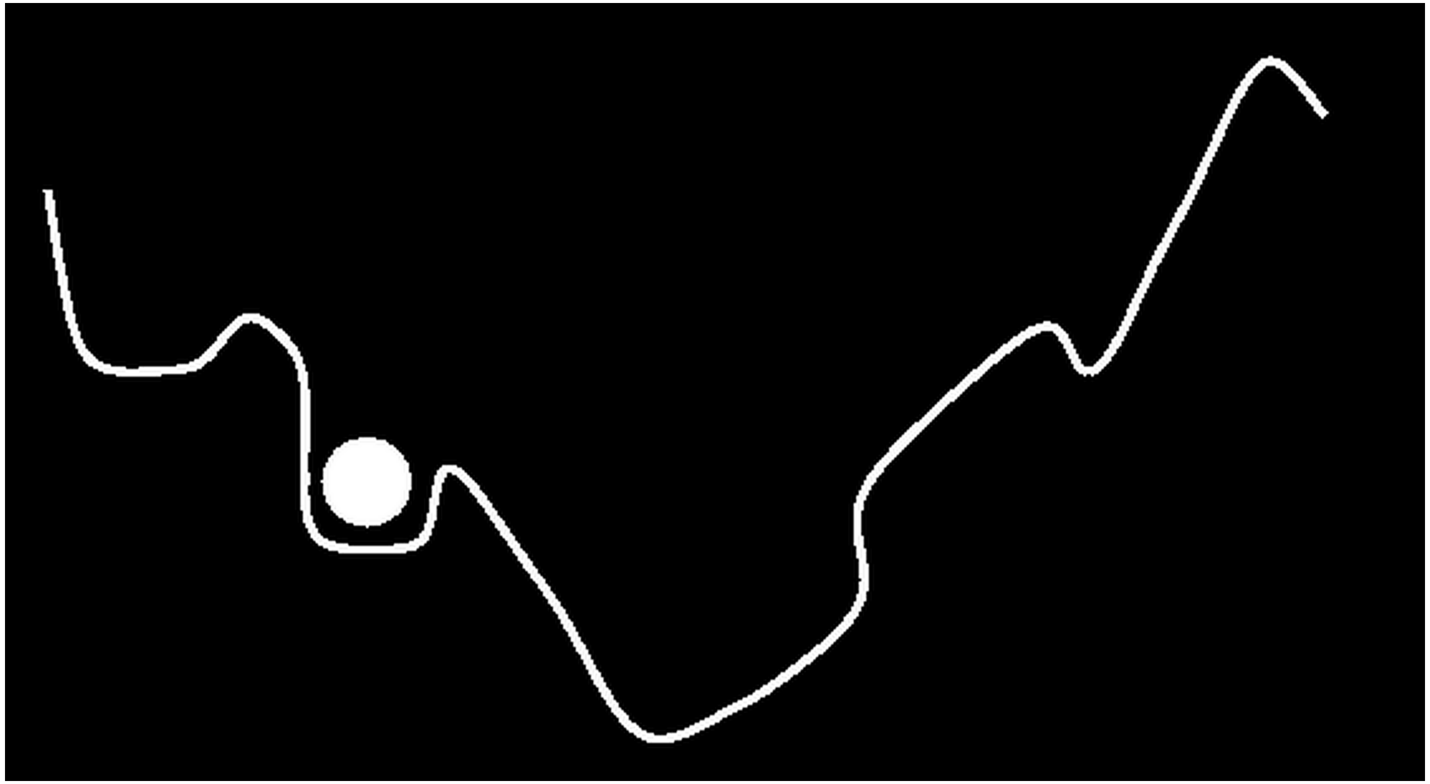



When optimizing the cost function for a weight, we might imagine a ball rolling down a hill amongst many hills. We hope that we get to some form of optimum.

# Momentum Optimizer

- The slope of the cost function is not actually such a smooth curve, but it's easier to plot to show the concept of the ball rolling down the hill.
- The function will often be much more complex, hence we might actually get stuck in a local minimum or significantly slowed down.
- Obviously, this is not desirable.
- The terrain is not smooth, it has obstacles and weird shapes in very high-dimensional space – for instance, the concept would look like this in 2D:

# Momentum Optimizer



- In the above case, we are stuck at a local minimum, and the motivation is clear – we need a method to handle these situations, perhaps to never get stuck in the first place.

# Momentum Optimizer

- Now we know why we should use momentum, let's introduce more specifically what it means, by explaining the mathematics behind it.
- **Explanation of momentum**
- Momentum is where we add a temporal element into our equation for updating the parameters of a neural network – that is, an element of time.

# Momentum Optimizer

# Momentum Optimizer

- Let's add those elements now. the temporal element, the explanation of vtvt.
- If you want to play with momentum and learning rate, I recommend visiting distill's page for Why Momentum Really Works.
- https://distill.pub/2017/momentum/

SGD with

# Momentum Optimizer

## Pros

Faster Convergence than Traditional SGD

## Cons

If momentum is too much, we get stuck in Local Minima

# Momentum Optimizer

- Used in conjunction Stochastic Gradient Descent (sgd) or Mini-Batch Gradient Descent, Momentum takes into account past gradients to smooth out the update. This is seen in variable v which is an exponentially weighted average of the gradient on previous steps. This results in minimizing oscillations and faster convergence.

$$v_{dW} = \beta v_{dW} + (1 - \beta) \frac{\partial \mathcal{J}}{\partial W}$$
$$W = W - \alpha v_{dW}$$

**Note**

- $v$ - the exponentially weighted average of past gradients
- $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer weight tensor
- $W$ - weight tensor
- $\beta$ - hyperparameter to be tuned
- $\alpha$ - the learning rate

Best

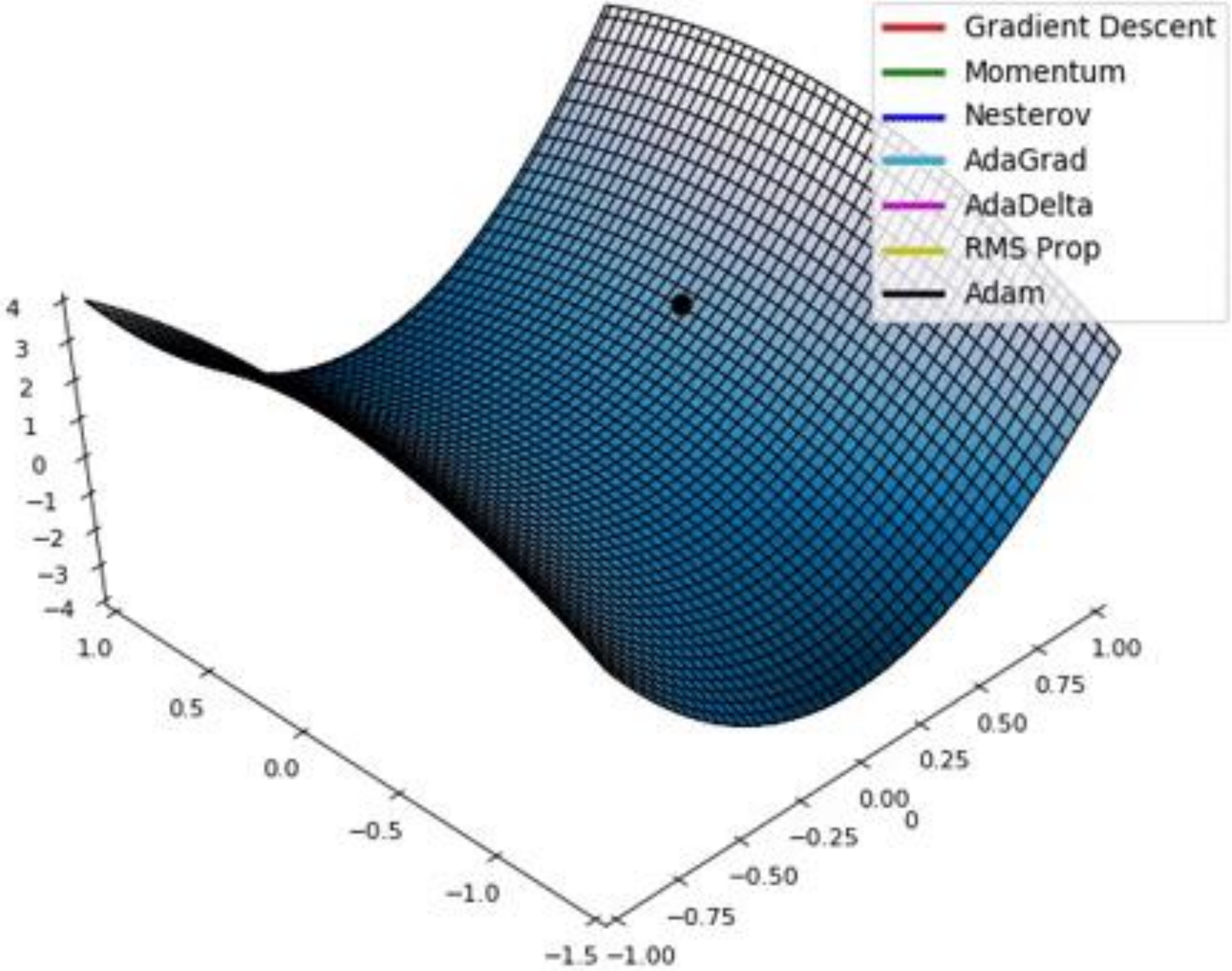
# Adam Optimizer ( Overcome disadv. of both Adagrad & RMS Prop )

- Adaptive Moment Estimation (Adam) is the next optimizer, and probably also the optimizer that performs the best on average. Taking a big step forward from the SGD algorithm to explain Adam does require some explanation of some clever techniques from other algorithms adopted in Adam, as well as the unique approaches Adam brings.

  SGD + Momentum + Adaptive η

- Adam uses Momentum and Adaptive Learning Rates to converge faster. We have already explored what Momentum means, now we are going to explore what adaptive learning rates means.

# Adam Optimizer



# Adam Optimizer

- Adaptive Moment Estimation (Adam) combines ideas from both RMSProp and Momentum. It computes adaptive learning rates for each parameter and works as follows.
- First, it computes the exponentially weighted average of past gradients $(v_{dW})$.
- Second, it computes the exponentially weighted average of the squares of past gradients $(s_{dW})$.
- Third, these averages have a bias towards zero and to counteract this a bias correction is applied $(v^{corrected}_{dW}, s^{corrected}_{dW})$.

# Adam Optimizer

- Lastly, the parameters are updated using the information from the calculated averages.

$$v_{dW} = \beta_1 v_{dW} + (1 - \beta_1)\frac{\partial \mathcal{J}}{\partial W}$$

$$s_{dW} = \beta_2 s_{dW} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W})^2$$

$$v^{corrected}_{dW} = \frac{v_{dW}}{1 - (\beta_1)^t}$$

$$s^{corrected}_{dW} = \frac{s_{dW}}{1 - (\beta_1)^t}$$

$$W = W - \alpha\frac{v^{corrected}_{dW}}{\sqrt{s^{corrected}_{dW}} + \varepsilon}$$

**Note**

- $v_{dW}$ - the exponentially weighted average of past gradients
- $s_{dW}$ - the exponentially weighted average of past squares of gradients
- $\beta_1$ - hyperparameter to be tuned
- $\beta_2$ - hyperparameter to be tuned
- $\frac{\partial \mathcal{J}}{\partial W}$ - cost gradient with respect to current layer
- $W$ - the weight matrix (parameter to be updated)
- $\alpha$ - the learning rate
- $\epsilon$ - very small value to avoid dividing by zero

# Adam Optimizer

- Adaptive Moment Estimation (Adam) combines ideas from both RMSProp and Momentum. It computes adaptive learning rates for each parameter and works as follows.
- First, it computes the exponentially weighted average of past gradients $(v_{dW})$.
- Second, it computes the exponentially weighted average of the squares of past gradients $(s_{dW})$.
- Third, these averages have a bias towards zero and to counteract this a bias correction is applied $(v_{dW}^{corrected}, s_{dW}^{corrected})$.