DEEP LEARNING

Trainer : Dr. Darshan Ingle

4. Hinge Loss

- Used for classification.

- **Code**

```
def Hinge(yHat, y):
    return np.max(0, 1 - yHat * y)
```

5. Huber Loss

- Typically used for regression.
- It’s less sensitive to outliers than the MSE as it treats error as square only inside an interval.

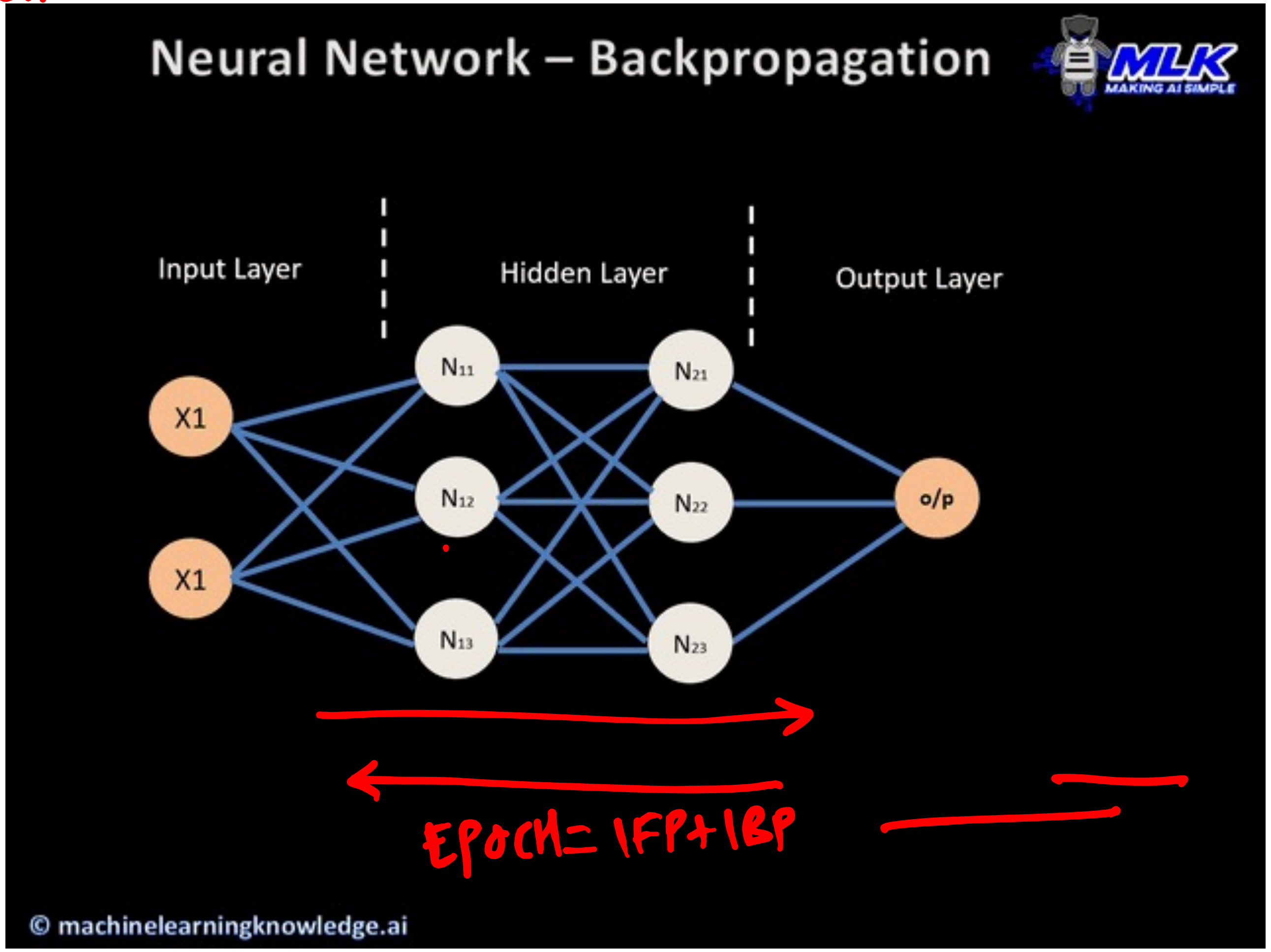
$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & if\,|(y - \hat{y})| < \delta \\ \delta((y - \hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

Code

```
def Huber(yHat, y, delta=1.):
    return np.where(np.abs(y-yHat) < delta,.5*(y-yHat)**2 , delta*(np.abs(y-yHat)-0.5*delta))
```

1. Gradient Descent (all points)
2. Mini Batch GD (one batch)
3. Stochastic GD (one points)

Optimizers



Optimizers

- **What is Optimizer ?**
- 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?

Ans: Optimizers. They tie together the loss function & model parameters by updating the model in response to o/p of Loss function.

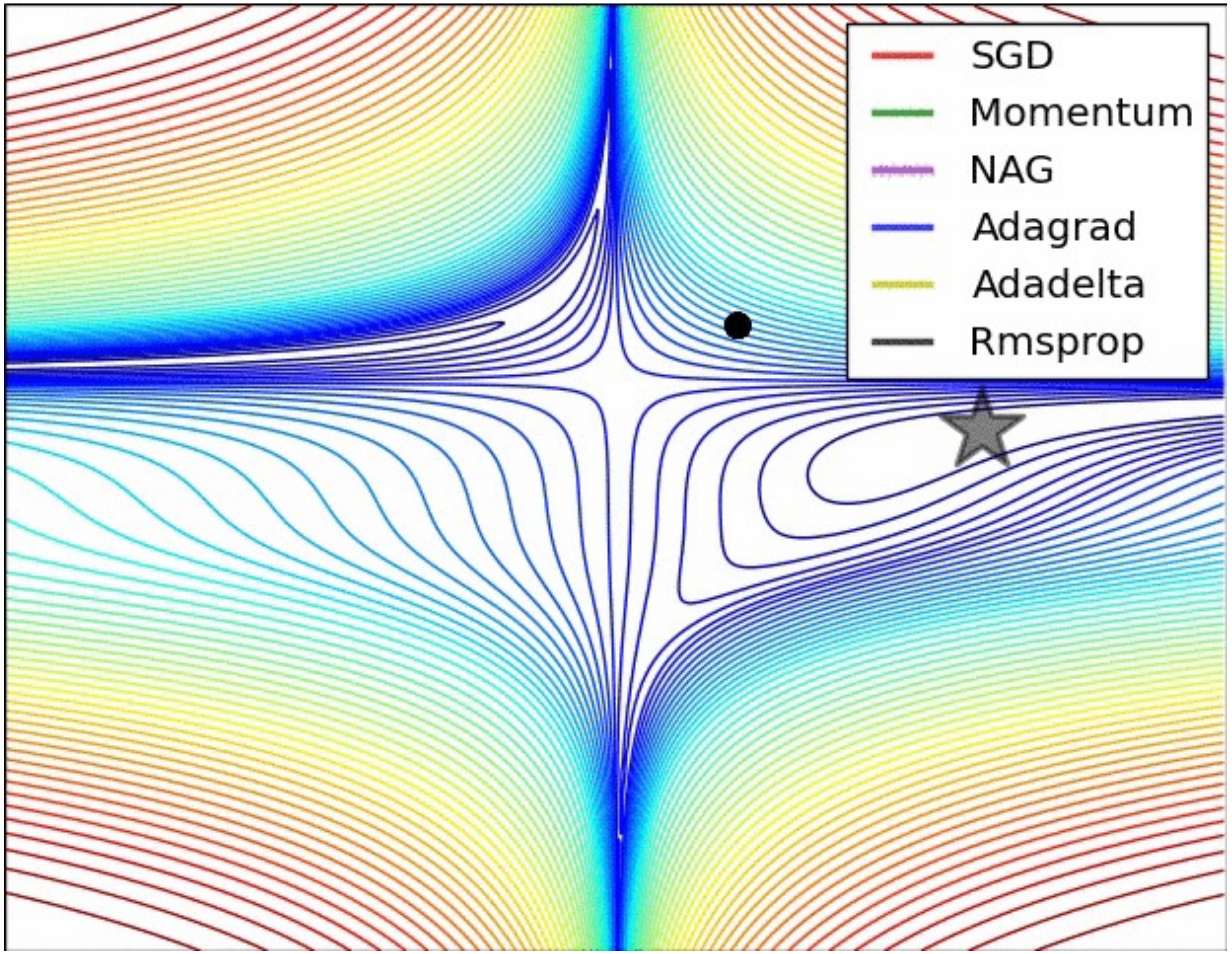
Optimizers

- Below are list of example optimizers

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

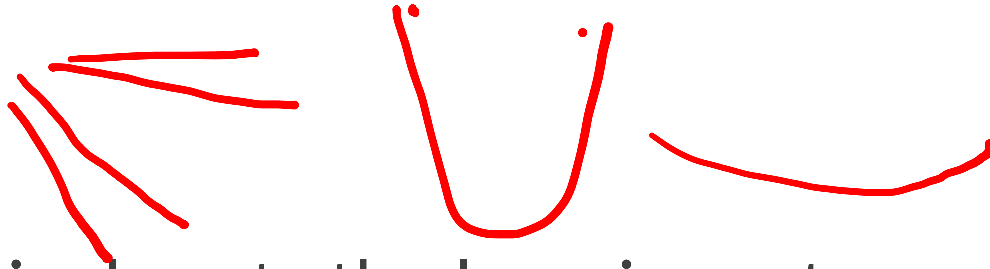
Optimizers

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



Adagrad Optimizer

usually we set it to $\boxed{\eta = 0.001}$



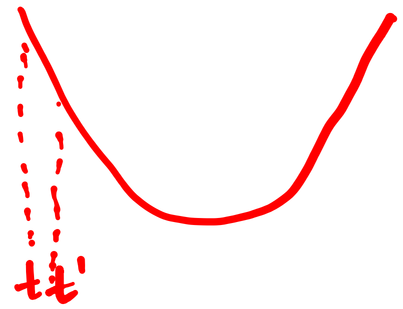
- Adagrad (short for adaptive gradient) adaptively sets the learning rate according to a parameter.

① Parameters that hv a higher gradient / frequent updates should hv slower Learning rate so that we dont overshoot the minimum value.

② —— " —— a low gradient / infrequent updates —" —— faster L.R so that they get trained quickly.

Adagrad Optimizer

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



$\eta = 0.01$

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

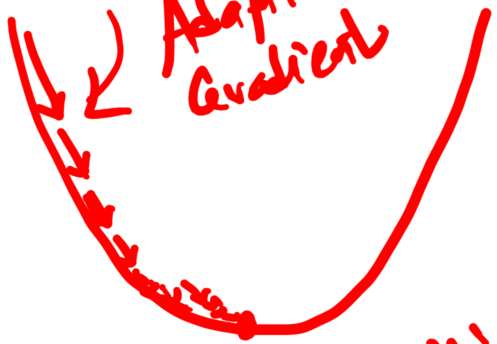
Adagrad:

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

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

→ small +ve no.

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



Global Minima.

Adagrad Optimizer

Adagrad Optimizer Disadvantage

$$w_t = w_{t-1} - \eta_t' * \frac{dL}{dw_{t-1}}$$

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

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{dL}{dw_i}\right)^2$$

We are squaring in α_t. It is possible that α_t becomes a v.v.v. high no. as much the # iterations increase.

$\therefore \eta_t'$ is v.v.v. small.

$\therefore w_t$ & w_{t-1} are almost same.

i.e. no weight updation happens.

RMSProp and Adadelta

RMSProp and Adadelta (They overcome disadv. of Adagrad)

→ are almost the same. Only difference is that they are created by different team.

They simply aim to control α_t.

$\alpha_t \uparrow$, L.R. $\downarrow$. We want this, but we don't want it to decrease to a v.v. small no.

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

$$\boxed{w_t = w_{t-1} - \eta_t' \cdot \frac{\partial L}{\partial w_t}}$$

$$w_{avg_t} = \underbrace{\overset{0.95}{\gamma} \cdot w_{avg(t-1)}}_{\text{newer weight are given higher weightage due to } \gamma} + \underbrace{\overset{0.05}{(1-\gamma)} \cdot \left(\frac{\partial L}{\partial w_t}\right)^2}_{\text{older weights are given lesser weights}}$$

newer weight are given higher weightage due to γ & older weights are given lesser weights

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

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.

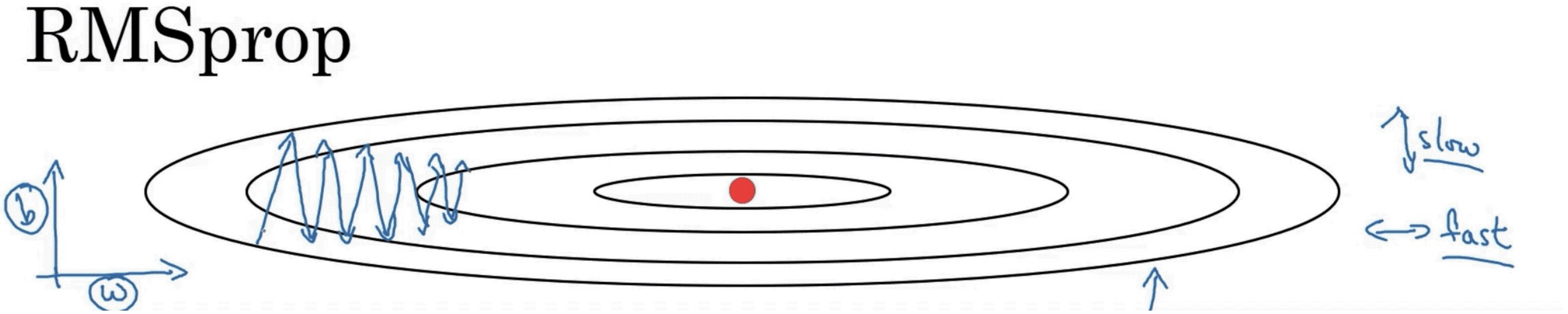
$$s_{dW} = \beta s_{dW} + (1-\beta)\left(\frac{\partial \mathcal{J}}{\partial W}\right)^2$$

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

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
- β - hyperparameter to be tuned
- α - the learning rate
- ϵ - 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.

α - the learning rate

ϵ - 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
```

additional info

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,

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

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

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

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

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
```

Gradient Descent (Vanilla | Plain GD)

Stochastic Gradient Descent

already covered earlier.

Stochastic Gradient Descent

Stochastic Gradient Descent

Pros ① Relatively fast compared to older GD approaches.

② easier to learn for beginners, ∵ it is not math heavy.

Cons

① Converges slow than newer algorithms

② Has more problems with being stuck in a local minimum than newer approaches.

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

Stochastic Gradient Descent with Momentum

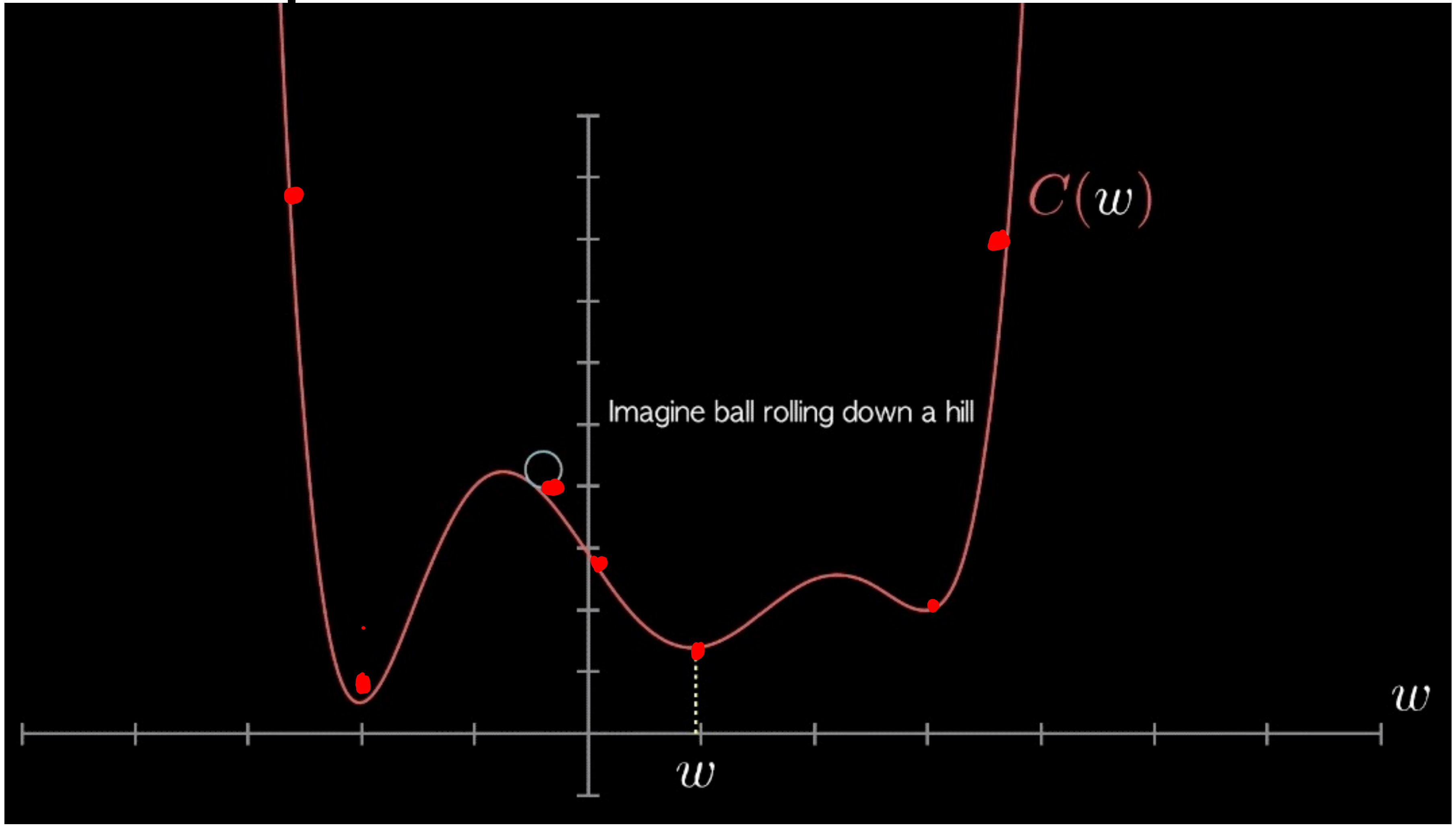
Stochastic Gradient Descent with Momentum

Stochastic Gradient Descent with Momentum

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

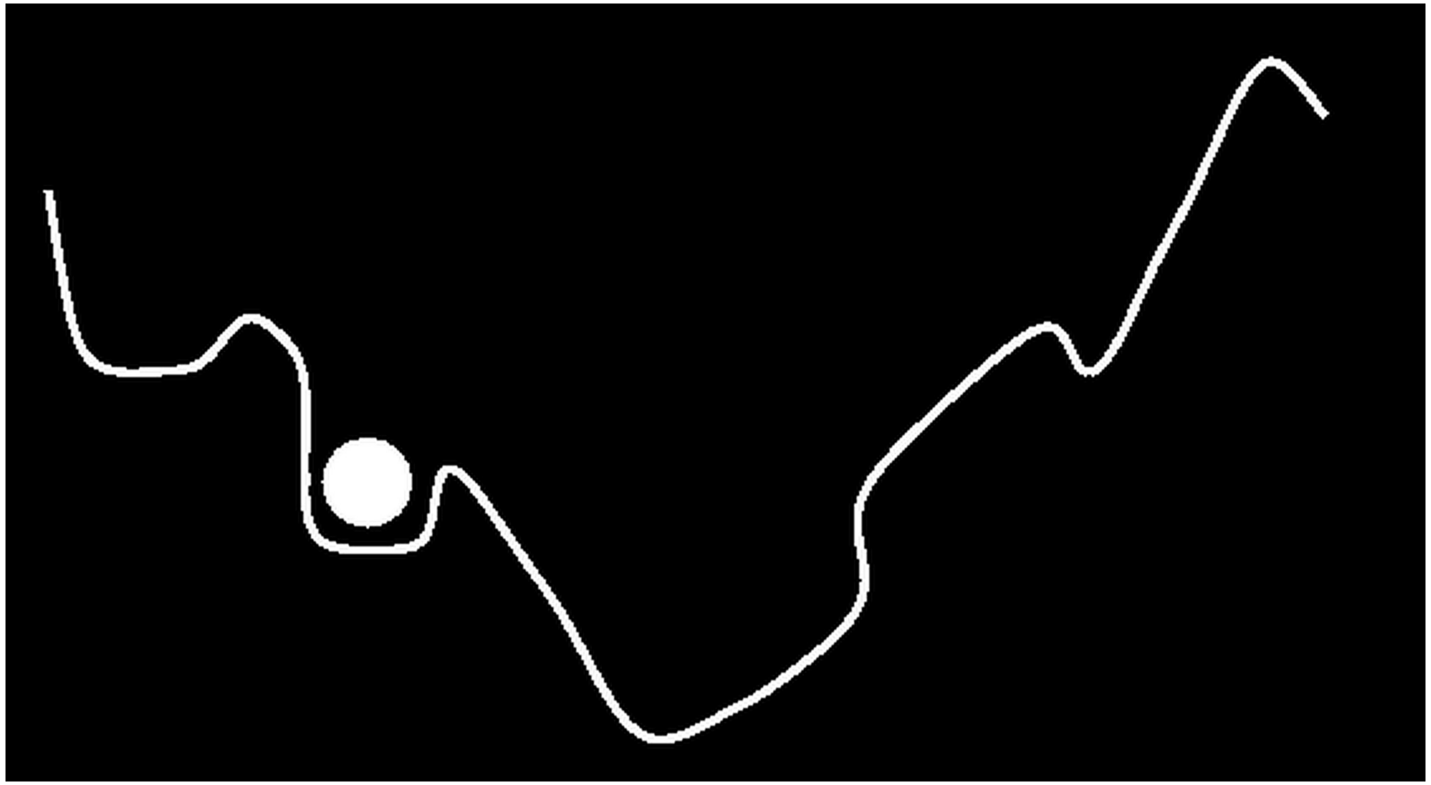


When optimizing the cost function for a weight, we might imagine a ball rolling down a hill amongst many hills. We hope that we get to some form of optimum.

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/

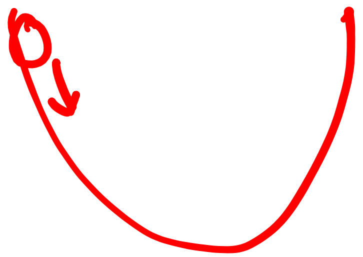
Momentum Optimizer

Pros

① Faster convergence than Traditional SGD.

Cons

①



If Momentum is too much, we will most likely miss 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.

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

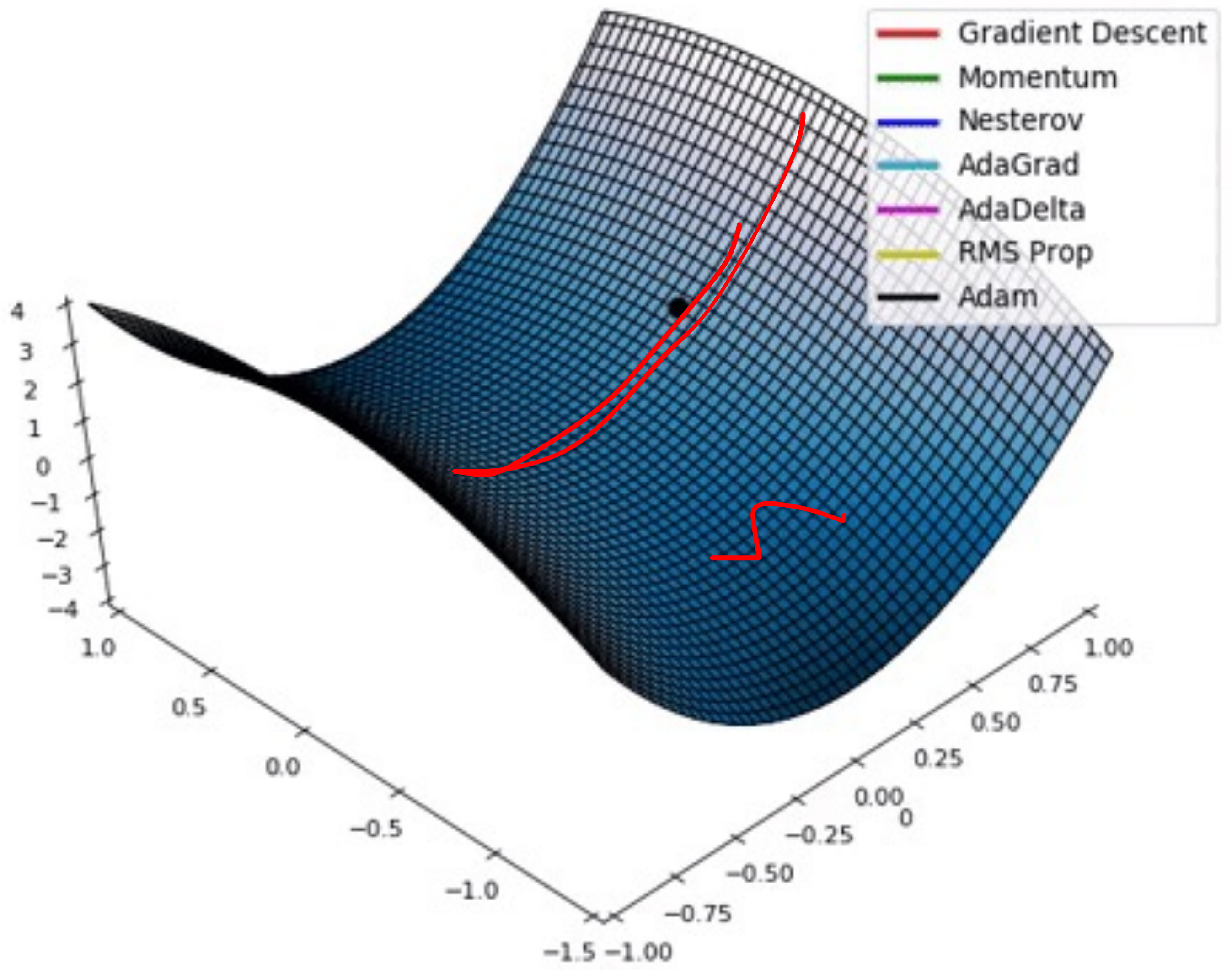
Note

- 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
- β - hyperparameter to be tuned
- α - the learning rate

Adam Optimizer

- 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.
- 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 $(v_{dW}^{corrected}, s_{dW}^{corrected})$.

Adam Optimizer

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

$\gamma = 0.95$

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

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

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

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

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

Note

- v_{dW} - the exponentially weighted average of past gradients
- s_{dW} - the exponentially weighted average of past squares of gradients
- β_1 - hyperparameter to be tuned
- β_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)
- α - the learning rate
- ϵ - 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 $(v_{dW}^{corrected}, s_{dW}^{corrected})$.