Optimization Algorithms

- Optimization algorithms train neural networks faster by adjusting weights and learning rate
- Training deep neural networks on large dataset is slow
- Fast optimization algorithms can speed up training process

https://ruder.io/optimizing-gradient-descent/

Gradient Descent

- Gradient descent adjust the parameters iteratively to minimize cost function

$$w_{new} = w - \alpha \frac{\partial J}{\partial w}$$

$$b_{new} = b - \alpha \frac{\partial J}{\partial b}$$

$$\theta = \begin{bmatrix} w \\ b \end{bmatrix}$$

$$\theta = \theta - \alpha . \nabla_\theta J(\theta)$$

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J}{\partial w} \\ \frac{\partial J}{\partial b} \end{bmatrix}$$

https://ruder.io/optimizing-gradient-descent/

Gradient Descent

- If we have dataset with 50,00,000 training examples

$$X = \begin{bmatrix} x^1 \\ x^2 \\ \cdot \\ \cdot \\ \cdot \\ x^m \end{bmatrix}$$

1. Will take more time for the convergence
2. Large memory to caliculate the gradient
3. May trap at local minimum

https://ruder.io/optimizing-gradient-descent/

Stochastic Gradient Descent

- Caliculate gradient based on single training example

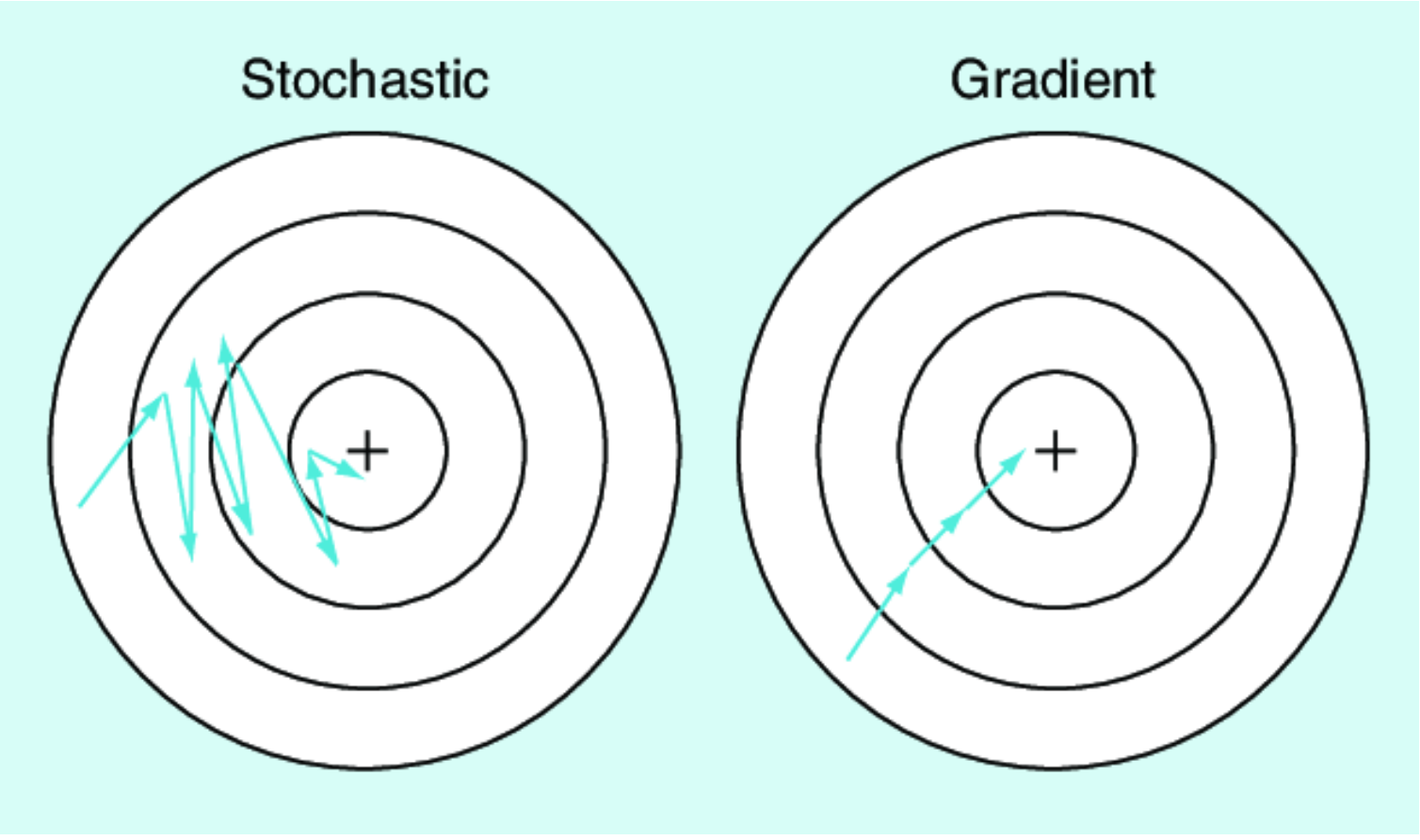
$$\theta = \theta - \alpha . \nabla_{\theta} J\left(\theta; x^{(i)}; y^{(i)}\right)$$

$$X = \begin{bmatrix} x^{1} \\ x^{2} \\ . \\ . \\ . \\ x^{m} \end{bmatrix}$$

1. Less memory to caliculate the gradient
2. Faster than Batch Gradient Descent but it lose speed up from vectorization
3. May trap at local minimum

https://ruder.io/optimizing-gradient-descent/

SGD vs GD



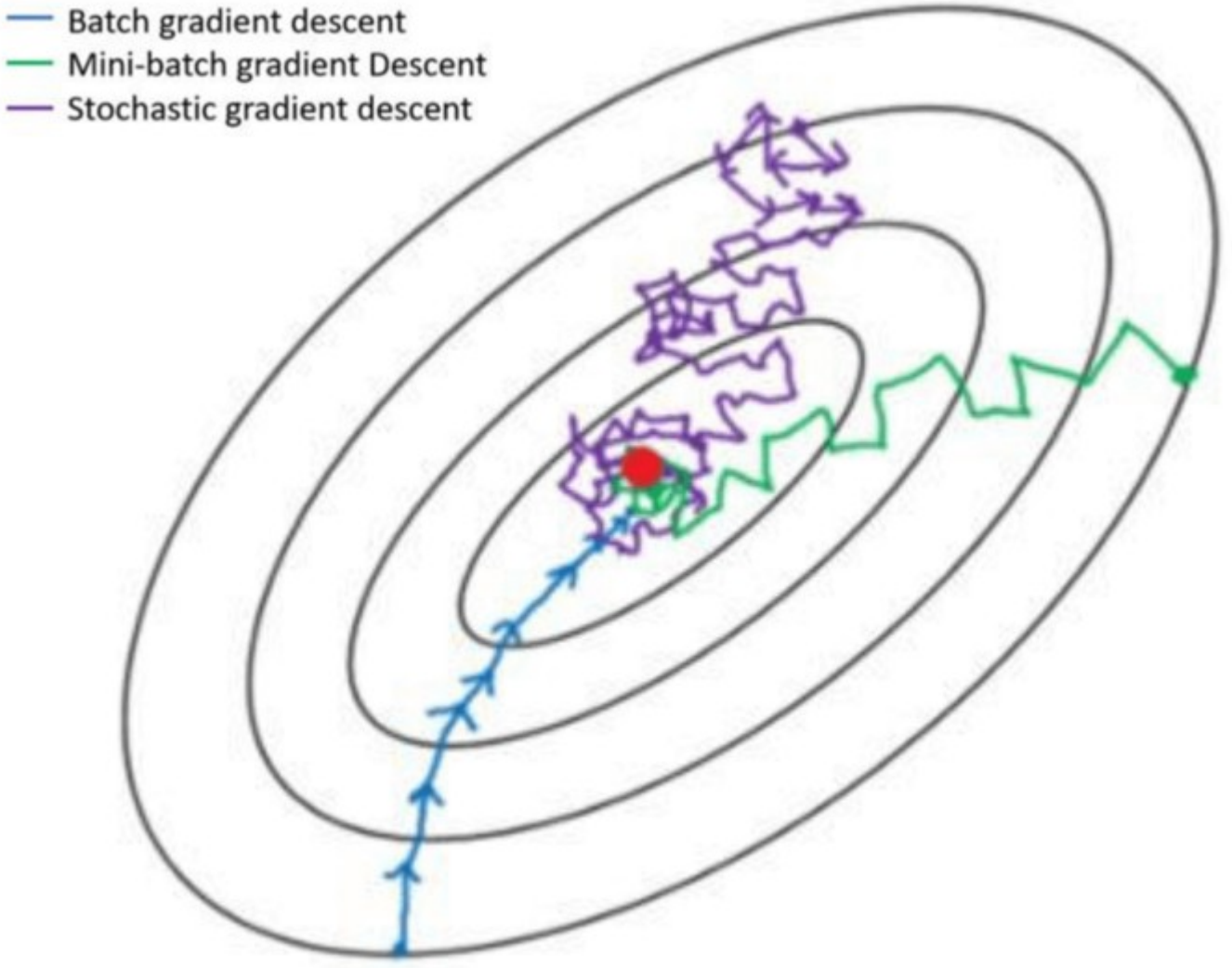
Mini-Batch Gradient Descent

- Caliculate gradient based on subset of training example

$$\theta = \theta - \alpha . \nabla_{\theta} J\left(\theta; x^{(i:i+n)}; y^{(i:i+n)}\right)$$

$$X = \begin{bmatrix} x^{1} \\ x^{2} \\ . \\ . \\ . \\ x^{m} \end{bmatrix}$$

https://ruder.io/optimizing-gradient-descent/


Batch gradient descent
Mini-batch gradient Descent
Stochastic gradient descent

Mini-Batch Gradient Descent

- Pros
  - Takes less noisy steps than SGD
  - Can utilize vectorization speedup
  - Takes less memory than GD
- Cons
  - Takes noisy steps compared to GD
  - Takes longer time compared to full batch GD
  - May stuck at local minimum

https://ruder.io/optimizing-gradient-descent/

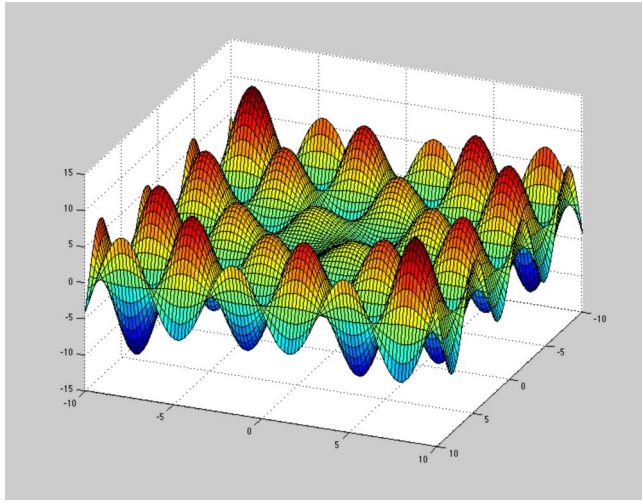
Mini-Batch Gradient Descent

Optimizer	Update Speed	Memory Usage
Batch Gradient Descent	Slow	High
Stochastic Gradient Descent	High	Low
Batch Gradient Descent	Medium	Medium

https://ruder.io/optimizing-gradient-descent/

Challenges

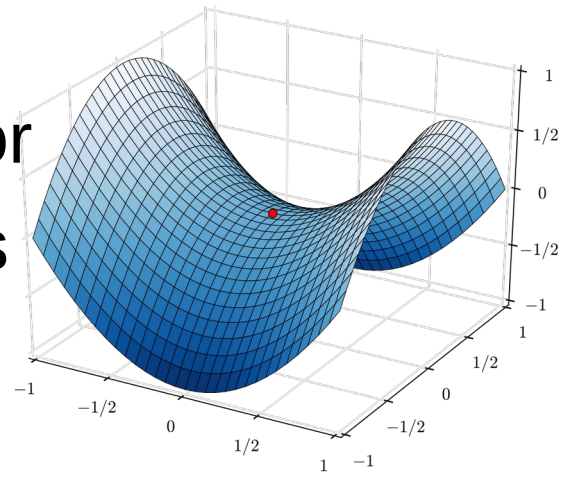
- **Local Minima**
  - With non-convex functions, such as neural nets, it is possible to have many local minima.
  - The gradients may change dramatically on complex loss functions
  - Local minima can be problematic if they have high cost in comparison to the global minimum.



https://ruder.io/optimizing-gradient-descent/

Challenges

- **Plateaus, Saddle Points and Other Flat Regions**
- These saddle points are usually surrounded by a plateau of the same error
- Plateau is the region where the surface is almost flat and the gradient is close to zero.
- which makes it notoriously hard for GD to escape, as the gradient is close to zero in all dimensions.



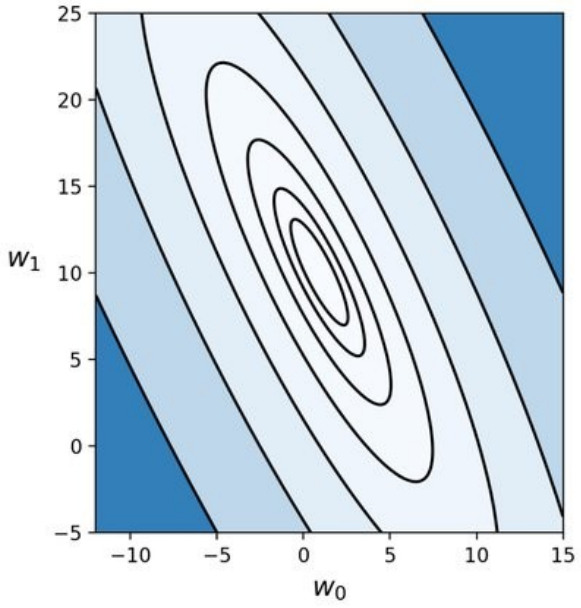
https://ruder.io/optimizing-gradient-descent/

Challenges

Choosing a proper learning rate can be difficult.

Challenges

- **Same learning rate applies to all parameter updates.**
- If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent.



https://ruder.io/optimizing-gradient-descent/

Challenges

- Vanilla GD it works based on only Gradients
  - it can't be fast
  - can't escape from local minima, saddle points and flat regions

$$w_{new} = w - \alpha \frac{\partial J}{\partial w}$$

- **A good algorithm finds the minimum fast and reliably well (i.e. it doesn't get stuck in local minima, saddle points, or plateau regions), but rather goes for the global minimum**

https://ruder.io/optimizing-gradient-descent/

Gradient Descent Optimization Algorithms

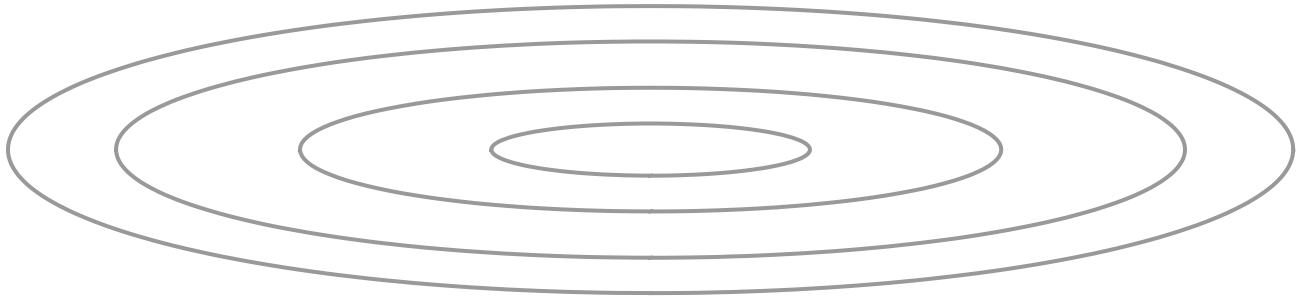
- Gradient descent variants
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent

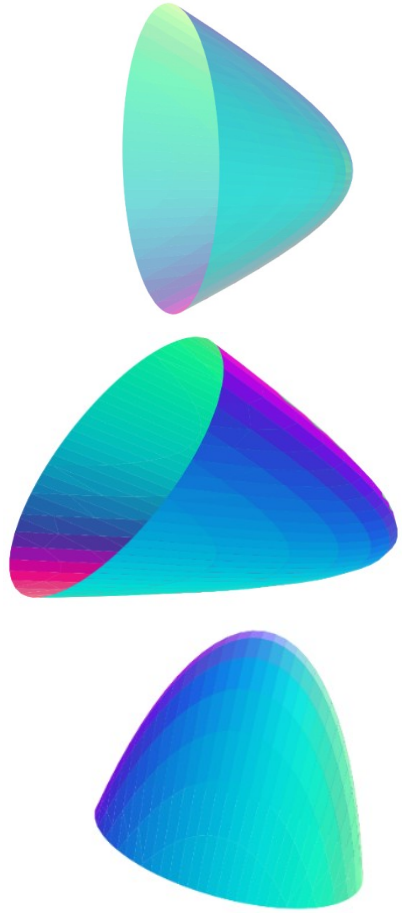
- **Gradient descent optimization algorithms**
  - Momentum
  - Adagrad
  - RMSprop
  - Adam

https://ruder.io/optimizing-gradient-descent/

Gradient Descent with Momentum

- the surface curves much more steeply in one dimension than in another



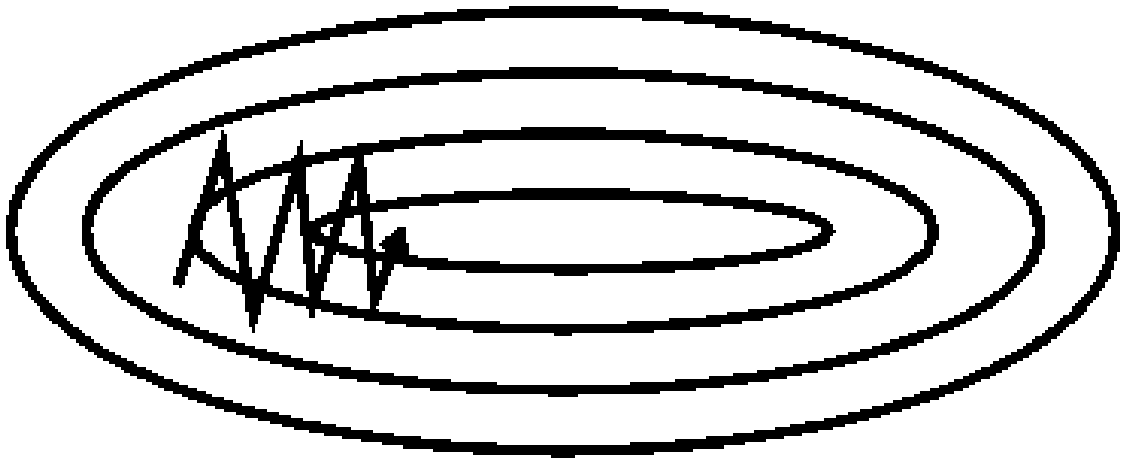


A contour plot is a graphical technique for representing a 3-dimensional surface by plotting constant z slices, called contours, on a 2-dimensional format.

https://ruder.io/optimizing-gradient-descent/

Gradient Descent with Momentum

- In these scenarios, GD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as shown in image



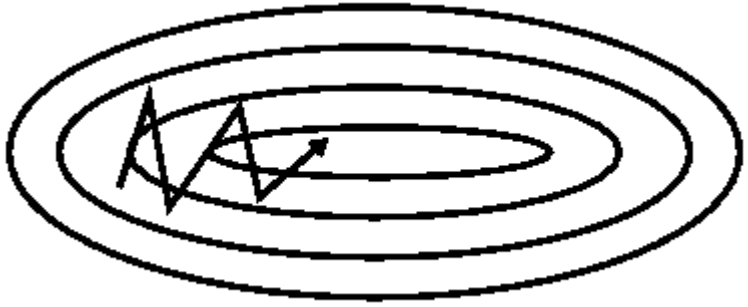
https://ruder.io/optimizing-gradient-descent/

Gradient Descent with Momentum

- Momentum is a method that helps accelerate GD in the relevant direction and dampens oscillations.
- Adding a fraction γ of the update vector of the past time step to the current update vector
- The momentum term γ is usually set to 0.9 or a similar value.

$$v_t = \gamma v_{t-1} + \alpha . \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$



https://ruder.io/optimizing-gradient-descent/

Gradient Descent with Momentum

- The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.
- As a result, we **gain faster convergence and reduced oscillation**.
- When we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way. The same thing happens to our parameter updates:

https://ruder.io/optimizing-gradient-descent/

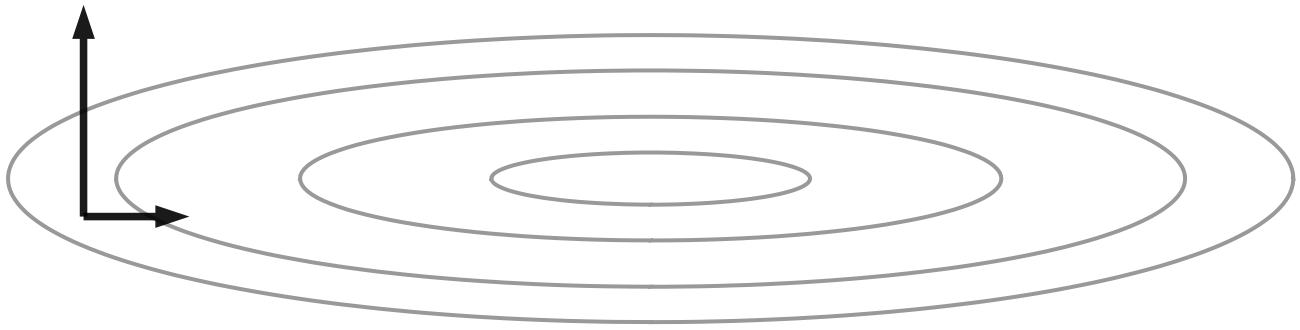
Problem with Momentum

- Learning rate should set manually.
- Same learning rate will be used for all parameters(in all dimentions).
- We may require small learning rate in some dimention and large learning rate in other dimention

https://ruder.io/optimizing-gradient-descent/

Motivation for Adagrad

- Consider two dimentional optimization problem

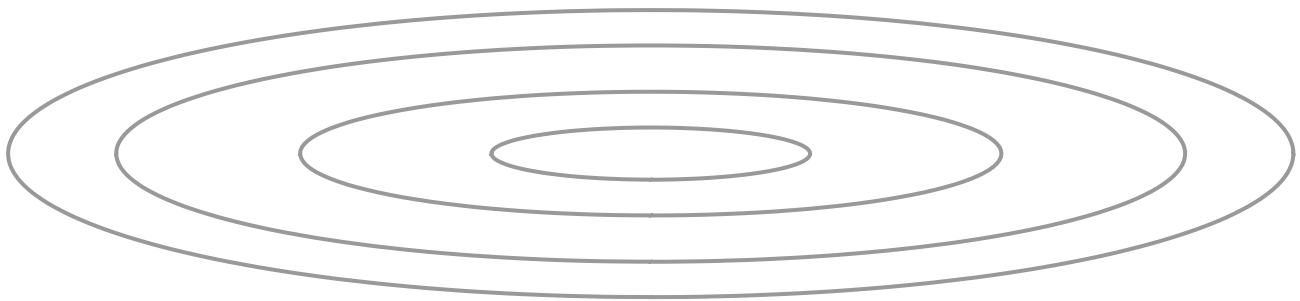


Motivation for Adagrad

- Consider two dimentional optimization problem

Adagrad

- Adagrad uses a different learning rate for every parameter θ_i at every time step t



- It adapts lower learning rate for the parameters having larger gradients
- high learning rates for parameters having smaller gradients.

https://ruder.io/optimizing-gradient-descent/

Adagrad

- At time t, it maintain the squared sum of past gradients .

$$s_t = s_{t-1} + g_t^2$$

- Update rule

$$\theta_{t+1} = \theta_t - \frac{\alpha_t}{\sqrt{s_t + \epsilon}} \odot g_t$$

$$\theta = \theta - \alpha . \nabla_\theta J(\theta)$$
$$\theta = \theta - \alpha . g$$

- Scale factor of a parameter is inversely proportional to the square root of squared sum of past gradients

https://ruder.io/optimizing-gradient-descent/

Adagrad

- One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01.
- Adagrad's main weakness is its accumulation of the squared gradients in the denominator:
- Since every added term is positive, the accumulated sum keeps growing during training.
- This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

https://ruder.io/optimizing-gradient-descent/

RMSprop(Root Mean Squared Propagation)

- In RMSprop is also the learning rate is adapted for each of the parameters.
- RMSprop have been developed to resolve Adagrad's radically diminishing learning rates.
- RMSprop divides the learning rate by an **exponentially decaying average of squared gradients**.

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t^2 \qquad s_t = s_{t-1} + g_t^2$$

- Replace the sum of the squares of the past gradients with the decaying average over past squared gradients

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{s_t + \epsilon}} g_t$$

https://ruder.io/optimizing-gradient-descent/

Adam

- Adaptive Moment Estimation (Adam) is also computes **adaptive learning rates** for each parameter.
- In addition to storing an exponentially decaying average of past squared gradients like RMSprop,
- Adam also keeps an exponentially decaying average of past gradients, similar to **momentum**.

https://ruder.io/optimizing-gradient-descent/

Adam

- We compute the decaying averages of past gradients v_t and and also past squared gradients s_t as follows
- $v_t = \beta_1 v_{t-1} + (1-\beta_1) g_t$
- $s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$
- As v_t and s_t are initialized as vectors of 0's,
- the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

https://ruder.io/optimizing-gradient-descent/

Adam

- They reduce these biases by computing bias-corrected first and second moment estimates:

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t} \qquad \hat{s}_t = \frac{s_t}{1-\beta_2^t}$$

- They then use these to update the parameters just as we have seen in RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{s}_t + \epsilon}} \hat{v}_t$$

https://ruder.io/optimizing-gradient-descent/

Adam

- It combines best features of adagrad and RMSprop
- It adds momentum and adaptive learning rate to the basic gradient descent.

- Adam is the best among adaptive optimizers in most of the cases

https://ruder.io/optimizing-gradient-descent/