APL 405: Machine Learning in Mechanics

Lecture 12: Parameter Optimization

by

Rajdip Nayek

Assistant Professor,
Applied Mechanics Department,
IIT Delhi

Instructor email: rajdipn@am.iitd.ac.in

Recap

- We looked at different types of loss functions for regression and classification
- **Learning** the parameters of a chosen parametric model often requires minimizing an appropriate loss funciton
- An ML engineer therefore needs to be familiar with some strategies to solve optimization problems
- In this lecture, we will introduce some ideas behind some of the optimization methods used in ML

Optimization in Machine Learning (ML)

- **Optimization** → Finding the minimum or maximum of an **objective function**

- A maximization problem can be formulated as a minimization problem

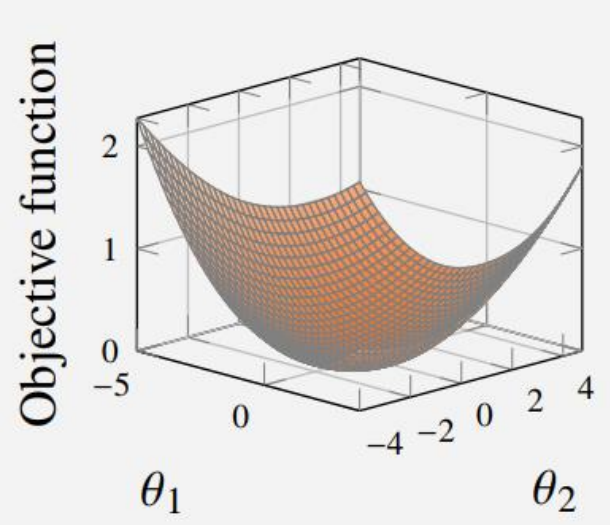
$$\widehat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} J(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} -J(\boldsymbol{\theta})$$

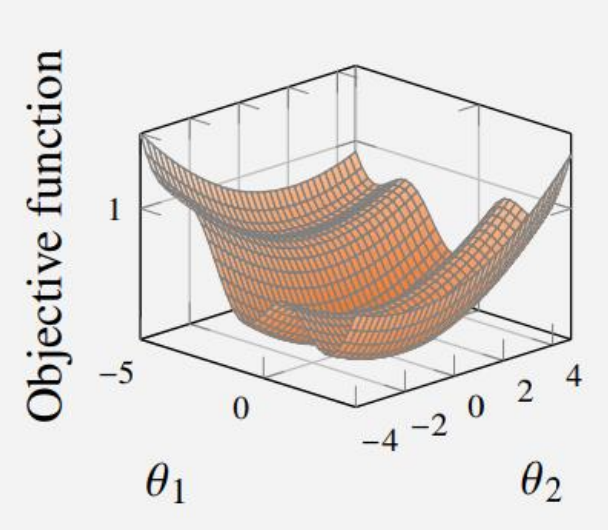
  So we will limit ourselves to minimization problems only

- Optimization in ML is primarily used in **two** ways:
  - **Training a model** by minimizing the cost function w.r.t. the model parameters
    - **Objective function** : $J(\boldsymbol{\theta})$
    - **Optimization variable**: $\boldsymbol{\theta}$
  - **Tuning hyperparameters**, such as the regularization parameter λ
    - **Objective function** : $E_{\text{hold-out}}$
    - **Optimization variable**: Hyperparameters (e.g. λ)

Convex functions

- An important class of objective functions are **convex functions**
- Optimization is *much easier* for convex objective functions, and it is a good idea to consider whether a non-convex optimization can be reformulated into a convex problem (but it is not always possible)
- Convex functions have unique minimum and no other local minima





- Examples of convex functions are cost functions for logistic regression and linear regression
- However, most problems in ML do not lead to convex functions

Convex functions

- An important class of objective functions are **convex functions**
- Convex functions are functions such that a straight line between any two points of the function lie above the function
- The function f is a convex function if for all x, y in the domain convex function of f, and for any scalar θ with $0 \leq \phi \leq 1$, we have

$$f(\phi x + (1-\phi)y) \leq \phi f(x) + (1-\phi)f(y)$$

- Furthermore, if f is a differentiable function, then we can specify convexity in terms of gradient $\nabla_x f(x)$

$$f(y) \geq f(x) + \nabla_x f(x) \cdot (y - x)$$

- If we further know that a function $f(x)$ is twice differentiable, that is, the Hessian (double derivative) exists for all values in the domain of x, then the function $f(x)$ is convex if and only if $\nabla_x^2 f(x)$ is positive semidefinite

Gradient of a cost function

- **Training examples:** $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \cdots, (\mathbf{x}_N, y_N)\}$

- Let's say the chosen model be: $y = f_{\boldsymbol{\theta}}(\boldsymbol{x})$

- **Cost function** → Average over individual training losses

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L\big(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i)\big)$$

- **Gradient of loss function** w.r.t. parameter $\boldsymbol{\theta}$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} L\big(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i)\big)$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Note: $\boldsymbol{\theta}$ would represent hyperparameters in the case of hyperparameter optimization

Gradient Descent

- **Gradient of loss function** w.r.t. parameter $\boldsymbol{\theta}$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

  - $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ has the same dimension as $\boldsymbol{\theta}$

  - $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ describes the direction in which $J(\boldsymbol{\theta})$ increases. Therefore, $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ describes the direction in which $J(\boldsymbol{\theta})$ decreases

- Taking a (small) step in the direction of the negative gradient will reduce the value of cost function

$$J\left(\boldsymbol{\theta} - \gamma \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\right) \leq J(\boldsymbol{\theta}) \quad \text{for some } \gamma > 0$$

- This suggests that if we have $\boldsymbol{\theta}^{(t)}$ and want to select $\boldsymbol{\theta}^{(t+1)}$ such that $J\left(\boldsymbol{\theta}^{(t+1)}\right) \leq J\left(\boldsymbol{\theta}^{(t)}\right)$, we should

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

γ is called the learning rate

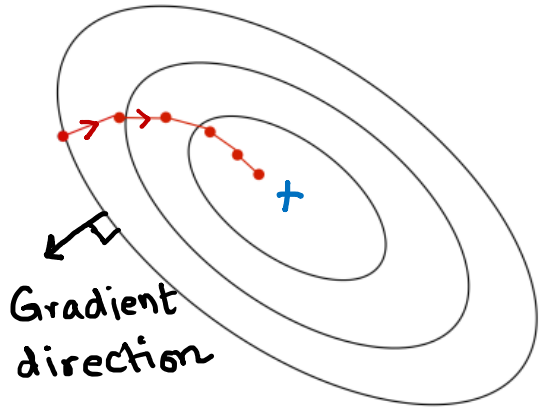
GD is a local optimizer, there is no guarantee that it will find the global minimum

Batch gradient descent (Batch GD)

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- Specify a learning rate, compute the total gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ by averaging over *all* individual loss function gradients for every training example, and then update the parameters $\boldsymbol{\theta}$
- The algorithm goes over the entire data once before updating the parameters
- This is known as **batch gradient descent (BGD)**, since we treat the entire training set as a batch
  - **Pros**: There is no approximation in gradient calculation. Each update step guarantees that the loss will decrease, if γ is small enough
  - **Cons**: However, Batch GD can be very time-consuming for a large datasets (very large N), due the summation over N datapoints



Batch gradient descent (Batch GD)

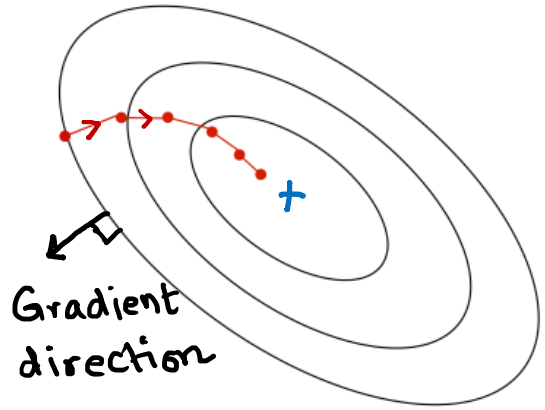
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- Batch gradient descent treat the entire training set as a single batch
- Updates the parameter vector after each full pass (**epoch**) over the entire dataset


```
theta   = -1        # initialize parameter vector
eta     = 0.001     # learning rate
epochs  = 100       # number of passes over entire dataset
Ntr     = 10000     # number of training points
for i in range(epochs):
  dtheta = 0        # initialize increment to zero
  for x,t in zip(X,T):
    dtheta += grad_theta(theta, x, t)

  theta = theta - eta * dtheta / Ntr
```



Stochastic gradient descent (SGD)

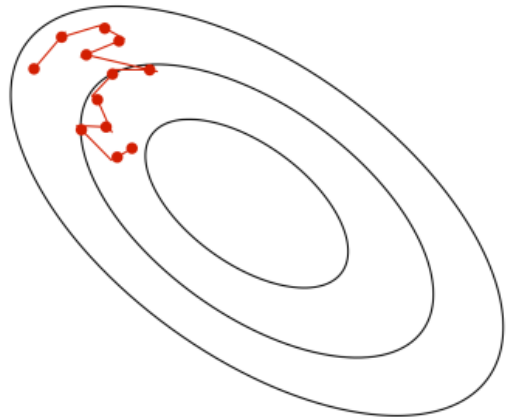
$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- When N is very large, the summation can involve summing a many terms
- Also, it can be an issue to keep all data points in the computer memory at the same time
- Subsampling a small set from the full training set might be more useful
- In **SGD**, one random samples (without replacement) a training pair $(\mathbf{x}_i, y_i)$ from the full training dataset, and

Updates $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} L^{(i)}\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

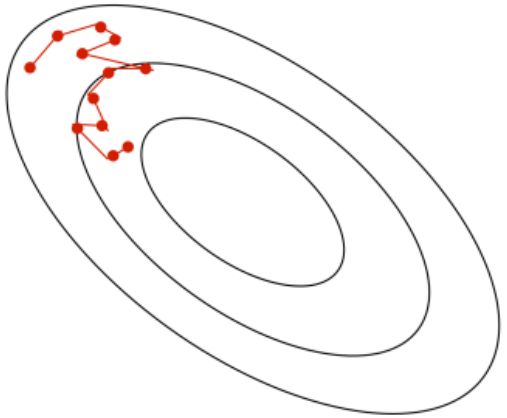
- **Pros**: SGD can make significant progress before it has even looked at the entire data!
- **Cons**: It uses an approximate estimate of gradient. There is no guarantee that each step will decrease the loss



Stochastic gradient descent (SGD)

- We see many **fluctuations**. Why ? Because we are making greedy decisions
- Each data point is trying to push the parameters in a direction most favorable to it (without being aware of how the parameter update affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- There is no guarantee that each local greedy move will reduce the global error

- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)?
- Yes, let's look at mini-batch SGD

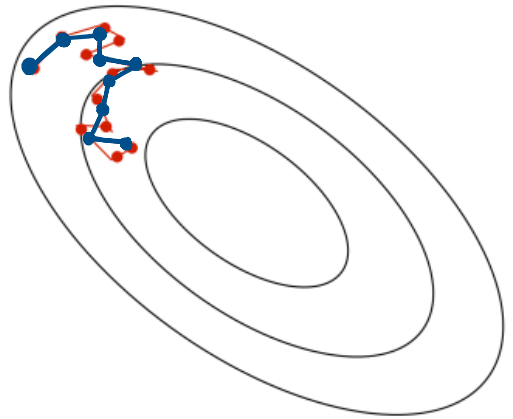


Mini-batch gradient descent

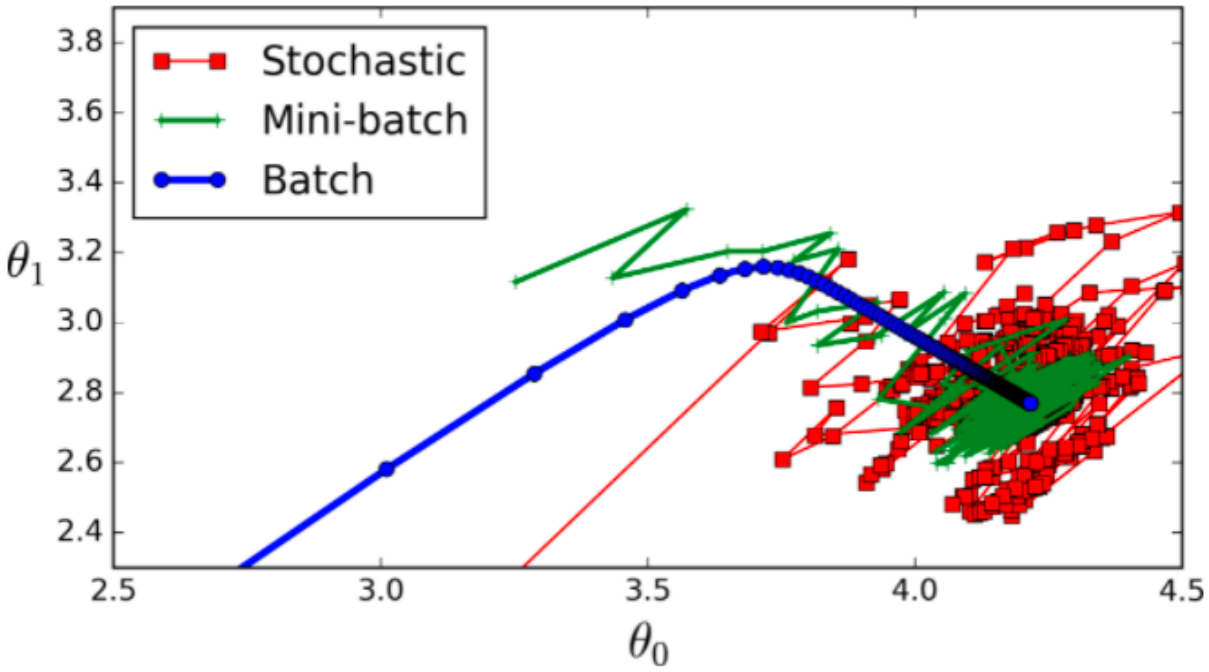
- Compute the gradients on a medium-sized set of training examples, called a ***mini-batch***
- Note that the algorithm updates the parameters after it sees a batch size B number of data points
- The stochastic estimates of gradients here are slightly better and less noisy
- Batch size $= 1$ leads to SGD! Typical batch sizes are 64, 128, 256

```
theta, eta, epochs  = -1, 0.001, 100
batch_size          = 64
num_points_seen     = 0
for i in range(epochs):
  dtheta = 0
  for x,t in zip(X,T):
    dtheta += grad_theta(theta, x, t)
    num_points_seen += 1

    if num_points_seen % batch_size == 0:
      # seen one mini-batch
      theta = theta - eta * dtheta / batch_size
      dtheta = 0 # reset gradients
```



Performance of mini-batch gradient descent



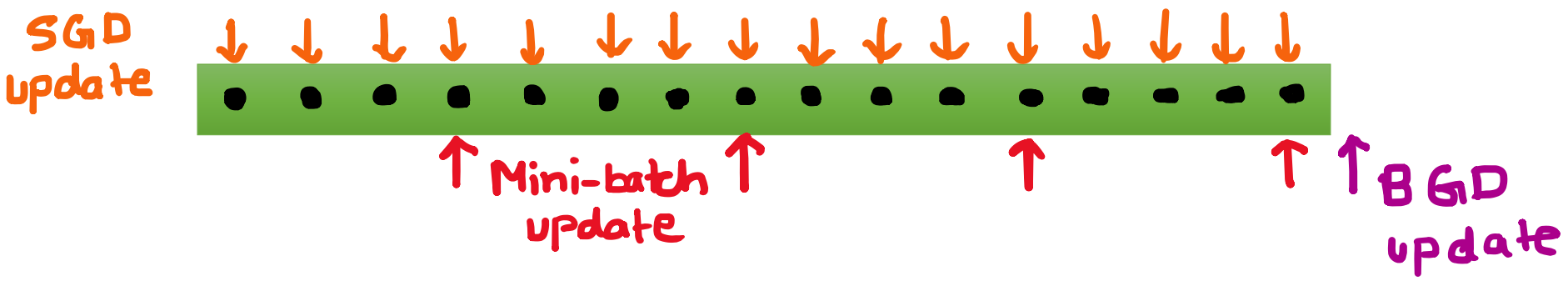
The mini-batch size *B* is a hyperparameter that needs to be set

- **Large batches:** converge in fewer parameter updates because each stochastic gradient is less noisy
- **Small batches:** perform more parameter updates because each one requires less computation

Things to remember

- N is the total number of training examples
- B is the mini batch size
- 1 epoch = one pass over the entire data
- 1 iteration = one update step of the parameters

Algorithm	Batch size	Number of iterations in 1 epoch
Batch GD	N	1
SGD	1	N
Mini-batch GD	B	$\sim N/B$

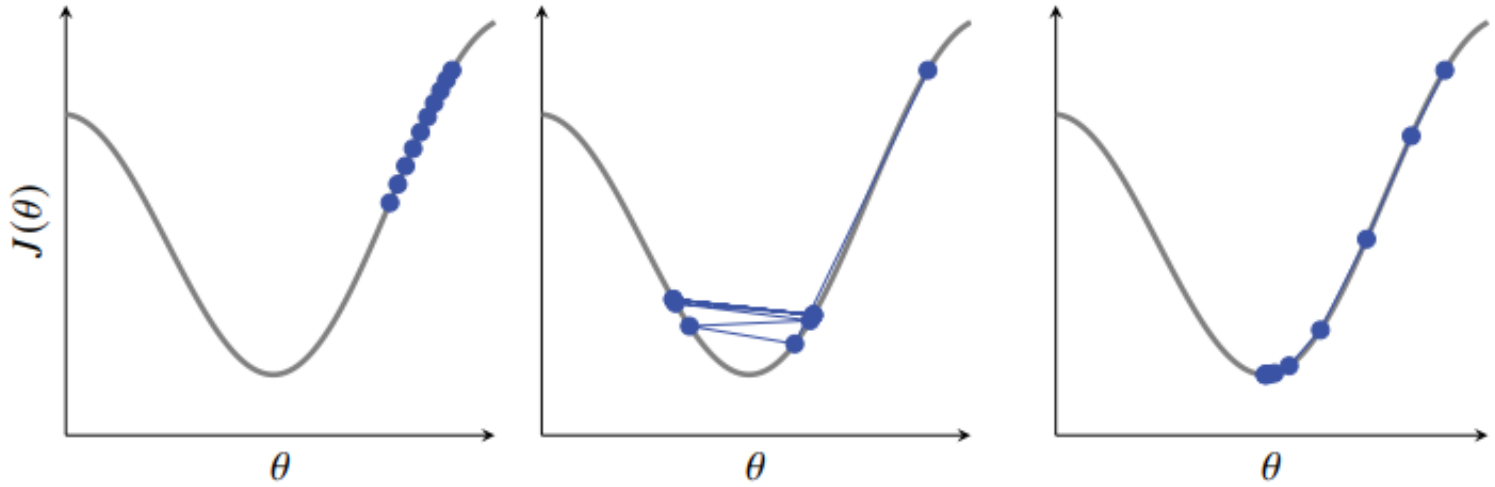


Learning rate

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} L^{(i)}$$

Update $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$
with some $\gamma > 0$

- Learning rate γ determines how big the $\boldsymbol{\theta}$-step to take at each iteration
- In practice we do not know what learning rate γ to choose



(a) Low learning rate $\gamma = 0.05$ **(b)** High learning rate $\gamma = 1.2$ **(c)** Good learning rate $\gamma = 0.3$

- γ is usually selected by the user, or it could be viewed as a hyperparameter

Different modifications to Gradient Descent

- Different modifications that can be applied to GD, SGD or mini-batch GD to improve convergence to a solution (possibly a local minimum)
- Two lines of improvements to traditional GD (or SGD or mini-batch GD)

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \, \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$$

learning rate

gradient of loss w.r.t. parameter $\theta^{(t)}$

Adaptively **modify the gradients** to accelerate learning	Adaptively **modify the learning rate** to prevent end oscillations
• Momentum-based gradients	• AdaGrad • RMSProp

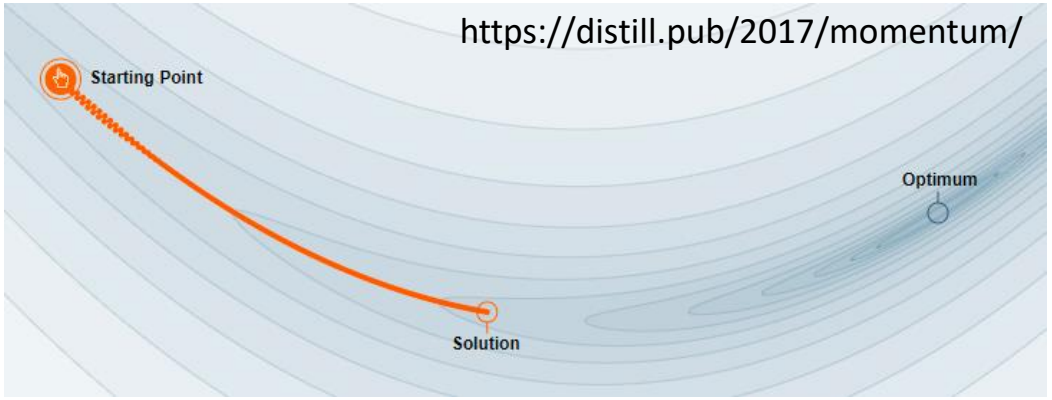
ADAM

- We will demonstrate these modifications on GD, but they are equally applicable to SGD and mini-batch GD as well

Momentum-based gradients

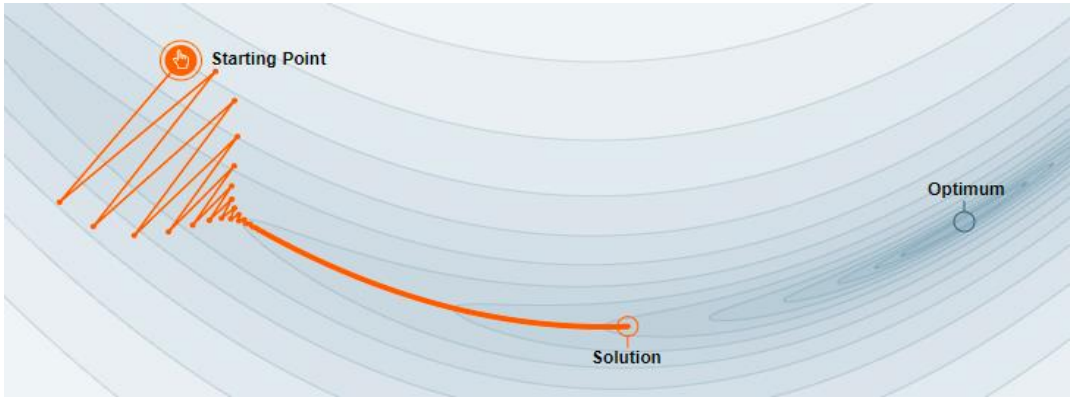
Intuition

- If you are repeatedly being asked to move in the same direction, then you should gain some confidence and start taking bigger steps in that direction



Slow learning along gentle slopes, many steps taken to converge

- If you move back-and-forth in different directions (i.e oscillations), then you should take smaller steps in the oscillatory directions



Oscillations across steep slopes

Gradient descent with momentum

- Can we accelerate learning by looking at the past behavior? Yes, use momentum
- If you are repeatedly being asked to move in the same direction, then you should gain some confidence and start taking bigger steps in that direction
- If you move back-and-forth in different directions (i.e., oscillations), then you should take smaller steps in the oscillatory directions
- Gain momentum by looking at the history of past gradients

history vector

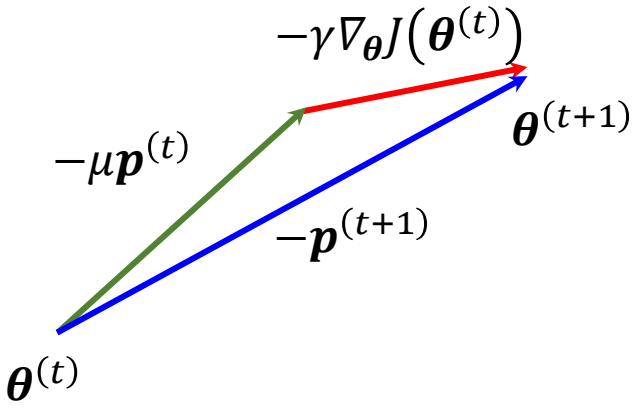
Update rule

- Compute momentum

$$\boldsymbol{p}^{(t+1)} \leftarrow \mu \boldsymbol{p}^{(t)} + \gamma \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$$

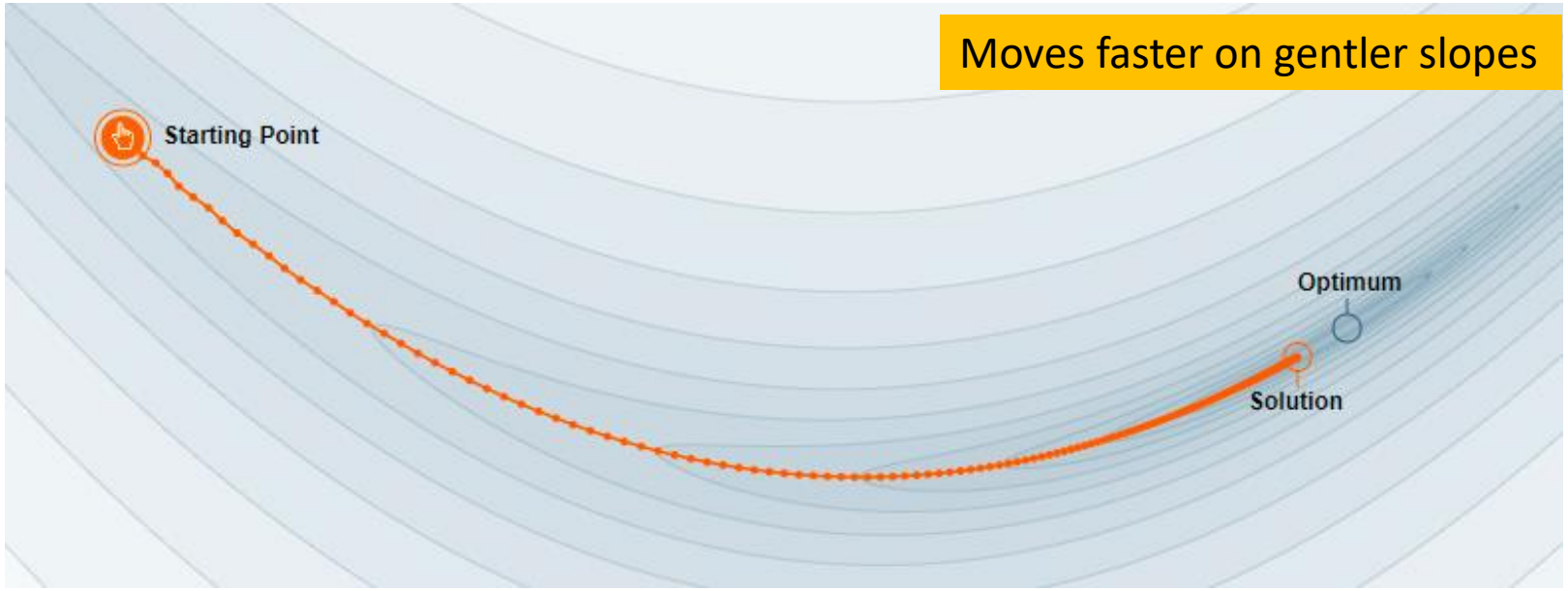
- Perform parameter update

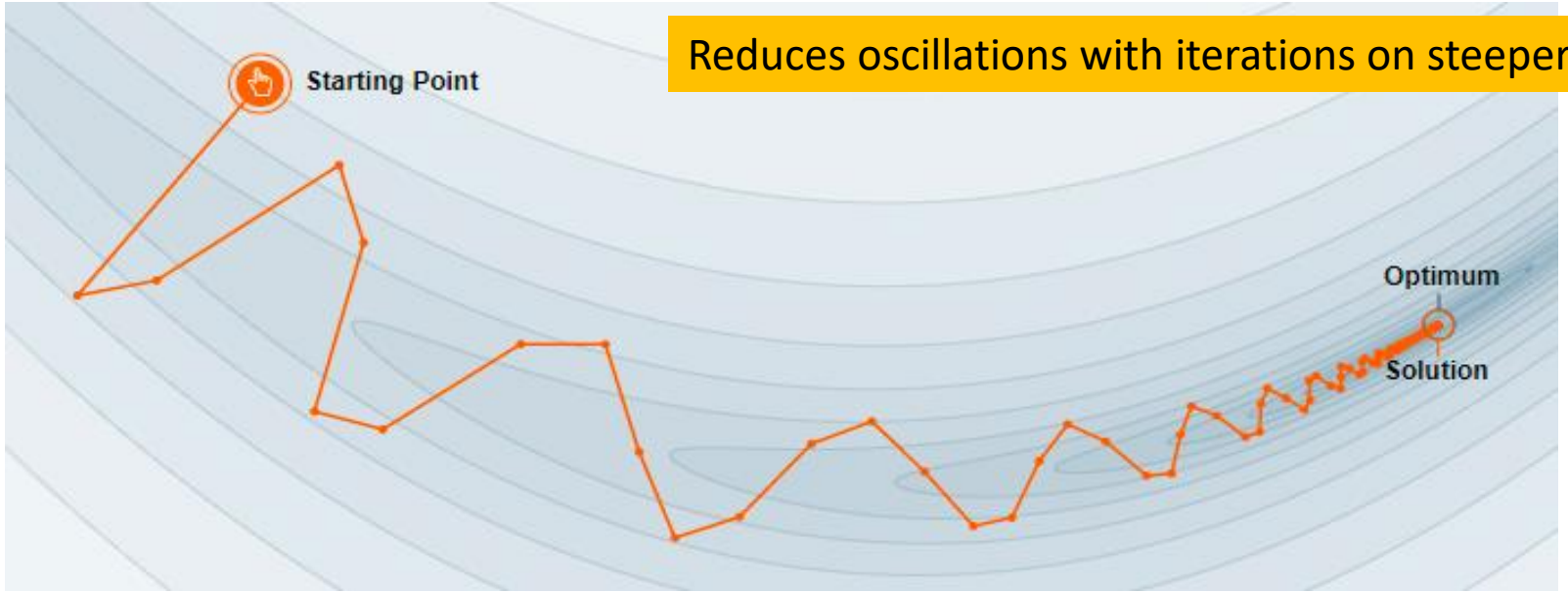
$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \boldsymbol{p}^{(t+1)}$$



- μ is a damping parameter, and should satisfying $0 \leq \mu \leq 1$
- μ should be slightly less than 1 (e.g. 0.9 or 0.99)

Gradient descent with momentum

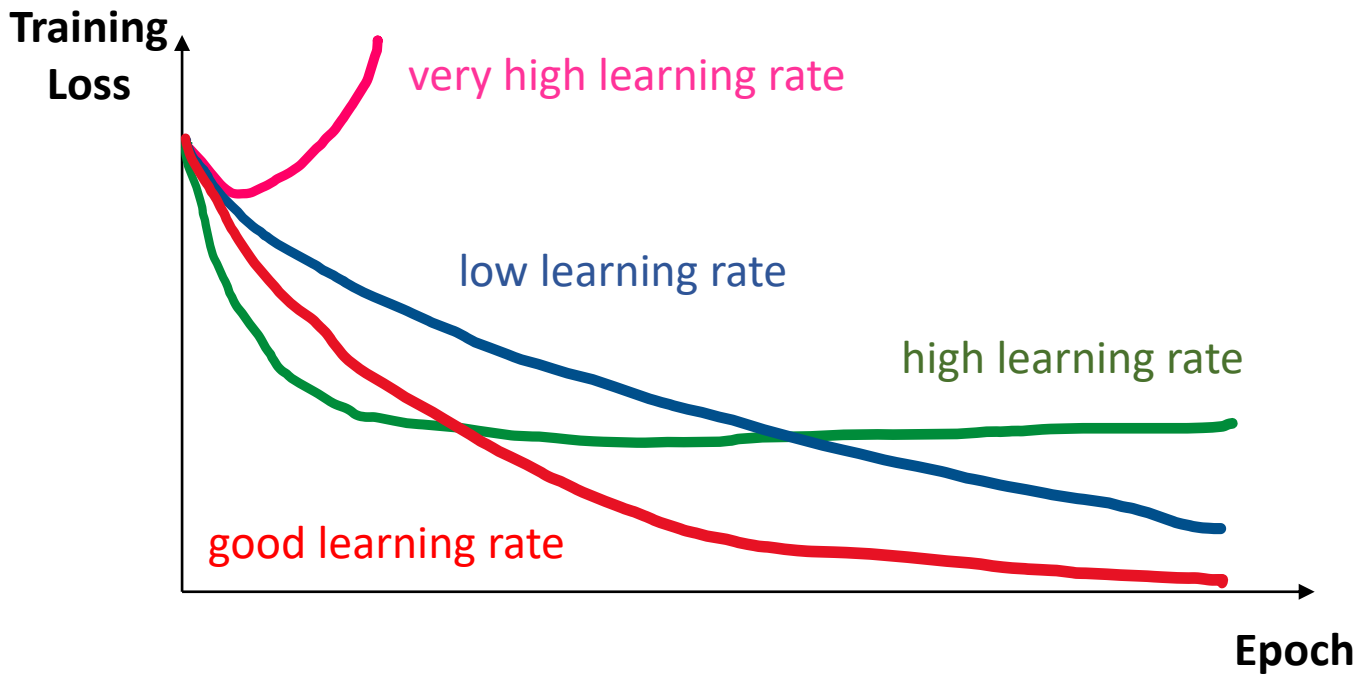




Modifying learning rate

- Ideal learning rate γ should be
  - Not too big (loss function may blow up, oscillations around minima)
  - Not too small (takes longer to converge)

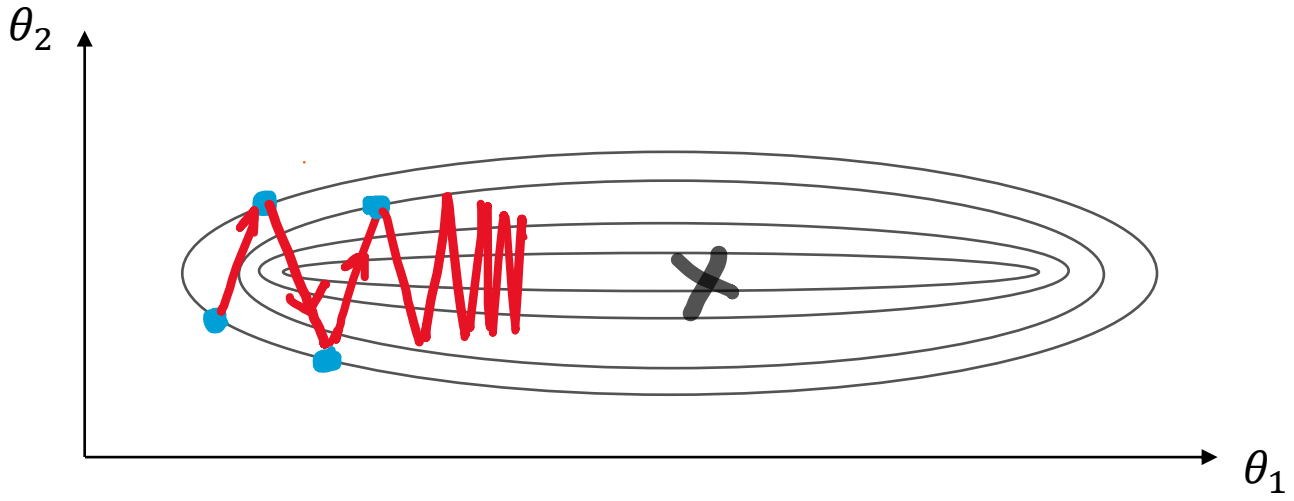
$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \gamma \, \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$$



Modifying learning rate

- One learning rate for all parameters is not good
- Can we tune the learning rate for each parameter directions separately?
  - E.g. We may want to move fast in one parameter direction compared to other

- Consider this toy problem with two parameters, we want to
  - Aggressively reduce learning rate in vertical direction
  - Gradually reduce learning rate in horizontal direction



Idea: Decay the learning rate for parameters in proportion to their **gradient magnitude history**

GD with Adaptive Gradients (AdaGrad)

- AdaGrad uses the magnitude of the gradient as a means of adjusting how quickly learning should occur
  - Parameters with large gradient magnitudes are provided with a smaller learning rate

Update rule for AdaGrad

- Get gradient

$$\boldsymbol{g}^{(t)} = \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$$

- Accumulate past gradient magnitudes in a history vector

history vector →

$$\boldsymbol{s}^{(t+1)} \leftarrow \boldsymbol{s}^{(t)} + \left(\boldsymbol{g}^{(t)}\right)^2$$

magnitude of gradient

elementwise squaring

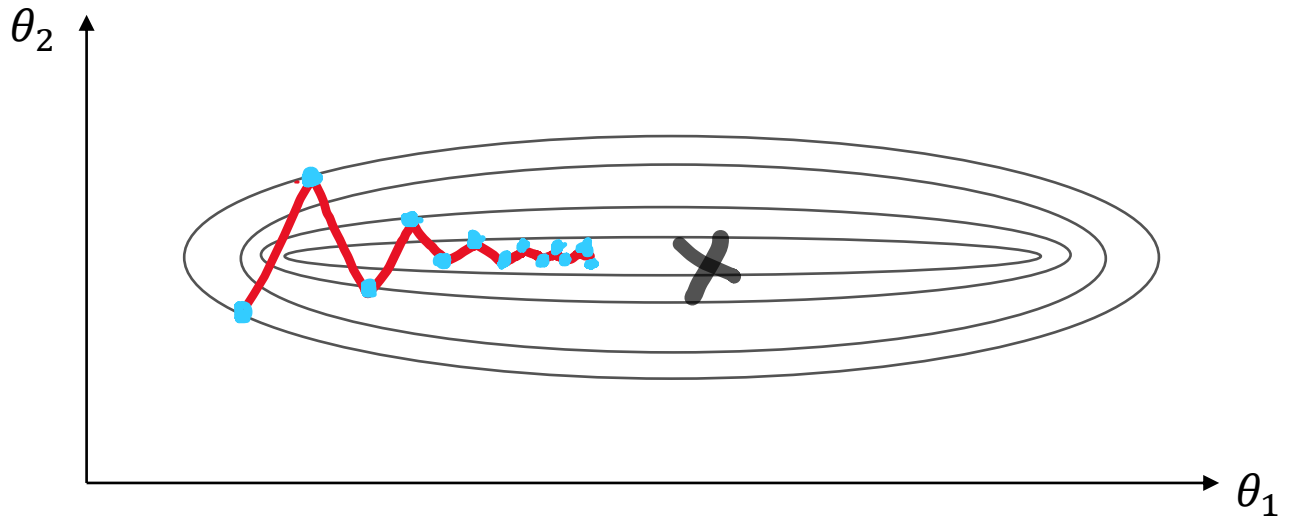
- Perform parameter update

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \frac{\gamma}{\sqrt{\boldsymbol{s}^{(t+1)} + \epsilon}} \boldsymbol{g}^{(t)}$$

- ϵ is a small additive constant (10^{-8}) that ensures that we do not divide by 0
- The squaring operation gets rid of signs (directions) of the gradients accumulated, hence we keep the magnitudes of gradients

- NOTE: The squaring and update operation is applied elementwise

Problems with AdaGrad

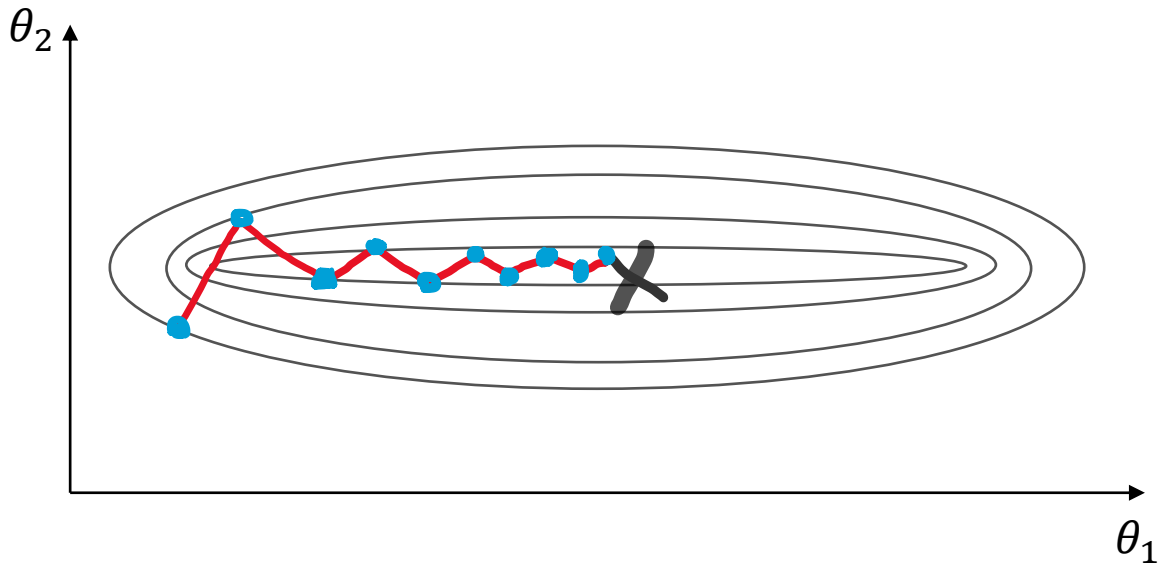


- However, Adagrad decays the learning rate very aggressively (since it accumulates all past gradient magnitudes and the denominator grows)
- As a result, during later epochs, some of the parameters will start receiving very small updates because of the decayed learning rate

- How can we prevent rapid growth of the denominator?
- Let's look at RMSProp

Root Mean Square Propagation (RMSProp)

Trick: Focus more on the recent past



Update rule for RMSProp

- Get gradient

$$\boldsymbol{g}^{(t)} = \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$$

- Accumulate moving average over the history vector

history vector →

$$\boldsymbol{s}^{(t+1)} \leftarrow \beta \boldsymbol{s}^{(t)} + (1-\beta)\left(\boldsymbol{g}^{(t)}\right)^2$$

- Perform parameter update

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \frac{\gamma}{\sqrt{\boldsymbol{s}^{(t+1)} + \epsilon}} \boldsymbol{g}^{(t)}$$

$0 \leq \beta \leq 1$

$\beta = 0.85, 0.9, 0.95$

$$\boldsymbol{s}^{(t+1)} = (1-\beta)\left[\left(\boldsymbol{g}^{(t)}\right)^2 + \beta\left(\boldsymbol{g}^{(t-1)}\right)^2 + \beta^2\left(\boldsymbol{g}^{(t-2)}\right)^2 + \cdots\right]$$

Adaptive Moment Estimation (ADAM)

Idea

- Do everything that RMSProp does to solve the decay problem of Adagrad
- Plus use momentum based on a cumulative history of the gradients
- ADAM = RMSProp + Momentum

- Get gradient

$$\boldsymbol{g}^{(t)} = \nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}^{(t)}\right)$$

- Compute momentum

$$\boldsymbol{p}^{(t+1)} \leftarrow \beta_1 \boldsymbol{p}^{(t)} + (1-\beta_1)\boldsymbol{g}^{(t)}$$

- Accumulate past gradient step sizes in a history vector

$$\boldsymbol{s}^{(t+1)} \leftarrow \beta_2 \boldsymbol{s}^{(t)} + (1-\beta_2)\left(\boldsymbol{g}^{(t)}\right)^2$$

- Perform parameter update

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \frac{\gamma}{\sqrt{\boldsymbol{s}^{(t+1)} + \epsilon}} \boldsymbol{p}^{(t+1)}$$

In practice

$\beta_1 = 0.9$

$\beta_2 = 0.999$