Optimization

Machine Learning (CSCI 5770G)

Faisal Z. Qureshi

http://vclab.science.ontariotechu.ca

Acknowledgements

- Lecture notes by Hinton and others found at CS UofT.

Lesson Plan

- Minimizing loss and the need for numerical techniques
- Gredient desent
  - Recipe
  - Update rule
- Batch update
- Mini-batch update
- Stochastic (or online) gradient descent
- Learning rate
  - Changing learning rate to achieve faster convergence
- Momentum and other variants of gradient descent
- Newton's method
  - How to choose a step size?
- Momentum

Gradient and Hessian

Consider a function

$$f(x_1, \cdots, x_d) \in \mathbb{R}$$

Gradient of f is (caputres slope of a function at a point)

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix} \in \mathbb{R}^d$$

Hessian of f is (caputres curvature of a function at a point)

$$\mathbf{H} = \nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_d \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_d^2} \end{bmatrix} \in \mathbb{R}^{d \times d}$$

Jacobian

Consider a vector-valued function

$$\mathbf{f}(x_1, x_2, x_3, \cdots, x_d) = (f_1, f_2, f_3, \cdots, f_n) \in \mathbb{R}^n$$

Jacobian of $\mathbf{f}$ is

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_d} \end{bmatrix} \in \mathbb{R}^{n \times d}$$

Example problem

Consider data points $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(N)}, y^{(N)})$. Our goal is to learn a function $f(x)$ that returns (predict) the value y given an x.

Choose a model

We assume that a linear model of the form $y = f(x) = \theta_0 + \theta_1 x$ best describes our data.

Problem

How do we determine the degree of "fit" of our model?

Example problem

Loss/cost/objective function measures the degree of fit of a model to a given data.

Least squares error

$$C(\theta_0, \theta_1) = \sum_{i=1}^{N} \left(y^{(i)} - f(x^{(i)}) \right)^2$$

Task (learning)

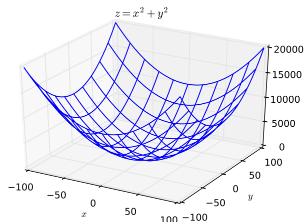
Our task is to find values for θ_0 and θ_1 (model parameters) to minimize the cost.

Example problem

Minimizing cost

$$(\theta_0, \theta_1) = \underset{(\theta_0,\theta_1)}{\arg\min}\, C(\theta)$$

This is a convex function. We can solve for θ_0 and θ_1 by setting $\frac{\partial C}{\partial \theta_0} = 0$ and $\frac{\partial C}{\partial \theta_1} = 0$.



Minimizing cost and the need for numerical techniques

In general cost functions are not convex and it is not possible to find a minima (there are absolutely no guarantees about finding the global minima) using analytical methods

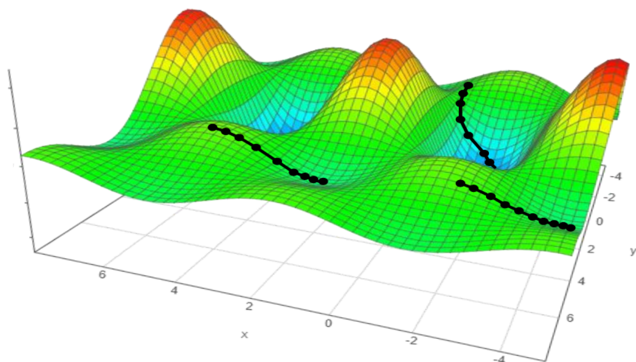
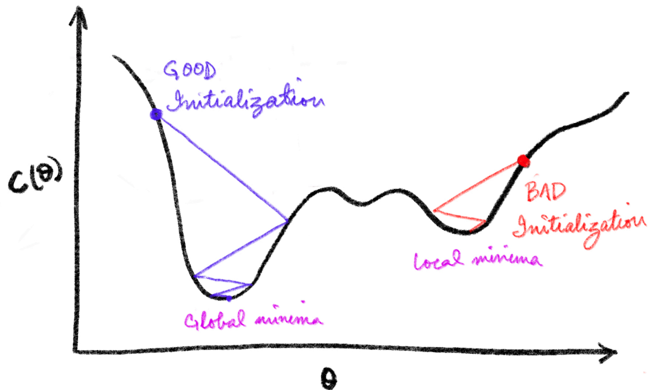


Figure from ATTRIBUTION MISSING

Gradient descent

A very powerful method of training the model parameters by minimizing the loss function. One of the simplest optimization methods. It is also referred to as *steepest descent*.



Gradient descent recipe

1. Initialize model parameters randomly (in our case θ)
2. Compute gradient of the loss function
3. Take a step in the direction of negative gradient (decreasing loss function) and update parameters
4. Repeat steps 2 to 4 until cannot decrease loss function anymore

Update rule

If $C(\theta)$ is the cost that we wish to minimize then the gradient descent update rule is

$$\begin{aligned} \theta^{(k+1)} &= \theta^{(k)} - \eta \frac{\partial C}{\partial \theta}\bigg|_k \\ &= \theta^{(k)} - \eta \mathbf{g}_k \end{aligned}$$

where η is referred to as the learning rate, which controls the size of the step taken at each iteration.

Notation alert: We set $\mathbf{g}_k = \nabla_\theta C|_k$ to simplify the notation.

Batch update

- Sum or average updates across every example, then change the parameter values

$$\theta^{(k+1)} = \theta^{(k)} - \eta \sum_{i=1}^{N} \mathbf{g}_k^{(i)}$$

where N is the number of examples (or data items).

Mini-batch update

- Sum or average updates across a subset of the examples, then change the parameter values
- Examples in each batch are selected at random

$$\theta^{(k+1)} = \theta^{(k)} - \eta \sum_{i=1}^{N_{\text{batch}}} \mathbf{g}_k^{(i)}$$

  where N_{batch} is the number of examples (or data items) in a mini-batch.
- This is particularly useful when dealing with very large datasets
  - Ability to exploit computational efficiencies
- Mini-batches need to be balanced for classes

Stochastic or online gradient desent

- Update parameter values for each training example in turn
- This assumes that sample is i.i.d. (independent, identically distributed)

$$\theta^{(k+1)} = \theta^{(k)} - \eta \mathbf{g}_k^{(i)}$$

  where i is the i-th example (or data item).
- Assumes that the dataset is highly redundant

The effects of using a subset of data to compute loss

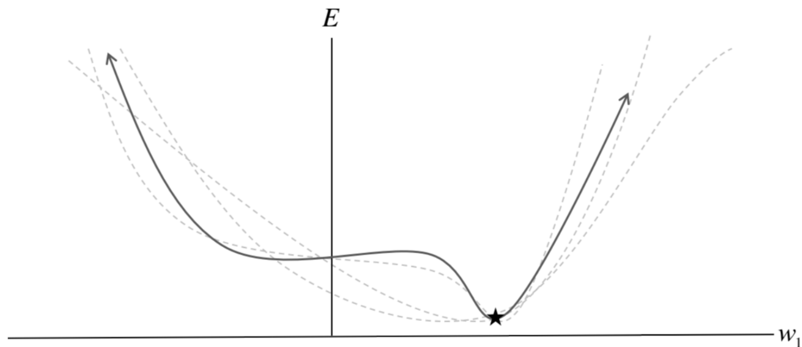
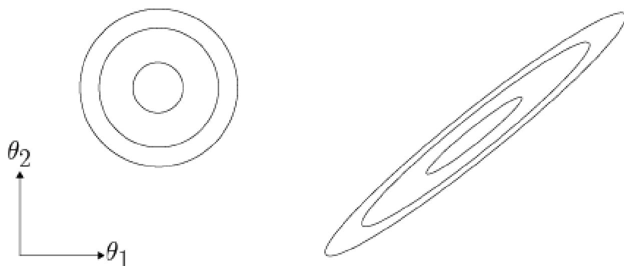


Figure from ATTRIBUTION MISSING

The error surface of a linear neuron

- Error surface is a quadratic bowl for linear neuron with squared error
  - Quadratic bowl view is a good approximation for error surface when dealing with multilayer networks
- Consider a quadratic bowl, does the gradient at a location points to its minima?
  - Only when the quadratic bowl is a circle



Convergence on elliptic quadratic bowls

- The gradient is big in one direction and small in the other direction
  - The minima lies such that we need to travel little along the direction of the big gradient and travel more along the direction of the small gradient.
- This results in a to and fro motion
  - The oscillations diverge if the learning rate is too large
- What is needed?
  - Take large steps along the direction of small, but consistent, gradients
  - Take small steps along the direction of large, but inconsistent, gradients

Learning rate

- η too large, and the system might not converge
- η too small, and the system might take too long to converge

Too small or too large a learning rate

learning late is too small

learning rate is too large

Too small or too large a learning rate

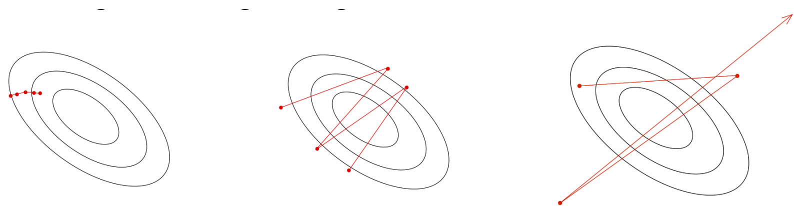


Figure from ATTRIBUTION MISSING

Changing learning rate mid-training

- Reduce learning rate during training
  - Raduces random fluctuations due to differences in gradient computation over mini-batches
  - Need to be done with care
- Avoids zig-zag behavior

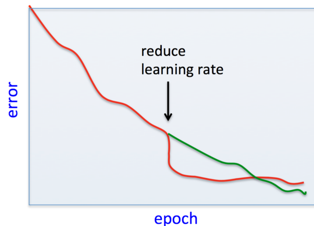


Figure from ATTRIBUTION MISSING

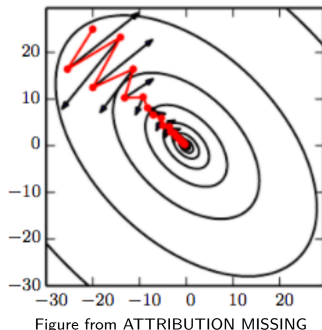


Figure from ATTRIBUTION MISSING

Gradient descent

$$\theta^{(k+1)} = \theta^{(k)} - \eta \mathbf{g}_k$$

- For gradient descent, $C(\theta)$, i.e., the objective, should decrease after every step.
- A common scheme is stop iterations (i.e., declare convergence) if decrease in $C(\theta)$ is less than some threshold ϵ (say 10^{-3}) in one iteration.
- General rule of thumb: If gradient descent isn't working, use a smaller η.
- Towards the end of minibatch training, reduce learning rate
  - Prevents oscillations in the final parameter values between different minibatches

Parameter initialization

- Small random values to break symmetry
- When two neurons have exactly same inputs and outputs (i.e., same bias and weights) they get the same gradient and they can cannot learn different features
- Often helps to initialize the incoming weights to be proportional to the square root of fan-in

Shifting the input values

- It is useful to shift to input values such that each component of the input vector has a zero mean throughout
  - Particularly so when using steepest descent algorithms for optimization
- Decorrelate the input dimensions
  - Use PCA
    - Drop dimensions with smallest eigenvalues to achieve some compression
    - Divide the remaining principal components by the square roots of their eigenvalues. For a linear neuron this makes the error surface cirvular.

Common pitfalls

- Large learning rates can make weights very large (positive or negative) and the hidden units may saturate leading to very small derivatives.
  - This looks like a local minima, but it is actually a plateau.
- For classification networks that use crossentropy losses, the best guessing strategy is to make each output unit always spit out a 1 equal to the proportion of time it shoudl be 1.
  - Network can quickly get stuck in this. This too is a plateau that behaves like a local minima.

Solutions

- Use momentum
- Use adaptive learning rates for each parameter
- Check out the optimization literature to look for more sophisticated methods for minimizing functions, say methods that make use of curvature information.

Exponential Moving Average (EMA)

Given a series of data:

$$\cdots, x^{(3)}, x^{(2)}, x^{(1)}$$

Compute EMA as

$$\langle x \rangle^{(i)} = \alpha x^{(i)} + (1 - \alpha) \langle x \rangle^{(i-1)}$$

where $\alpha \in [0, 1]$ and $\langle x \rangle^{(i)}$ is the running average at time i.

Gradient descent

$$\theta^{(k+1)} = \theta^{(k)} - \eta \mathbf{g}_k$$

- Learning rate $\eta = 0.0001$

Momentum

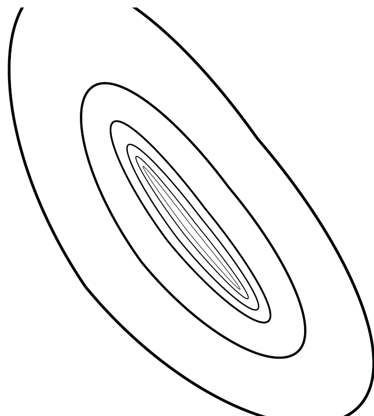
- Akin to giving a short-term memory to gradient descent

$$\mathbf{m}^{(k)} = \beta \mathbf{m}^{(k-1)} + (1-\beta)\mathbf{g}_k$$
$$\theta^{(k+1)} = \theta^{(k)} - \eta \mathbf{m}^{(k)}$$

- Momentum $\mathbf{m}^{(k)}$ is the first moment (the mean) of gradients at time k
- $\beta = 0.9$
- Momentum helps with tight ravines in the loss landscap
- Learning rate $\eta = 0.001$

Polyak, Boris. (1964). Some methods of speeding up the convergence of iteration methods. Ussr Computational Mathematics and Mathematical Physics. 4. 1-17. 10.1016/0041-5553(64)90137-5.

Momentum



https://distill.pub/2017/momentum/

Nesterov accelerated gradient

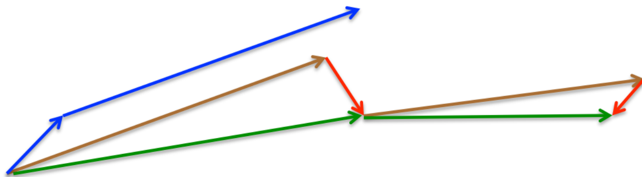
- Consider the momentum update rule

$$\begin{aligned}\theta^{(k+1)} &= \theta^{(k)} - \eta \mathbf{m}^{(k)} \\ &= \theta^{(k)} - \eta \left(\beta \mathbf{m}^{(k-1)} + (1-\beta)\mathbf{g}_k \right)\end{aligned}$$

- We know that we will update the parameters using momentum term $\mathbf{m}^{(k-1)}$ and the gradient computed at the current parameter value
- Instead let's update the parameters using the momentum term and compute gradient at the updated parameter value

Nesterov Accelerated Gradient (NAG)

$$\mathbf{m}^k = \beta \mathbf{m}^{k-1} + \eta \mathbf{g}_{(k+\beta \mathbf{m}^{(k-1)})}$$
$$\theta^{(k+1)} = \theta^{(k)} - \mathbf{m}^k$$



Source G. Hinton lecture 6

Adaptive Gradient (AdaGrad)

- Updates the learning rate by dividing it by the square root of cumulative sum of current and past squared gradients

$$G^{(k)} = G^{(k-1)} + \mathrm{diag}(\mathbf{g}_k^2)$$
$$\theta^{(k+1)} = \theta^{(k)} - \left(\frac{\eta}{\sqrt{G^{(k)} + \epsilon}}\right)\mathbf{g}_k$$

- $\mathbf{g}_k^2$ denotes element-wise square of gradient vector at step k
- $G^{(k)}$ is a diagonal matrix.
- $G_{ii}^{(k)}$ contains sum of squares of derivative w.r.t. parameter θ_i.
- $\epsilon = 10^{-7}$ prevents a divison by zero.

John Duchi, Elad Hazan, Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12(61):2121-2159, 2011.

AdaGrad

- One doesn't need to tune the learning rate manually
- Often $\eta = 0.01$ is used
- The learning rate shrinks over time, eventually becoming infinitesimally small at which point the model can no longer learn

AdaGrad notation simplification

- With some abuse of notation, we re-write AdaGrad as follows

$$v^{(k)} = v^{(k-1)} + \mathbf{g}_k^2$$
$$\theta^{(k+1)} = \theta^{(k)} - \left(\frac{\eta}{\sqrt{v^{(k)} + \epsilon}}\right) \mathbf{g}_k$$

- v takes place of G and we assume that the shape of v can be infered from the surrounding context
- We'll stick to this notation for the next little while

Adadelta

- AdaGrad monotonically decreases the learning rate, Adadelta aims to fix this behavior
- Adadelta restricts sum of squared gradients to the previous w steps (in practice it is easier to just maintain the running average)

$$
\begin{aligned}
v^{(k)} &= \beta_1 v^{(k-1)} + (1-\beta_1)\mathbf{g}_k^2 \\
\Delta_k &= \frac{\sqrt{u^{(k-1)}+\epsilon}}{\sqrt{v^{(k)}+\epsilon}}\mathbf{g}_k \\
\theta^{(k+1)} &= \theta^{(k)} - \Delta_k \\
u^{(k)} &= \beta_2 u^{(k-1)} + (1-\beta_2)\Delta_k^2
\end{aligned}
$$

- $v^{(k)}$ is the second moment (uncentered variance) of gradients up to step k
- $u^{(k-1)}$ is the second moment (uncentered variance) of the updates up to step k

Adadelta

- $\epsilon \approx 10^{-7}$ prevents a divison by zero.
- The learning rate also has been eliminated.
- However, some implementations still use a learning rate (see PyTorch)

RMSProp

$$v^{(k)} = \beta v^{(k-1)} + (1-\beta)\mathbf{g}_k^2$$
$$\theta^{(k+1)} = \theta^{(k)} - \left(\frac{\eta}{\sqrt{v^{(k)} + \epsilon}}\right)\mathbf{g}_k$$

- $\beta = 0.9$
- $\epsilon \approx 10^{-7}$ prevents a divison by zero.
- Learning rate $\eta = 0.001$

Adam

$$\mathbf{m}^{(k)} = \beta_1 \mathbf{m}^{(k-1)} + (1-\beta_1)\mathbf{g}_k$$
$$\hat{\mathbf{m}}^{(k)} = \frac{\mathbf{m}_t}{(1-\beta_1)}$$
$$v^{(k)} = \beta_2 v^{(k-1)} + (1-\beta_2)\mathbf{g}_k^2$$
$$\hat{v}^{(k)} = \frac{v_t}{(1-\beta_2)}$$
$$\theta^{(k+1)} = \theta^{(k)} - \left(\frac{\eta}{\sqrt{\hat{v}^{(k)} + \epsilon}}\right)\hat{\mathbf{m}}^{(k)}$$

- $\mathbf{m}_t$ and $\mathbf{v}_t$ are estimates of first moment and second moment of gradients, respectively

Adam

- Compute bias-corrected first and second moments, $\hat{\mathbf{m}}$ and $\hat{v}$, since $\mathbf{m}$ and v are initialized to 0, so they are biased towards 0, especially when 1) β_1 and β_2 are small and 2) during initial timesteps
- $\epsilon \approx 10^{-8}$ prevents a divison by zero.
- $\beta_1 = 0.9$ and $\beta_2 = 0.999$

Vector norms

- The vector norm $|\mathbf{x}|_p$ for $p = 1, 2, 3, \cdots$

$$|\mathbf{x}|_p = \left(\sum_i |\mathbf{x}_i|_p \right)^{\frac{1}{p}}$$

- $|\mathbf{x}|_\infty$ is a special case

$$|\mathbf{x}|_\infty = \max_i \mathbf{x}_i$$

AdaMax

- In Adam $v^{(k)}$ is updated via l_2 norm, we can change it to use l_p norm instead

$$v^{(k)} = \beta_2^p v^{(k-1)} + (1 - \beta_2^p)|\mathbf{g}_k|^p$$

- Using large values of p leads to numerical unstability
- l_∞ exhibits stable behavior, however

$$\begin{aligned} v^{(k)} &= \beta_2^\infty v^{(k-1)} + (1 - \beta_2^\infty)|\mathbf{g}_k|^\infty \\ &= \max\left(\beta_2^\infty v^{(k-1)}, |\mathbf{g}_k|\right) \end{aligned}$$

AdaMax

- No need to compute $\hat{\mathbf{v}}$, since $\max$ operation makes $\mathbf{v}$ less susceptible to bias towards 0

$$
\begin{aligned}
\mathbf{m}^{(k)} &= \beta_1 \mathbf{m}^{(k-1)} + (1-\beta_1)\mathbf{g}_k \\
\hat{\mathbf{m}}_t &= \frac{\mathbf{m}^{(k)}}{(1-\beta_1)} \\
v^{(k)} &= \max\left(\beta_2^{\infty} v^{(k-1)}, |\mathbf{g}_k|\right) \\
\theta^{(k+1)} &= \theta^{(k)} - \left(\frac{\eta}{\sqrt{v^{(k)}} + \epsilon}\right)\hat{\mathbf{m}}^{(k)}
\end{aligned}
$$

- $\beta_1 = 0.9$
- $\beta_2^{\infty} = 0.99$
- $\eta = 0.001$

Other approaches

Nesterov-accelerated Adaptive Momentum Estimation (NADAM)

- Combines Adam and NAG

AMSGrad

- Uses the maximum of past squared gradients rather then the uncentered variance of these gradients (i.e., average of the past squared gradients)

And many others

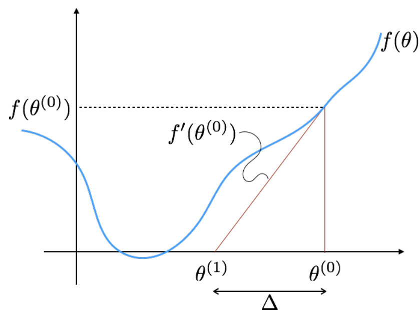
- See here for some recent gradient descent based algorithms (*last accessed on Feb 4, 2023*)
- Check out this for playing around with different algorithms (*last accessed on Feb 4, 2023*)

Newton's method

Problem: Given $f(\theta)$, find θ such that $f(\theta) = 0$

Observation: Gradient is the slope of the function.

$$f'(\theta^{(0)}) = \frac{f(\theta^{(0)})}{\Delta} = \frac{f(\theta^{(0)})}{\theta^{(1)} - \theta^{0}}$$



Newton's method

An iterative method to solve the above problem is:

1. Start with an initial value of θ
2. Update θ as follows

$$\theta^{(1)} = \theta^{(0)} - \frac{f(\theta^{(0)})}{f'(\theta^{(0)})}$$

3. Repeat until $f(\theta) \approx 0$ or maximum number of iterations have reached.

Using Newton's method to minimize a function

- We are interested in maximizing or minimize a function, e.g., we minimize the negative log likelihood to estimate the parameters.
- Gradient is 0 at maxima, minima and saddle points of a function.
- We can use the following update rule to maximize or minimize a function.

$$\theta^{(1)} = \theta^{(0)} - \frac{C'(\theta^{(0)})}{C''(\theta^{(0)})}$$

- This update rule is a direct application of the Newton's method and finds $C'(\theta) = 0$.

Newton's method to minimize a function in higher dimensions

Using Hessian we can write down the update rule as follows

$$\theta^{(k+1)} = \theta^{(k)} - \mathbf{H}_k^{-1}\mathbf{g}_k$$

where $\mathbf{H}_k = \nabla^2 C\big|_k$ denotes Hessian at step k and $\mathbf{g}_k$ denotes gradient at step k.

Summary

- Gradient descent
  - Batch update
  - Online or stochastic
- Learning rate
- Momentum
- Hessian
- These methods work well in practice
- All of these methods converge as long as the learning rate is sufficiently small
- The speed of convergence differs great, however
- Newton's method converges quadratically
  - Every iteration of Newton's method doubles the number of digits to which your solution is accurate, e.g., error goes from 0.01 to 0.0001 in one step
  - This only holds when already close to the solution
- θ is initialized randomly

Copyright and License

©Faisal Z. Qureshi

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.