Numerical Optimization for Data Science and Machine Learning (NOFDS&ML)

Information Technology Institute

Session 4 Contents

Motivation

Momentum-based, and Nesterov accelerated gradient descent solved the problem of going too slow with gentle slope loss function.

Learning rate stays untouched.

Review and Summary of Previously Discussed Topics

- **Common Notations**
- **Hadamard Product**
- **Review of Previously Discussed Optimization Methods**
- **Motivation for other Methods**

Common Notation

- Parameter vector: $\boldsymbol{\theta} = [\theta_1, \theta_2, \dots \theta_n]^T,$

(note that in neural networks, the parameters can be denoted w_j and b_j (weights and biases)).

- Feature index (subscript j)
  - $j = 1, \dots, n$: original input features (*n* features)
  - In linear regression, we introduce a **bias (zero-feature)**:

$x_0 = 1 \;\Rightarrow\; j = 0,1, \dots, n \Rightarrow \boldsymbol{\theta} = [\theta_0, \theta_1, \dots \theta_n]^T,$

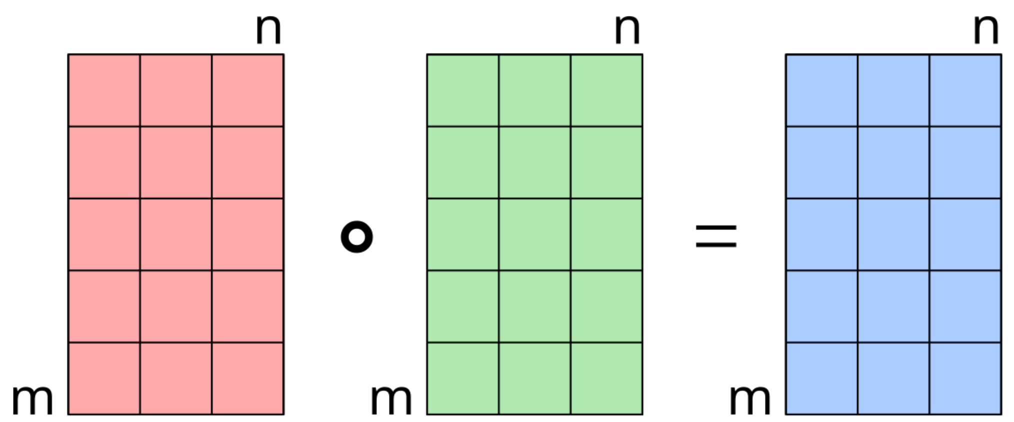
- Example index (superscript i)
  - $i = 1, \dots, m$, where m is the dataset size (m examples)
  - $x_j^{(i)}$: feature j of example i

Common Notation

- Objective (cost) function (total loss): $J(\boldsymbol{\theta})$
- Gradient (full dataset): $\nabla J(\boldsymbol{\theta})$
- Mini-batch size: $\mathcal{B}$
- Learning rate: α (or η)
- Hadamard (element-wise) product: $\odot$

Hadamard Product (for Vectors and Matrices)

- Hadamard product (also known as the element-wise product) is an operation that takes in two matrices of the same dimensions and returns a matrix of the multiplied corresponding elements.



$$(A \odot B)_{ij} = (A)_{ij}(B)_{ij}.$$

$$\mathbf{A} = \underbrace{\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,N} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{bmatrix}}_{MxN} \qquad \mathbf{B} = \underbrace{\begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,N} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ b_{M,1} & b_{M,2} & \cdots & b_{M,N} \end{bmatrix}}_{MxN}$$

$$\mathbf{A} \circ \mathbf{B} = \underbrace{\begin{bmatrix} a_{1,1}b_{1,1} & a_{1,2}b_{1,2} & \cdots & a_{1,N}b_{1,N} \\ a_{2,1}b_{2,1} & a_{2,2}b_{2,2} & \cdots & a_{2,N}b_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1}b_{M,1} & a_{M,2}b_{M,2} & \cdots & a_{M,N}b_{M,N} \end{bmatrix}}_{MxN}$$

Gradient Descent Variants

1. Vanilla / Batch GD

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla J(\boldsymbol{\theta}_t)$$

2. SGD

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla J_i(\boldsymbol{\theta}_t)$$

3. Mini-batch GD

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla J_i(\boldsymbol{\theta}_t)$$

Gradient Descent Variants

1. Vanilla / Batch GD (uses the full dataset)

- **Single parameter update:**

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \alpha \frac{\partial J\left(\boldsymbol{\theta}^{(t)}\right)}{\partial \theta_j}$$

- **Vector form**

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla J(\boldsymbol{\theta}_t)}$$

Gradient Descent Variants

3. Mini-Batch Gradient Descent (uses a batch $\mathcal{B}$)

- **Single parameter update:**

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \alpha \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial J_i\left(\boldsymbol{\theta}^{(t)}\right)}{\partial \theta_j}$$

- **Vector form:**

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla J_i(\boldsymbol{\theta}_t)}$$

Momentum

- **Momentum GD**

$$\boxed{\begin{aligned}\mathbf{v}_{t+1} &= \beta \mathbf{v}_t - \alpha \nabla J(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{v}_{t+1}\end{aligned}}$$

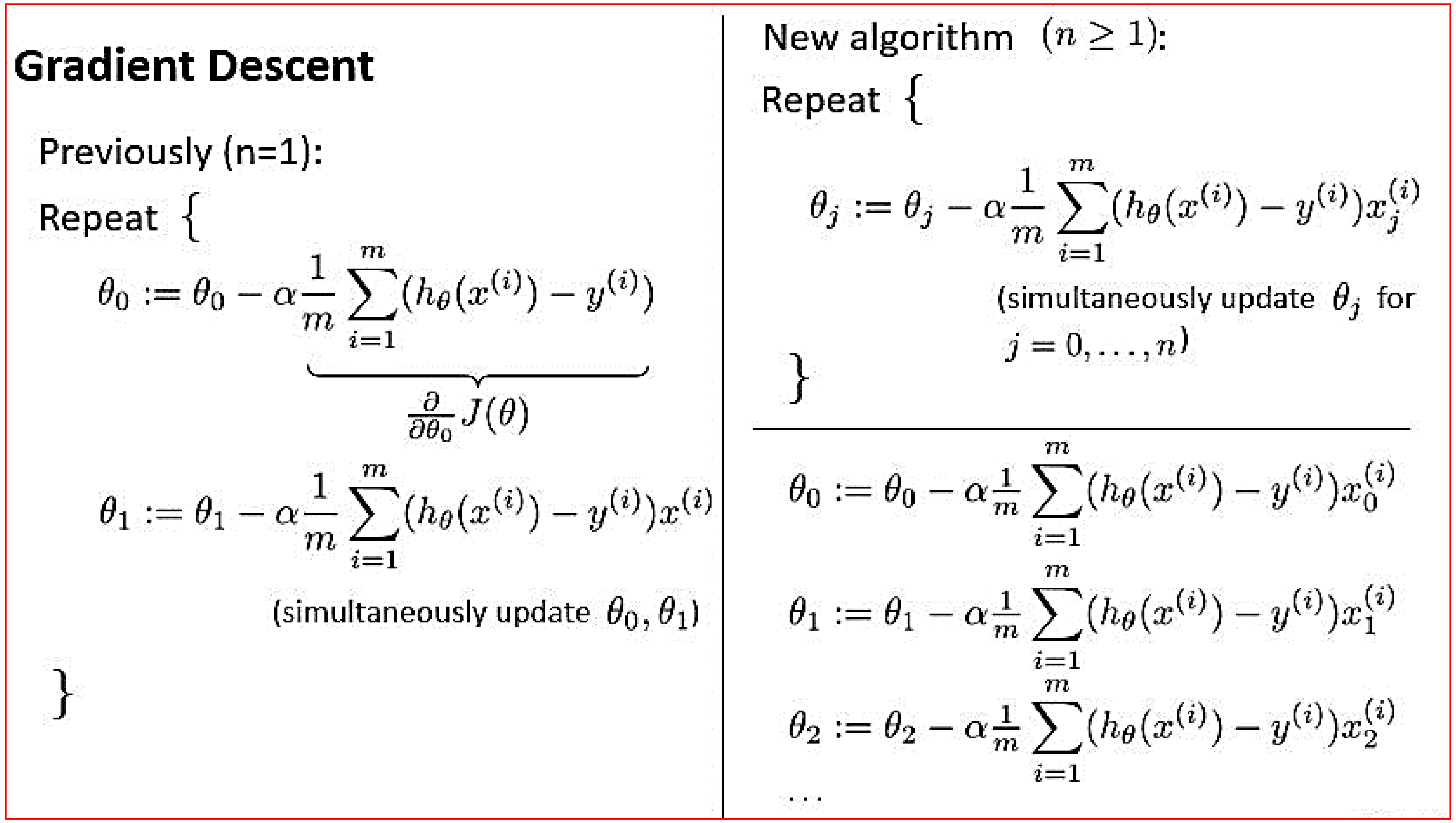
- **Nesterov Accelerated Gradient (NAG)**

$$\boxed{\begin{aligned}\mathbf{v}_{t+1} &= \beta \mathbf{v}_t - \alpha \nabla J(\boldsymbol{\theta}_t + \beta \mathbf{v}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{v}_{t+1}\end{aligned}}$$

Motivation for changing the learning rate

$$\boldsymbol{\Theta = \Theta - \alpha \nabla_{\Theta} J(\Theta)}$$

- The update is directly proportional to the gradient. The smaller the gradient the smaller the update.
- The gradient is directly proportional to the input. **Therefore, the update is dependent on the input also.**



Gradient Descent

Previously (n=1):

Repeat {

$$\theta_0 := \theta_0 - \alpha \underbrace{\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})}_{\frac{\partial}{\partial\theta_0}J(\theta)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$$

(simultaneously update θ_0, θ_1)

}

New algorithm $(n \geq 1)$:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

(simultaneously update θ_j for $j = 0, \ldots, n$)

}

$$\theta_0 := \theta_0 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}$$

$$\ldots$$

1. Learning Rate Scheduling

- **Learning-rate scheduling** simply means that the learning rate (denoted η or α) is **not constant**, but changes with iteration:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha_t \nabla J(\boldsymbol{\theta}_t)$$

- **Intuition:**
- Early in training, we want **large steps** to move quickly toward a good region.
- Later, we want **small steps** to avoid oscillation and fine-tune around the minimum.

Learning Rate Scheduling

- The learning rate is made **iteration-dependent**: $\alpha_t \downarrow$ as $t \uparrow$
- **Why this helps:**
  - Improves stability near the optimum
  - Reduces noise effects (especially in SGD)
  - Often required for theoretical convergence guarantees
- **Typical schedules:**
  - Step decay, Inverse time, Piecewise constant (drops at fixed epochs)
  - In short, learning-rate scheduling controls **how aggressively** GD moves at different stages of optimization.

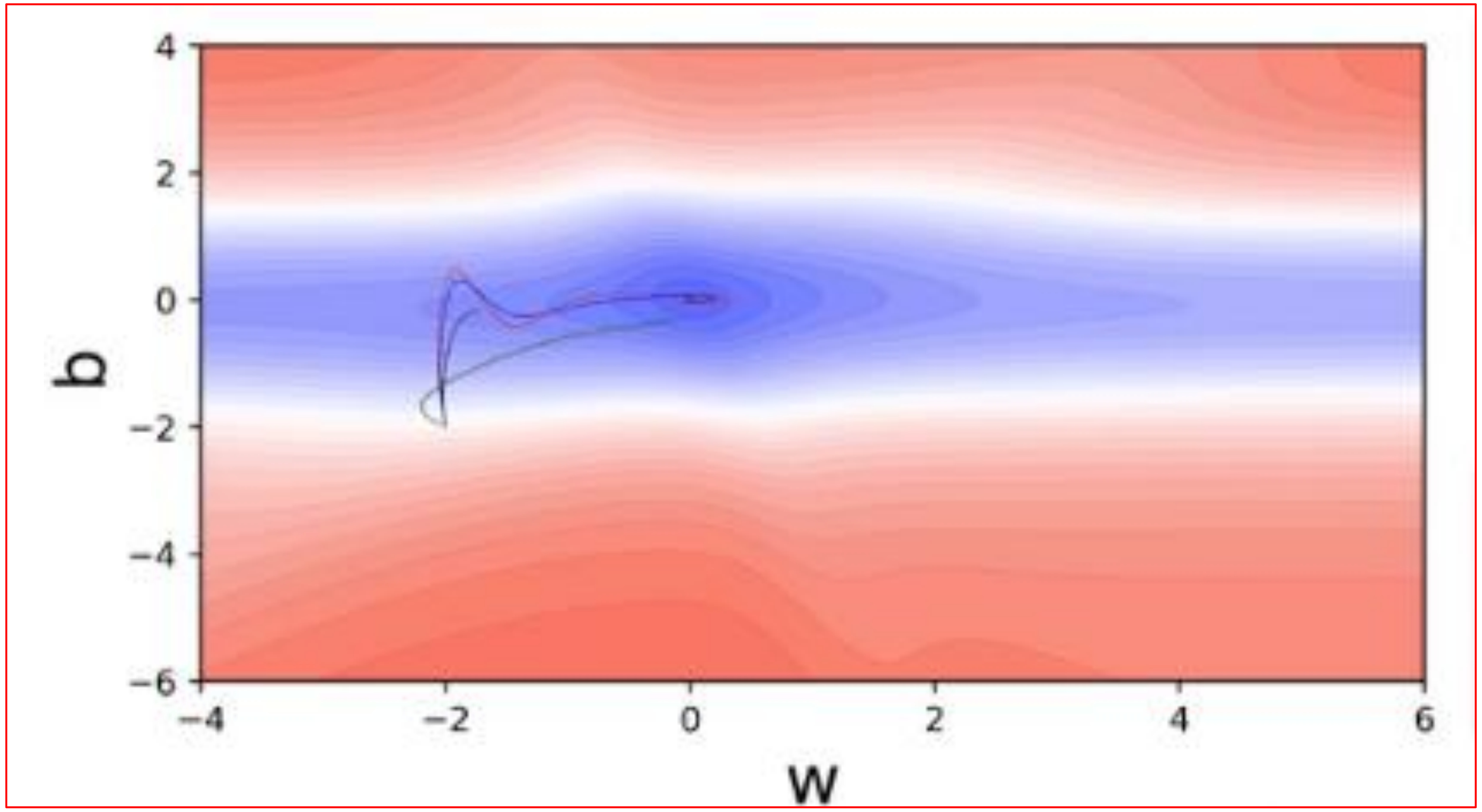
2. AdaGrad (Adaptive Gradient)

- **Adagrad** adapts the learning rate based on the sparsity of features (uses different learning rate for each feature).
- So, the parameters with small updates **(sparse features)** have high learning rate whereas the parameters with large updates **(dense features)** have low learning rate.
- Can you suggest what we need to do to the parameter update equation in order to adjust the learning rate according to the sparsity of the features?

Motivation

$$\Theta = \Theta - \alpha \nabla_{\Theta} J(\Theta)$$

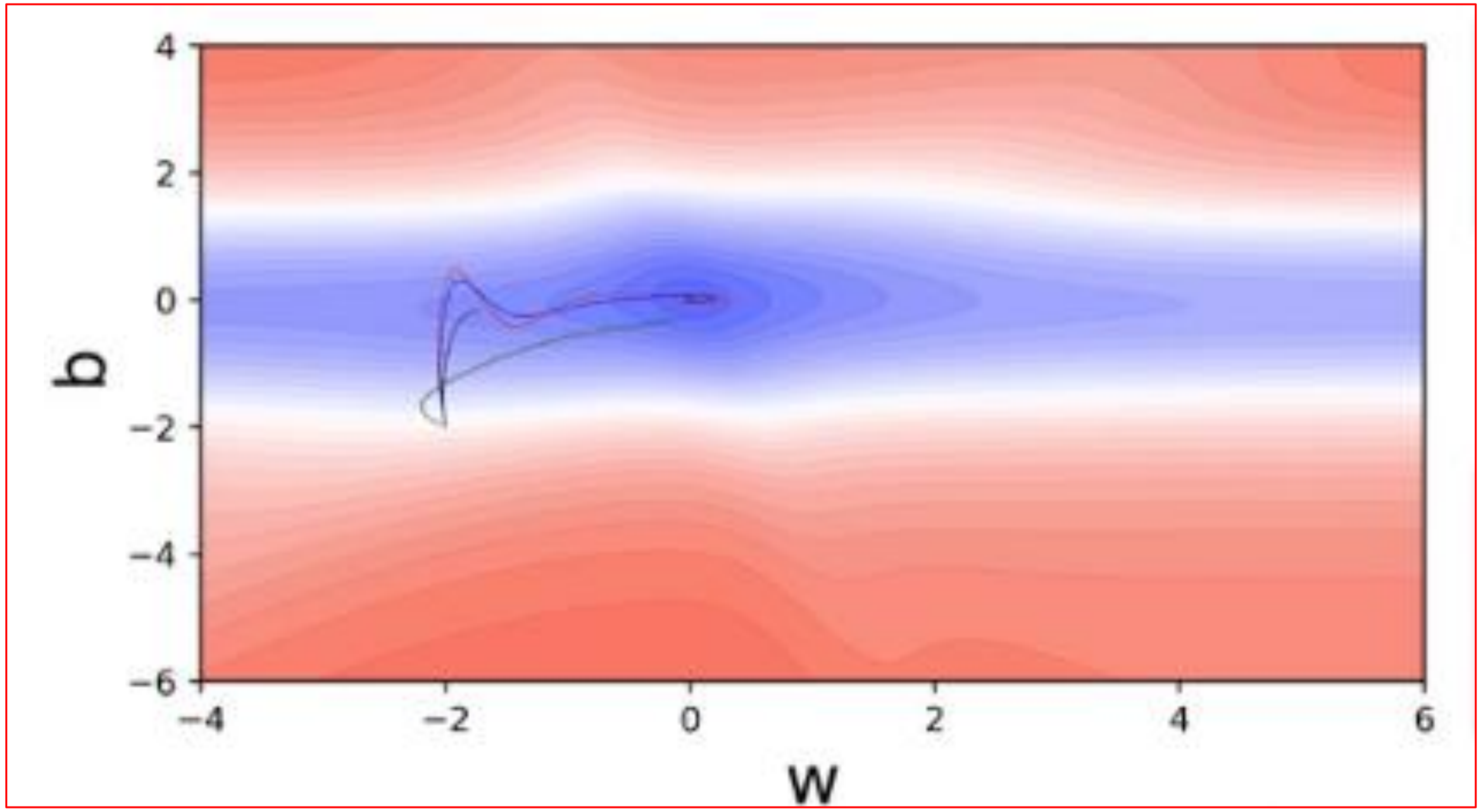
- For the real datasets, some of the features are sparse **i.e., having zero values**.
- Due to this for most of the cases, the corresponding gradient is zero and therefore the parameters update is also zero.



Motivation

$$\Theta = \Theta - \alpha \nabla_{\Theta} J(\Theta)$$

- The update should be boosted i.e., a high learning rate for sparse features.
- The learning rate should be adaptive for sparse data.
- If we are dealing with sparse features, then learning rate should be high whereas for dense features learning rate should be low.



AdaGrad (Adaptive Gradient)

- **Adagrad** adapts the learning rate based on the sparsity of features (uses different learning rate for each feature).
- So, the parameters with small updates **(sparse features)** have high learning rate whereas the parameters with large updates **(dense features)** have low learning rate.
- Can you suggest what we need to do to the parameter update equation in order to adjust the learning rate according to the sparsity of the features?

$$\begin{aligned} \mathbf{s}_{t+1} &= \mathbf{s}_t + \mathbf{g}_t^{\odot 2} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \mathbf{g}_t \odot \left(\sqrt{\mathbf{s}_{t+1}} + \varepsilon\right)^{\odot -1} \end{aligned}$$

AdaGrad

- Accumulated squared gradients and parameter update equations:

$$\begin{aligned} \mathbf{v}_{t+1} &= \mathbf{v}_t + \mathbf{g}_t^{\odot 2} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \mathbf{g}_t \odot \left(\sqrt{\mathbf{v}_{t+1}} + \varepsilon\right)^{\odot -1} \end{aligned}$$

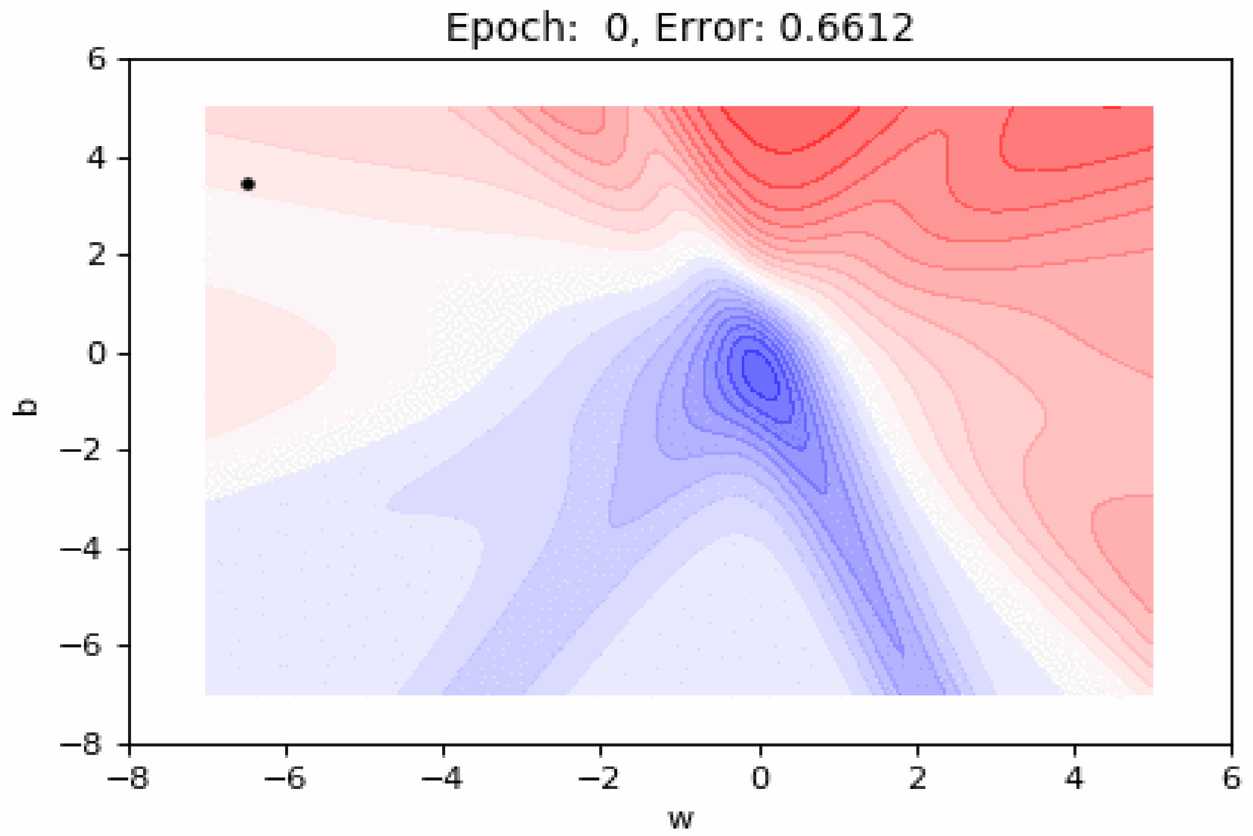
- Gradient at iteration t: $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$
- Hadamard (element-wise) product: $\odot$
- Element-wise square: $\mathbf{g}_t^{\odot 2} = \mathbf{g}_t \odot \mathbf{g}_t$
- Element-wise square root and division are also **component-wise**
- $\varepsilon > 0$: numerical stabilizer

Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla w_t$$

- **v(t)** accumulates the running sum of square of the gradients.
- Square of **∇w(t)** neglects the sign of gradients.
- **v(t)** indicates accumulated gradient up to time **t**.
- **Epsilon (ε)** in the denominator avoids the chances of divide by zero error.

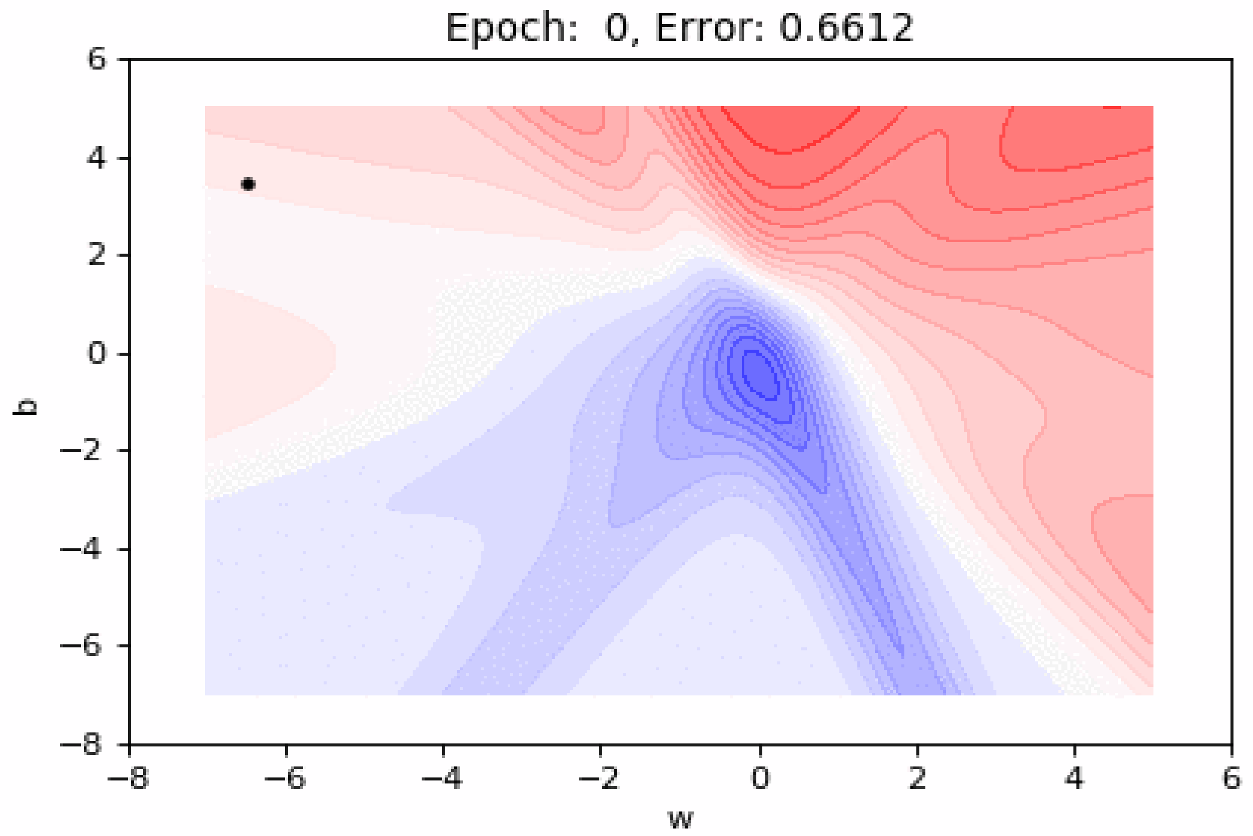


Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \nabla w_t$$

- If ***v*(*t*)** is low (due to less update up to time t) for a parameter, then the effective learning rate will be high
- And if ***v*(*t*)** is high for a parameter then effective learning rate will be low.



Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla w_t$$

- **Advantage:**
- Parameters corresponding to sparse features get better updates.

- **Disadvantages:**
- The learning rate decay very aggressively as the denominator grows **(not good for parameter corresponding to dense feature)**. Hence there is no update in value of parameter.
- Learning rate gets killed because denominator growing very fast. it reaches to **near** the minima point but not at the minima.

3. RMSProp (Root Mean Square Propagation)

- **How to solve the problem of Killing the learning rate?**

RMSProp

- Accumulated squared gradients and parameter update equations:

$$
\begin{aligned}
\mathbf{v}_{t+1} &= \beta \mathbf{v}_t + (1-\beta)\mathbf{g}_t^{\odot 2} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \mathbf{g}_t \odot \left(\sqrt{\mathbf{v}_{t+1}} + \varepsilon\right)^{\odot -1}
\end{aligned}
$$

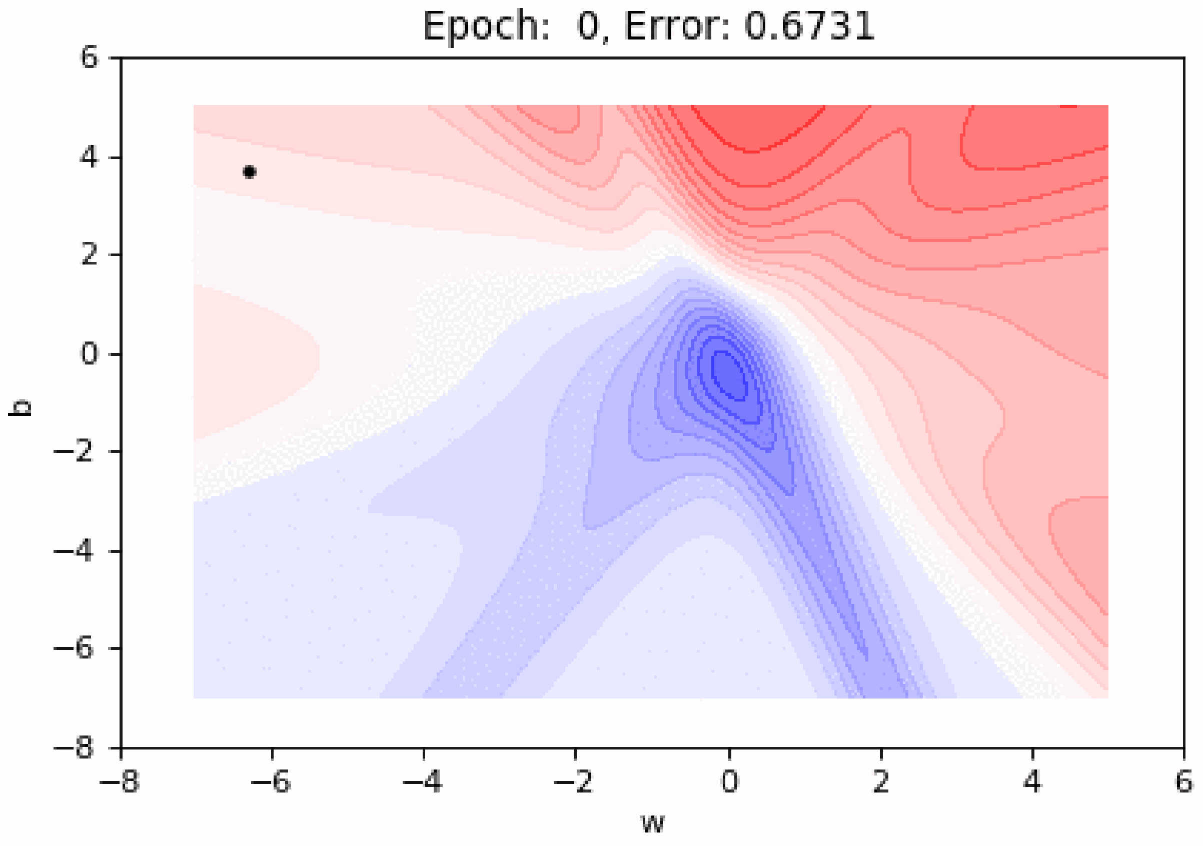
- Gradient at iteration t: $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$
- Hadamard (element-wise) product: $\odot$
- Element-wise square: $\mathbf{g}_t^{\odot 2} = \mathbf{g}_t \odot \mathbf{g}_t$
- Element-wise square root and division are also **component-wise**
- $\varepsilon > 0$: numerical stabilizer

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla w_t$$

RMSProp

- **RMSProp** Overcomes the decaying learning rate problem of **Adagrad** and prevents the rapid growth in ***v*(*t*)**.
- Instead of accumulating squared gradients from the beginning, it accumulates the previous gradients in some portion (weight).

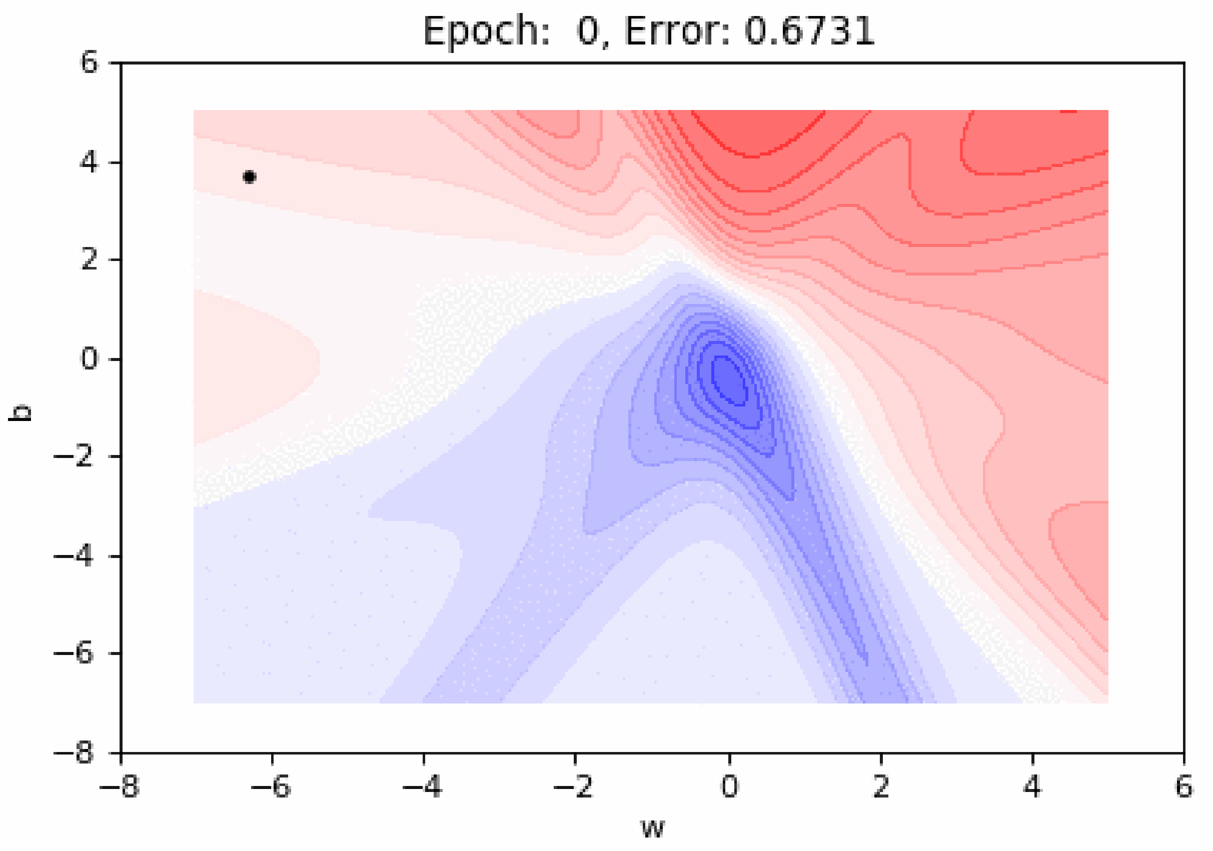


RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

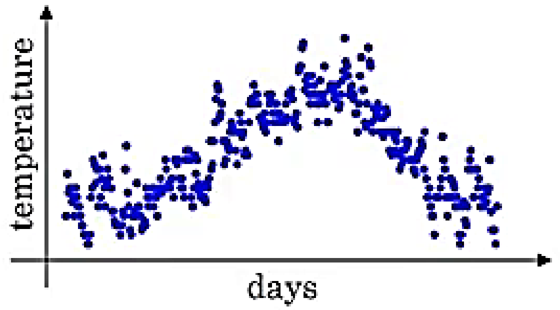
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \varepsilon}} \nabla w_t$$

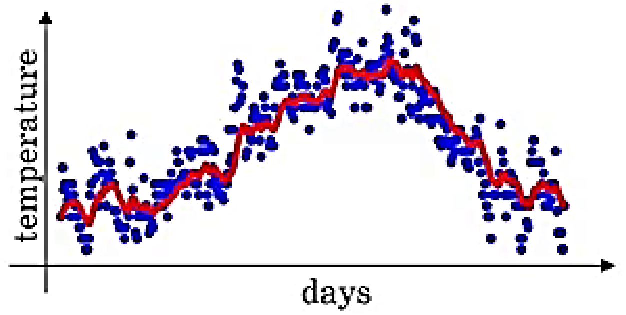
- ***v*(*t*)** is exponentially decaying average of all the previous squared gradients.
- Prevents rapid growth of ***v*(*t*)**.
- The algorithm keeps learning and tries to converge.

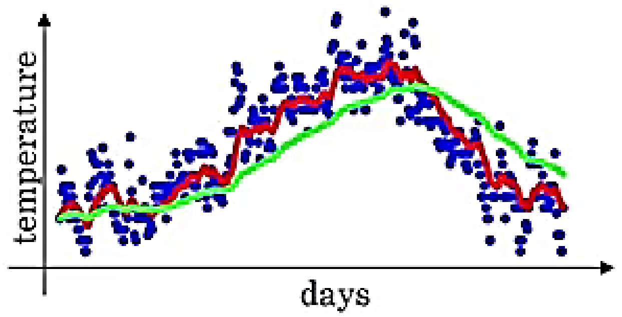


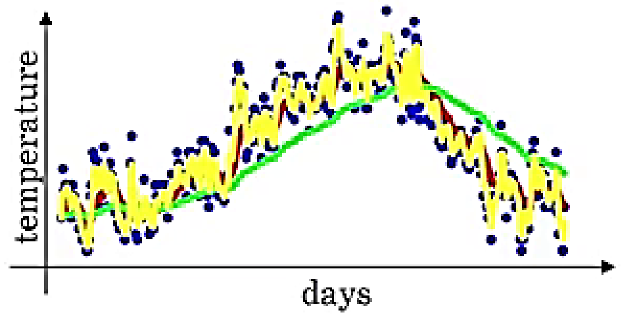
Exponentially Weighted Moving Average (EWMA)

- $v_t = \beta * v_{t-1} + (1 - \beta) T_t$
- Averaging over $\frac{1}{1-\beta}$ days
- β**=0.9** averaging over **10** days
- β**=0.98** averaging over **50** days
- β**=0.5** averaging over **2** days









4. Adam (Adaptive Momentum)

- **Can we do better by combining both the adaptive gradient and the momentum concepts?**

- **Adagrad** adapts the learning rate based on the sparsity of features.
- In **Adagrad** and **RMSProp** we were calculating different **learning rates** for different parameter. Can we have different **momentums** for different parameters.
- **Adam** algorithm introduces the concept of **adaptive momentum** along with **adaptive learning rate**.

Adam (Adaptive Moment) Estimation

- **Adam** is a combined form of **Momentum-based GD** and **RMSProp**.
- **Adam** computes the exponentially decaying average of previous gradients ***m*(*t*)** along with an adaptive learning rate.
- In **Momentum-based GD**, previous gradients (history) are used to compute the current gradient whereas, in **RMSProp** previous gradients (history) are used to adjust the learning rate based on the features.
- ***Adam deals with adaptive learning rate and adaptive momentum***

Adam

$$\begin{aligned}\mathbf{m}_{t+1} &= \beta_1 \mathbf{m}_t + (1-\beta_1)\mathbf{g}_t \\ \mathbf{v}_{t+1} &= \beta_2 \mathbf{v}_t + (1-\beta_2)\mathbf{g}_t^{\odot 2}\end{aligned}$$

- Bias correction:

$$\hat{\mathbf{m}}_{t+1} = \frac{\mathbf{m}_{t+1}}{1-\beta_1^{t+1}}, \hat{\mathbf{v}}_{t+1} = \frac{\mathbf{v}_{t+1}}{1-\beta_2^{t+1}}$$

- Update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \hat{\mathbf{m}}_{t+1} \odot \left(\sqrt{\hat{\mathbf{v}}_{t+1}} + \varepsilon\right)^{\odot -1}$$

- Gradient at iteration t: $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$
- Hadamard (element-wise) product: $\odot$
- Element-wise square: $\mathbf{g}_t^{\odot 2} = \mathbf{g}_t \odot \mathbf{g}_t$
- Element-wise square root and division are also **component-wise**
- $\varepsilon > 0$: numerical stabilizer

Adam

- Typically β1 = 0.9, β2 = 0.999, and ε = 1e-8
- η can work fine for the values 0.0001 and 0.001
- Generally, Adam with mini-batch is preferred for the training of deep neural networks.

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Adam

$$m_t = \beta_1 * m_{t-1} + (1-\beta_1)(\nabla w_t)$$

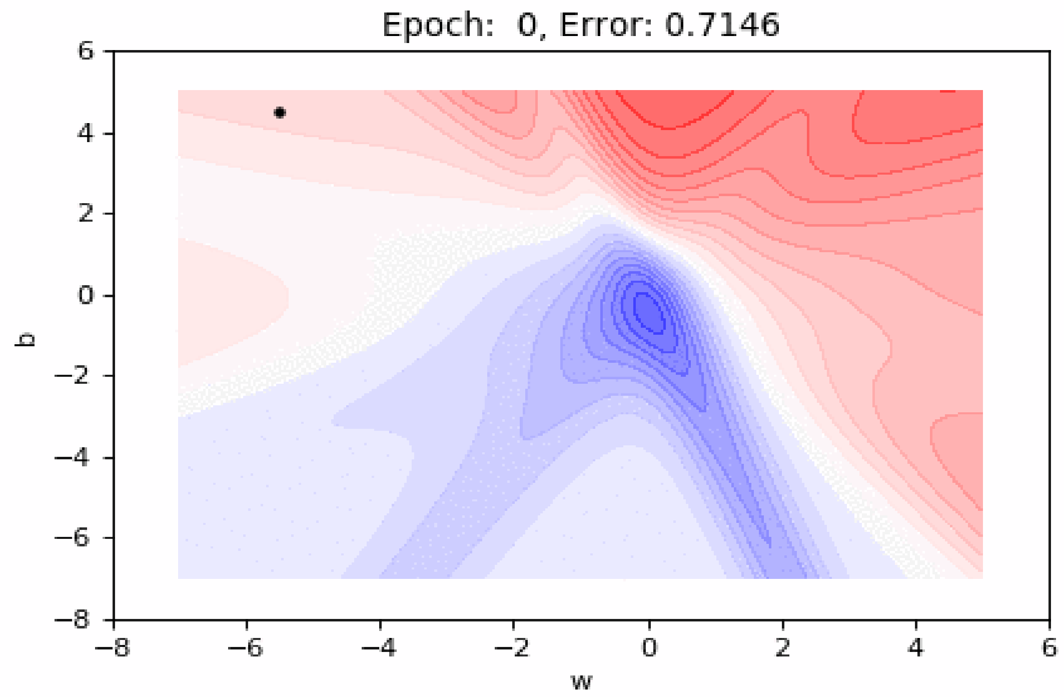
$$v_t = \beta_2 * v_{t-1} + (1-\beta_2)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(\hat{v}_t)} + \epsilon} \hat{m}_t$$

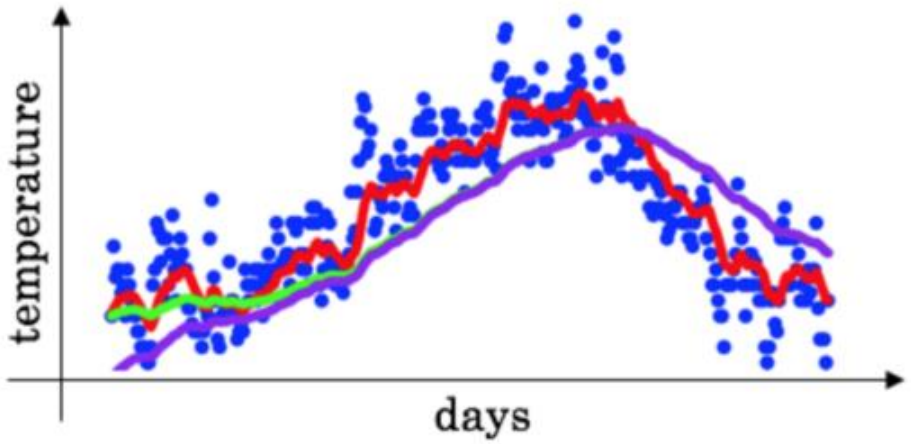
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Bias correction terms



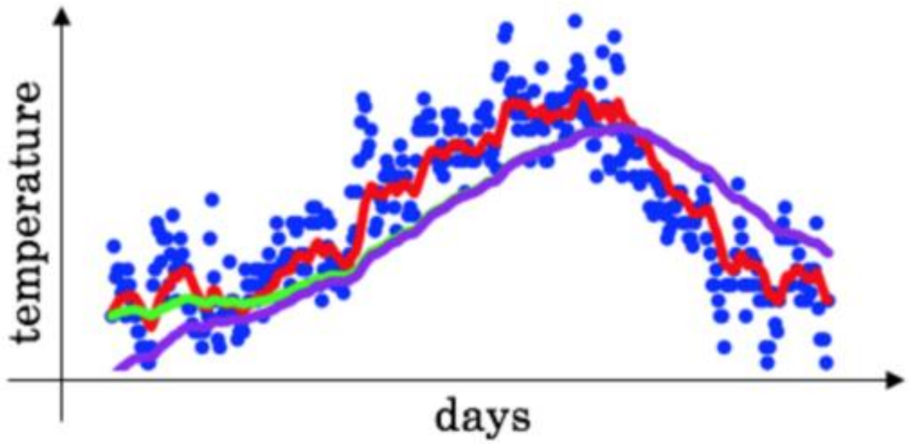
Bias Correction



- Making **EWMA** more accurate — the curve starts from 0, there are not many values to average on in the initial days. Thus, the curve is lower than the correct value initially and then moves in line with expected values.
- **Figure:** The ideal curve should be the GREEN one, but it starts as the PURPLE curve since the values initially are zero.

Bias Correction

- $v_t = \beta * v_{t-1} + (1 - \beta) T_t$
- **Example (β=0.98):** Starting from $t = 0$ and moving forward,
- $v_0 = 0$
- $v_1 = 0.98 * v_0 + 0.02 * T_1 = 0.02 * T_1$
- $v_2 = 0.98 * v_1 + 0.02 * T_2 = 0.0196 * T_1 + 0.02 * T_2$
- The initial values of v_t will be very low which need to be compensated. This is done as follows.
- For bias correction, modify $v_t = \frac{v_t}{1-\beta^t}$
- For t = 1 ➔ $v_1 = \frac{v_1}{1-\beta^1} = \frac{v_1}{1-0.98} = \frac{v_1}{0.02} = \frac{0.02 * T_1}{0.02} = T_1$
- For t = 2 ➔ $v_2 = \frac{v_2}{1-\beta^2} = \frac{v_2}{1-(0.98)^2} = \frac{0.0196*T_1+0.02*T_2}{0.0396}$
- When **t** is large $1 - \beta^t \approx 1$

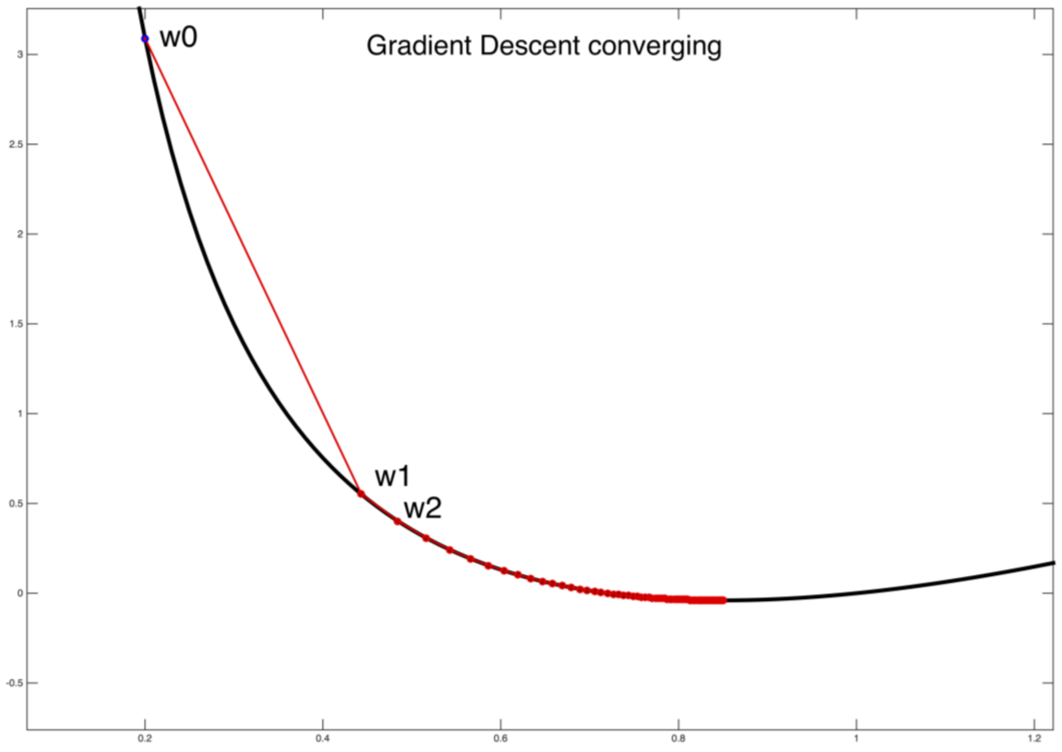


5. Newton’s Method

- GD algorithm is a first-order optimization method, as it uses first-order information (i.e., the gradient) to find the minimum.
- We linearly approximate the cost function at the current iteration.
- As the update step being larger and larger, this approximation is no longer valid.
- By approximating our objective function linearly at each step, we are only working with very limited local information, and we thus have to be cautious.

- Can we use information from the **second derivative** to do better?

Newton’s Method, the intuition



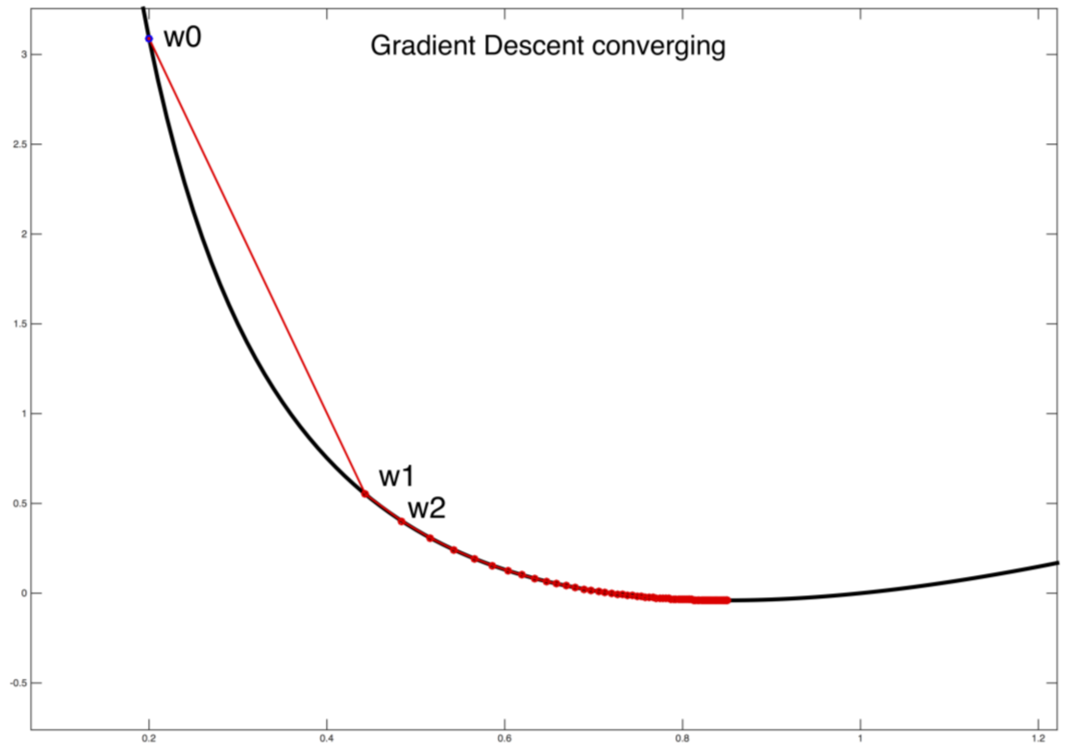
- Newton’s method can be understood as **gradient descent with a curvature-adaptive learning rate**.
- The second derivative measures the "rate of change of the rate of change," revealing the curve's concavity (how it bends).
- The reciprocal of the second derivative automatically scales the step size to match the local curvature.
- Gradient descent update (1D):

$$\theta_{t+1} = \theta_t - \alpha f^{'}(\theta_t)$$

- Newton Update (1D):

$$\theta_{t+1} = \theta_t - \frac{f'(\theta_t)}{f''(\theta_t)}$$

Newton’s Method, the intuition



- Gradient descent update (1D):
$$\theta_{t+1} = \theta_t - \alpha f'(\theta_t)$$
- Newton Update (1D):
$$\theta_{t+1} = \theta_t - \frac{f'(\theta_t)}{f''(\theta_t)}$$
- This looks exactly like GD with an effective learning rate $\alpha_{eff} = \frac{1}{f''(\theta_t)}$
- **For large curvature;**
- $f''(\theta_t) \gg 1 \Rightarrow \alpha_{eff} \ll 1$ → **Small step** (prevents overshooting)
- **For small curvature;**
- $f''(\theta_t) \ll 1 \Rightarrow \alpha_{eff} \gg 1$ → **Large step** (accelerates progress)

Newton's Method, (N-dimensional)

- Generalizing to **n** dimensions, the first derivative is replaced by the gradient, and the second derivative is replaced by the Hessian $\boldsymbol{H} = \nabla^2 J$.
- For Newton method the **update step** is $-\boldsymbol{H}^{-1}\boldsymbol{\nabla} J(\boldsymbol{\theta})$
- Gradient descent:

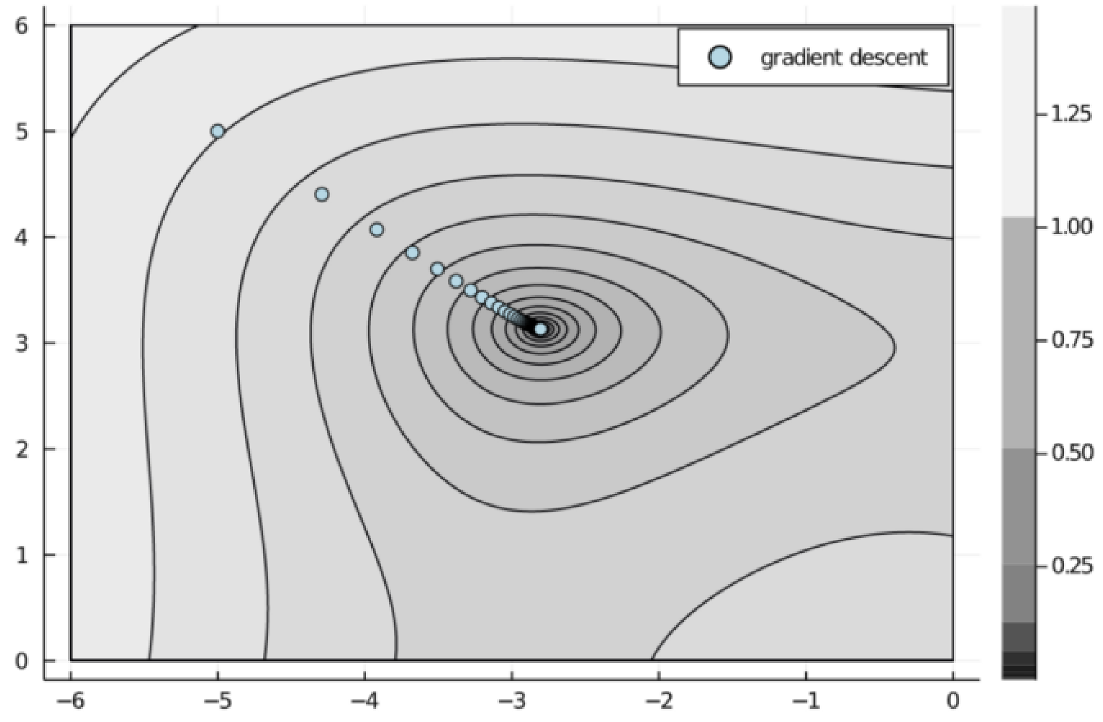
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla J(\boldsymbol{\theta}_t)$$

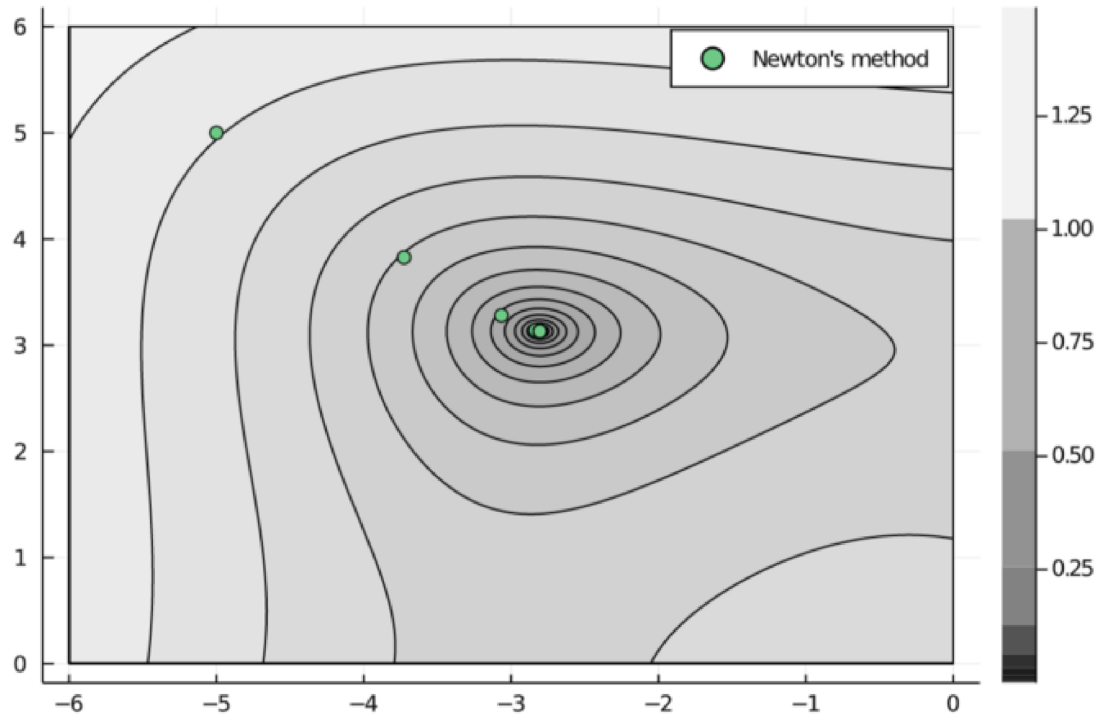
- Newton's method:

$$\boxed{\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{H}^{-1} \nabla J(\boldsymbol{\theta}_t)}$$

Newton's Method

- Consider this figure where we want to minimize our loss function, starting from the point **(-5, 5)**. For a suitably chosen learning rate, gradient descent takes **229** steps to converge to the minimum. On the other hand, Newton's method converges to the minimum in only **six** steps!





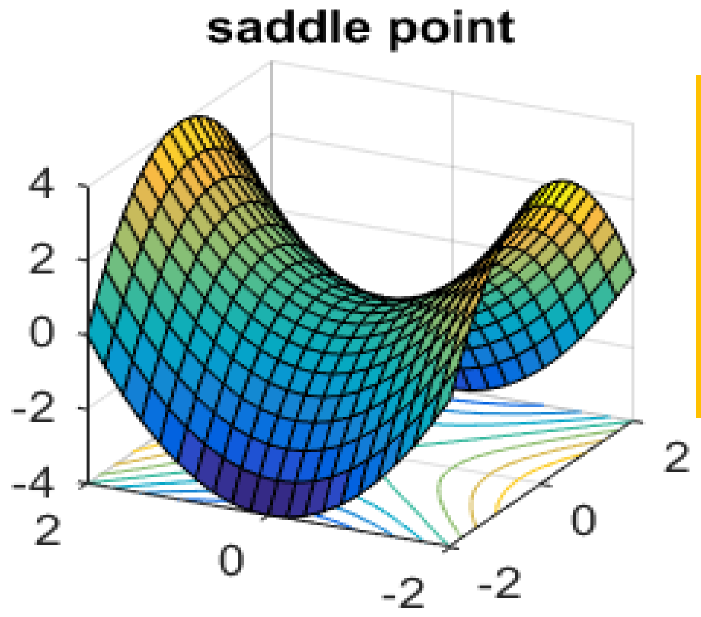
Newton’s Method, Drawbacks

- **Why ever use gradient descent if Newton’s method converges to the minimum so much faster?**

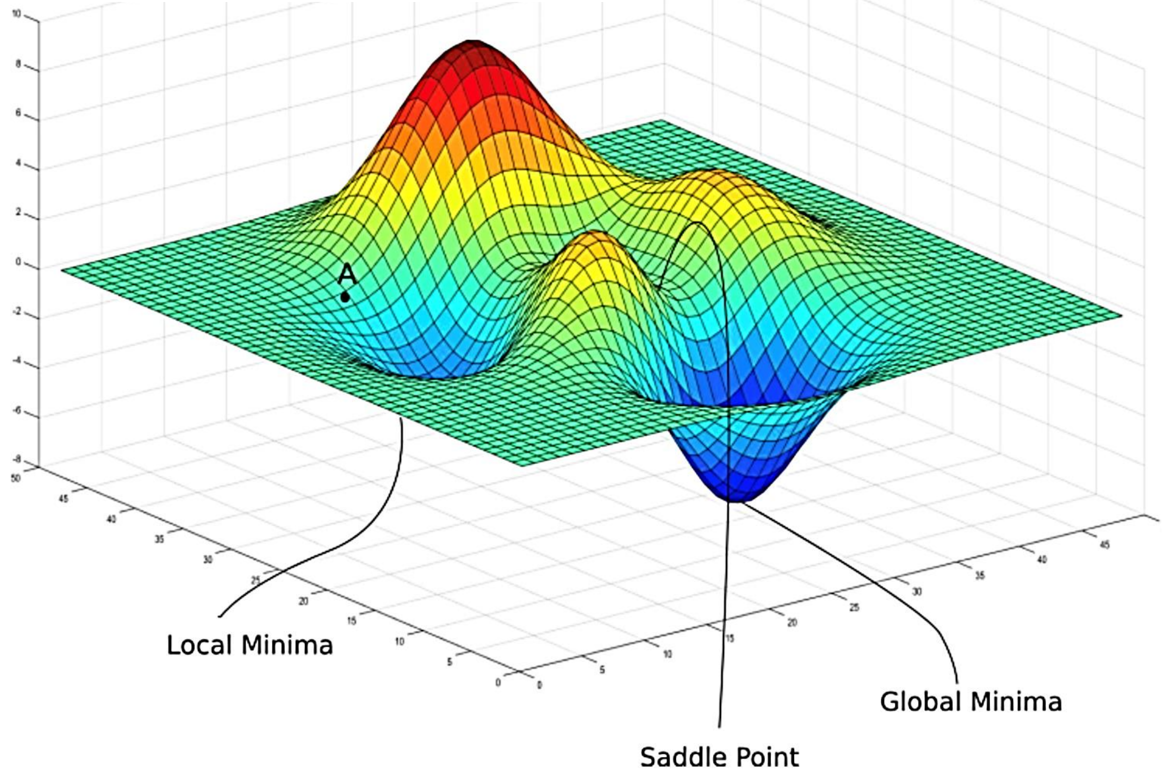
- It is sensitive to initial conditions. This is especially apparent if our objective function is non-convex.
- Unlike gradient descent, which ensures that we are always going downhill by always going in the direction opposite the gradient, Newton’s method fits a paraboloid to the local curvature and proceeds to move to the stationary point of that paraboloid.

Newton's Method, Drawbacks

- Depending on the local behavior of our initial point, **Newton's method could take us to a maximum or a saddle point instead of a minimum**.
- This problem is worsened in higher dimensional non-convex functions, where saddle points are much more likely to occur compared to minimum points.



Saddle point is a point on a curved surface at which the curvatures in two mutually perpendicular planes are of opposite signs



6. Quasi-Newton Methods

- For **n** dimensions
- In quasi-Newton methods, instead of computing the actual Hessian, we just approximate it with a positive-definite matrix **B**, which is updated from iteration to iteration using information computed from previous steps. (we require **B** to be positive-definite because we are optimizing a convex function, and this automatically takes care of the symmetry requirement of the Hessian).
- This scheme would yield a much less costly algorithm compared to Newton’s method, because instead of computing a large number of new quantities at each iteration, we’re largely making use of previously computed quantities.

Quasi-Newton Methods

- Finding $\boldsymbol{B}$ depends on the quasi-Newton used method.
- Quasi-Newton methods are a generalization of the secant method to find the root of the first derivative for multidimensional problems.
- In more than one dimension, the quasi-Newton condition does not uniquely specify the Hessian estimate $\boldsymbol{B}$, and we need to impose further constraints on $\boldsymbol{B}$ to solve for it.
- **Different quasi-Newton methods offer their own method for constraining the solution**.

Quasi-Newton Methods

- One of the main advantages of quasi-Newton methods over Newton's method is that the Hessian matrix (or, in the case of quasi-Newton methods, its approximation) $\boldsymbol{B}$ does not need to be inverted.
- Newton's method, and others such as interior point methods, require the Hessian to be inverted, which is typically implemented by solving a system of linear equations and is often quite costly.
- In contrast, quasi-Newton methods usually generate an estimate of $\boldsymbol{B}^{-1}$ directly.

BFGS

- Various quasi-Newton methods have been developed over the years, and they differ in how the approximate Hessian $\boldsymbol{B}$ is updated at each iteration.
- One of the most widely used quasi-Newton methods is the **BFGS** method.
- (Suggested **independently** by **B**royden, **F**letcher, **G**oldfarb, and **S**hanno, in 1970).

BFGS

- For multidimensional the quasi-Newton condition $\boldsymbol{B_{k+1}\Delta x_k = y_k}$ is underdetermined.
- To determine an update scheme for $\boldsymbol{B}$ then, we will need to impose additional constraints.
- One such constrain that we've already mentioned is the symmetry and positive-definiteness of $\boldsymbol{B}$ — these properties should be preserved in each update.
- Another desirable property we want is for $\boldsymbol{B_{k+1}}$ to be sufficiently close to $\boldsymbol{B_k}$ at each update $\boldsymbol{k+1}$ ***(for the secant line to be a good approximation)***. A formal way to characterize the notion of "closeness" for matrices is the matrix norm. Thus, we should look to minimize the quantity $||\boldsymbol{B_{k+1} - B_k}||$.
- Putting all our conditions together, we can formulate our problem as:
  - $\min\limits_{\boldsymbol{B_{k+1}}} ||\boldsymbol{B_{k+1} - B_k}||$ subject to $\boldsymbol{{B_{k+1}}^T = B_{k+1}}$ and $\boldsymbol{B_{k+1}\Delta x_k = y_k}$.

BFGS

- One final implementation detail that we previously glossed over: since an update of $\mathbf{B}^{-1}$ depends on its previous value, we have to initialize $\mathbf{B}_0^{-1}$ to kick off the algorithm.
- There are two natural ways to do this. The first approach is to set $\mathbf{B}_0^{-1}$ to the **identity matrix**, in which case the first step will be equivalent to vanilla gradient descent, and subsequent updates will incrementally refine it to get closer to the inverse Hessian.
- Another approach would be to compute and invert the true Hessian at the initial point. This would start the algorithm off with more efficient steps but comes at an initial cost of computing the true Hessian and inverting it.
- ***Hint:*** practically in BFGS implementation we multiply the update term by a **learning rate**.

Resources

- Andrew Ng, Machine Learning, Stanford University, Coursera
- https://ruder.io/optimizing-gradient-descent/index.html#batchgradientdescent
- https://towardsdatascience.com/calculus-behind-linear-regression-1396cfd0b4a9
- https://towardsdatascience.com/why-gradient-descent-isnt-enough-a-comprehensive-introduction-to-optimization-algorithms-in-59670fd5c096
- https://towardsdatascience.com/why-gradient-descent-isnt-enough-a-comprehensive-introduction-to-optimization-algorithms-in-59670fd5c096
- https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3
- https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3
- https://www.charlesbordet.com/en/gradient-descent/#how-does-it-work
- https://www.charlesbordet.com/en/gradient-descent/#how-to-find-a-good-value-for-the-learning-rate
- https://www.charlesbordet.com/en/gradient-descent/#what-to-do-in-case-of-local-minima
- https://towardsdatascience.com/why-gradient-descent-isnt-enough-a-comprehensive-introduction-to-optimization-algorithms-in-59670fd5c096
- https://medium.com/hackernoon/implementing-different-variants-of-gradient-descent-optimization-algorithm-in-python-using-numpy-809e7ab3bab4
- https://ranasinghiitkgp.medium.com/optimization-algorithm-d62ac8f597b1
- http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Resources

- https://towardsdatascience.com/why-gradient-descent-isnt-enough-a-comprehensive-introduction-to-optimization-algorithms-in-59670fd5c096
- https://ranasinghiitkgp.medium.com/optimization-algorithm-d62ac8f597b1
- https://www.youtube.com/watch?v=lAq96T8FkTw&ab_channel=DeepLearningAI
- https://www.youtube.com/watch?v=NxTFlzBjS-4&ab_channel=DeepLearningAI
- https://www.youtube.com/watch?v=lWzo8CajF5s&ab_channel=DeepLearningAIDeepLearningAI
- https://medium.datadriveninvestor.com/exponentially-weighted-average-for-deep-neural-networks-39873b8230e9
- https://towardsdatascience.com/understanding-gradient-descent-and-adam-optimization-472ae8a78c10
- DOI: 10.1177/0735633118757015
- https://www.asimovinstitute.org/author/fjodorvanveen/
- guru99.com
- https://towardsdatascience.com/an-introduction-to-reinforcement-learning-1e7825c60bbe

Resources

- Justin Solomon, Numerical Algorithms, Methods for Computer Vision, Machine Learning, and Graphics.
- https://towardsdatascience.com/bfgs-in-a-nutshell-an-introduction-to-quasi-newton-methods-21b0e13ee504
- https://ekamperi.github.io/machine%20learning/2019/07/28/gradient-descent.html
- https://en.wikipedia.org/wiki/Taylor_series
- https://en.wikipedia.org/wiki/Quasi-Newton_method#Search_for_extrema:_optimization
- https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm
- https://www.coursera.org/learn/multivariate-calculus-machine-learning?specialization=mathematics-machine-learning