Neural Networks

(P-ITEEA-0011)

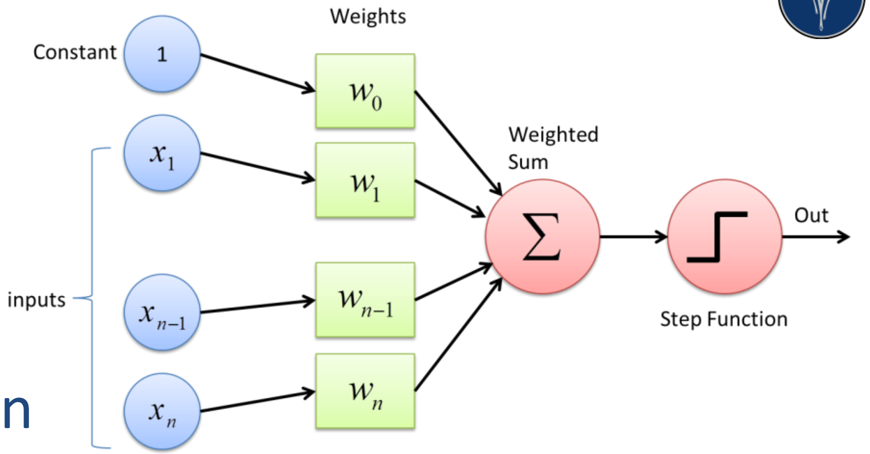
Gradient based optimization methods

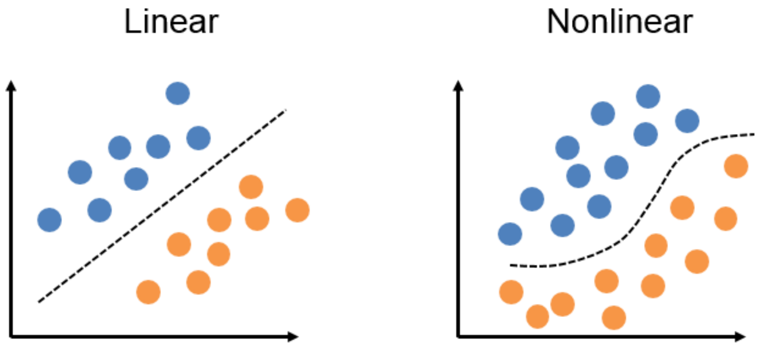
Akos Zarandy
Lecture 4
October 1, 2019

Contents

Recall: Single layer perceptron

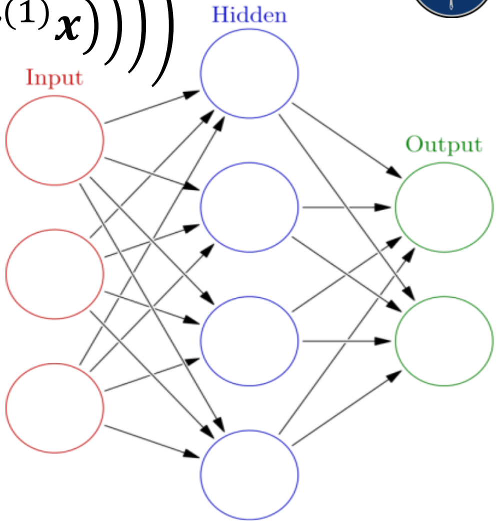
- $y = \varphi(\mathbf{w}^T \mathbf{x})$
- Decision boundary is a hyperplan
- Simple training method
- Convergence of training was proven
- Good for making decision in linearly separable cases
- In more complex decision situation
  - It turns out to be a toy

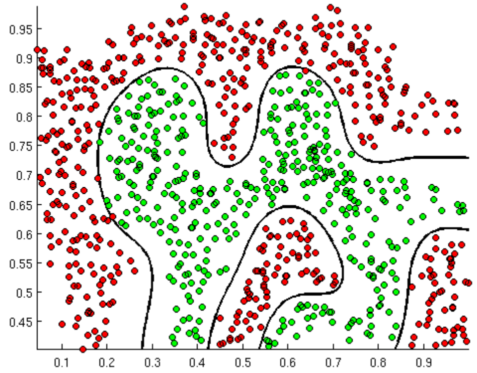




Recall: Multi-layer perceptron

- $Net(\mathbf{x}, \mathbf{W}) = \varphi^{(L)}\left(\boldsymbol{w}^{(L)}\varphi^{(L-1)}\left(\boldsymbol{w}^{(L-1)} \quad \ldots \quad \varphi^{(2)}\left(\boldsymbol{w}^{(2)}\varphi^{(1)}\left(\boldsymbol{w}^{(1)}\boldsymbol{x}\right)\right)\right)\right)$
- Can approximate an arbitrary function with arbitrary precision
- The same way, it can implement arbitrary decision boundary
- It can be trained even if F (or the boundary surface) is not known analytically or not even fully known
  - *Statistical learning:* It is enough to know equally distributed input/output pairs
- The partial gradient of the network can be also calculated for each weight coefficient or hidden layer neuron (back propagation)





What is learning (training)?

- Given:
  - Definition of the network architecture
    - Topology
    - Initial weights
    - Activation functions (nonlinearities)
    - Training set ($\boldsymbol{x}_i \rightarrow \boldsymbol{y}_i$)
- Goal:
  - Calculation of the optimal weight composition: W_{opt}
    1. *Having a function to approximate*

$$\mathbf{w}_{\text{opt}} : \min_{\mathbf{w}} \left\| \mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right\|^2 = \min_{\mathbf{w}} \int .. \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x},\mathbf{w}) \right)^2 dx_1 ... dx_N$$

    2. *Having a set of observations from a stochastic process*

$$\mathbf{w}_{\text{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \left(d_k - Net(\mathbf{x}_k, \mathbf{w}) \right)^2$$

Stochastic process is a process, where we cannot observe the exact values. In these processes, our observations are always corrupted with some random noise.

OPTIMIZATION!!!

Optimization

- Given an ***Objective function*** to optimize
  - Also called: Error function, Cost function, Loss function, Criterion
  - Derived from the network topology and the input/output pairs
- Function types:
  - Quadratic, in case of regression (stochastic process)

$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

  - Conditional log-likelihood, in case of classification (classification process)
    - The sum of the negative logarithmic likelihood (probability) is minimized

$$\Theta(\mathbf{w}) = \sum_{k=1}^{K} -logP(\mathbf{y_k}|\mathbf{x_k};\mathbf{w})$$

Optimizations

- *Here we always minimize the objective function*
  - *Parametric equation*
    - **x** *are the variables*
    - **w** *are the parameters*
- *Optimization targets to find the optimal weights*

$$\mathbf{w}_{opt} = \min f\left(\mathbf{x}, \mathbf{d}, Net(\mathbf{x},\mathbf{w})\right)$$

goals:

- *Acceptable error level*
- *Acceptable computational time assuming reasonable computational effort*

Mathematics behind: Function analysis

- Assumptions
  - Poor conditioning
  - Conditioning number (Ratio of Eugen values): $\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$

$$f(x) = A^{-1}x \qquad A \in \Re^{n \times n}$$

  - Applied functions should be Lipschitz continuous or have Lipschitz continuous derivate

$$\forall x, \forall y, \ \left| f(x) - f(y) \right| \leq \mathrm{L} \left\| \mathrm{x} - \mathrm{y} \right\|_2$$

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output. (e.g. Matrix inversion)

(where:
L is the Lipschitz constant)

Basic idea of Gradient Descent

- There is a function, where

  $f(x)$

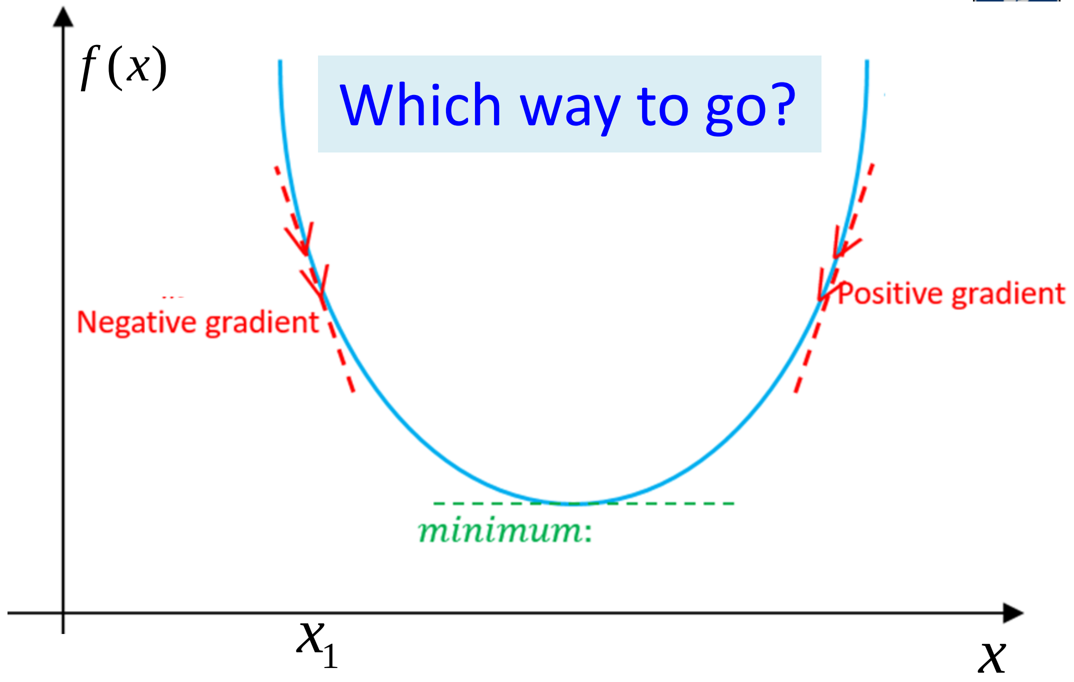
  and

  $f'(x)$

  - can be calculated at any points, but

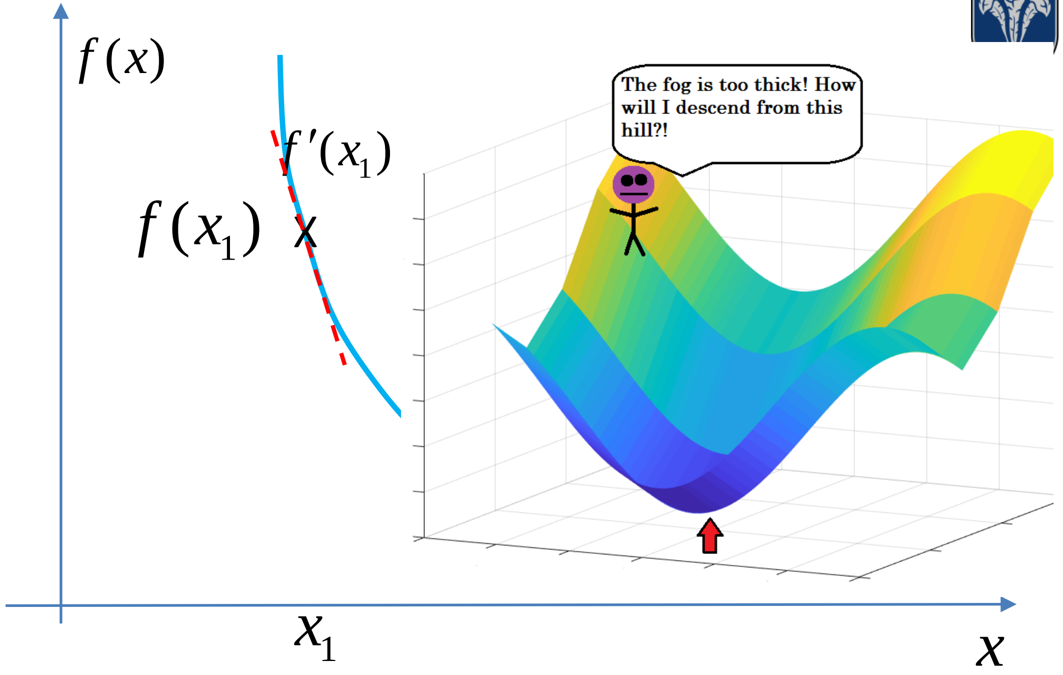
  $f'(x) = 0$

  - cannot.
  - Therefore the trace of the light blue line is not known.
  - We have to start out from one point (say x_1) and with an iterative method, we need to go towards the minimum



Basic idea of Gradient Descent

- We do not know where the curve is
- We know the value at $f(x_1)$
- We know the derivative at x_1 $f'(x_1)$
- Which way to go?
- Idea: follow the descending gradient!



Basic idea of Gradient Descent

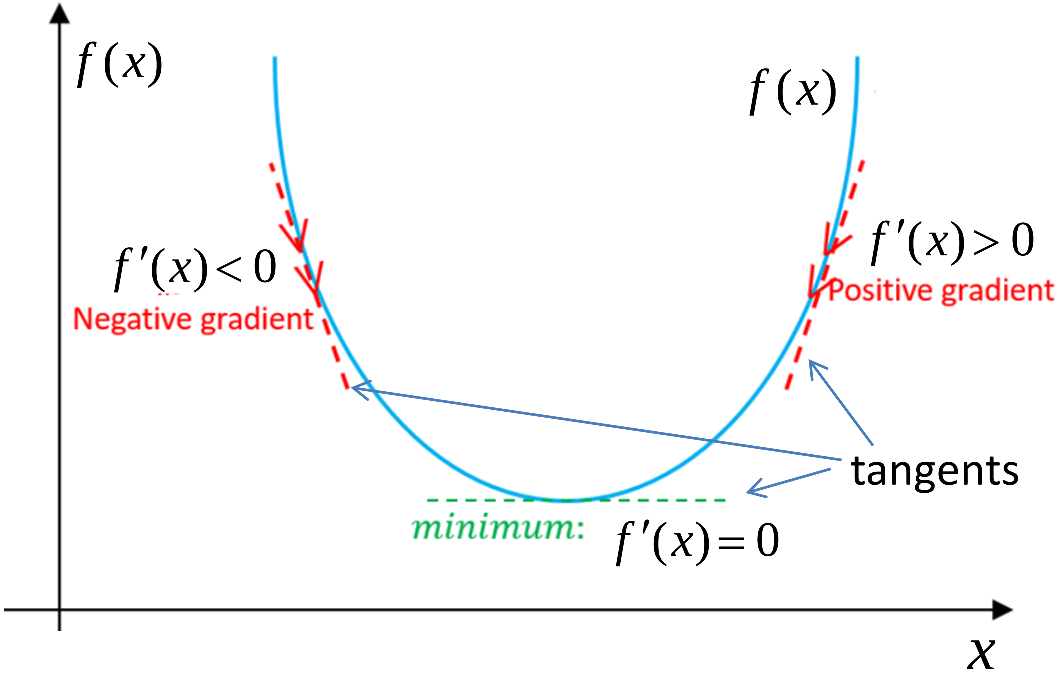
- Derivative means for small ε

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

- *therefore*

$$f\left(x - \varepsilon\, sign\left(f'(x)\right)\right) \leq f(x)$$

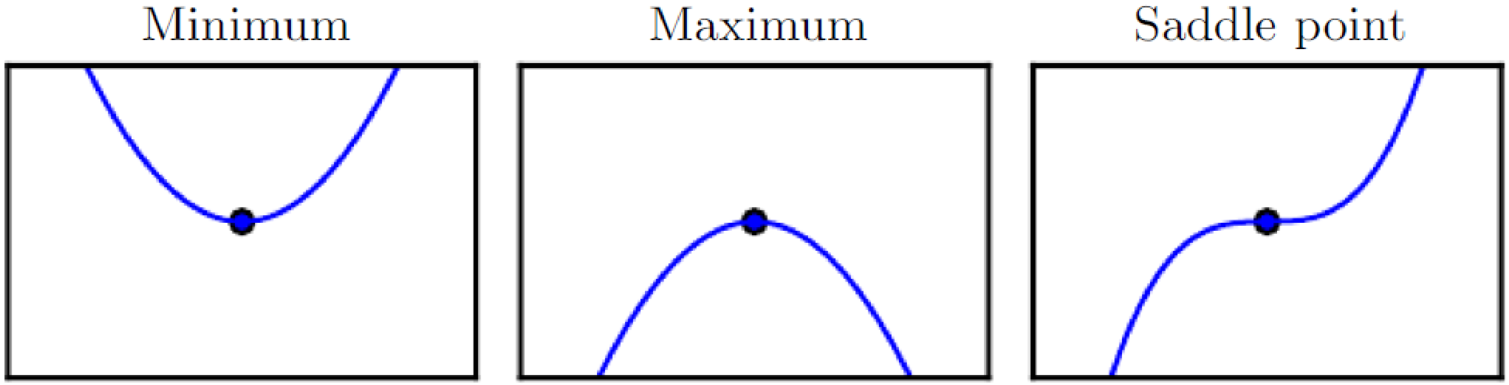
- This technique is called **Gradient Descent (Cauchy, 1847).**



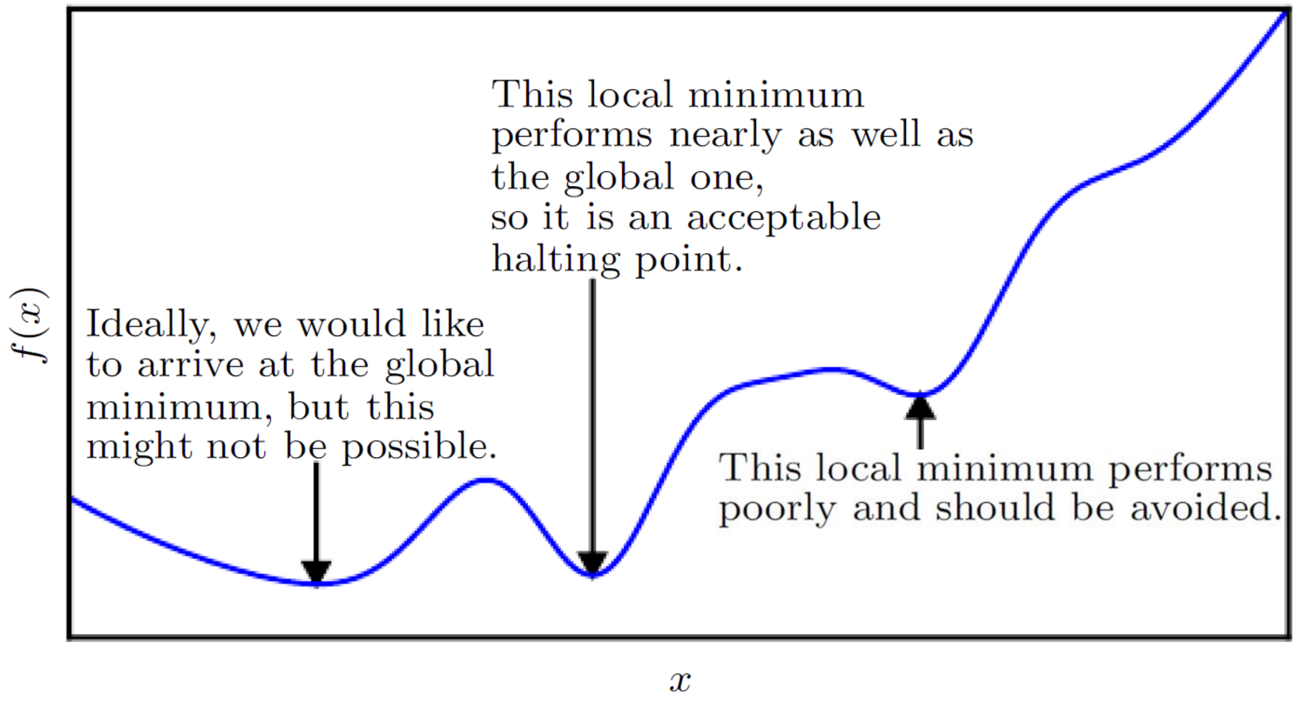
Optimization goal is to find the $f'(x) = 0$ position.
(Critical or stationary points)

Stationary points

- Local minimum, where *f`(x)=0*, and *f(x)* is smaller than all neighboring points
- Local maximum, where *f`(x)=0*, and *f(x)* is larger than all neighboring points
- Saddle points, where *f`(x)=0*, and neither minimum nor maximum



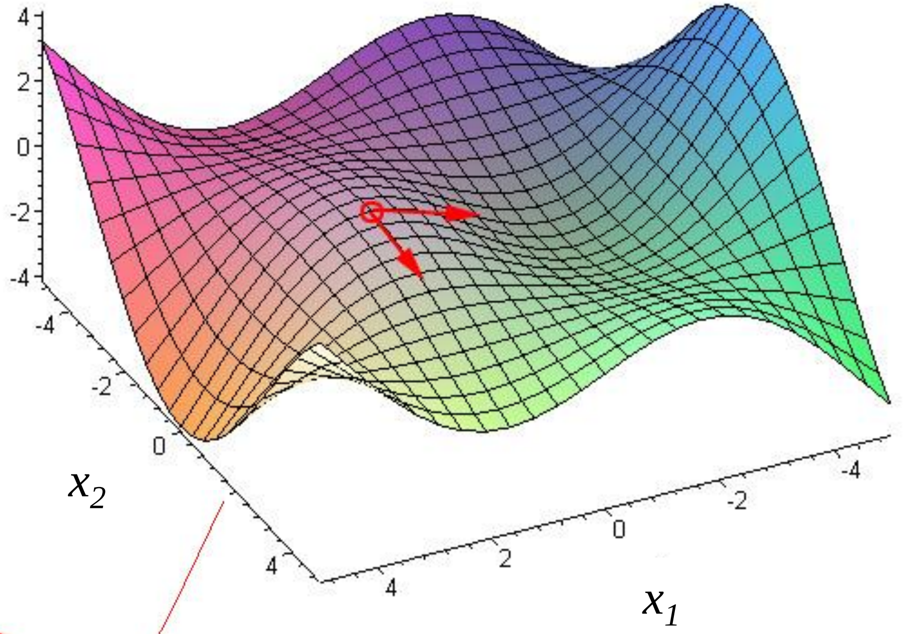
Local and global minimum



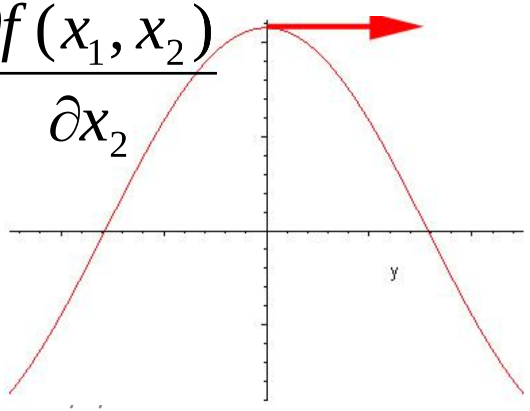
In neural network parameter optimization we usually settle for finding a value of f that is very low, but not necessarily minimal in any formal sense.

Multidimensional input functions I

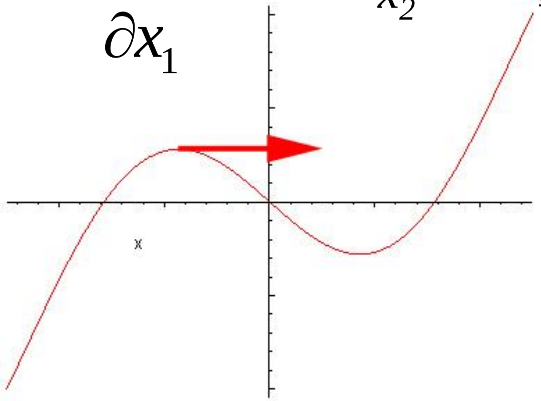
- In case of a vector scalar function
- In 2D, directional derivatives (slope towards x_1 and x_2):



$$\frac{\partial f(x_1, x_2)}{\partial x_2}$$



$$\frac{\partial f(x_1, x_2)}{\partial x_1}$$

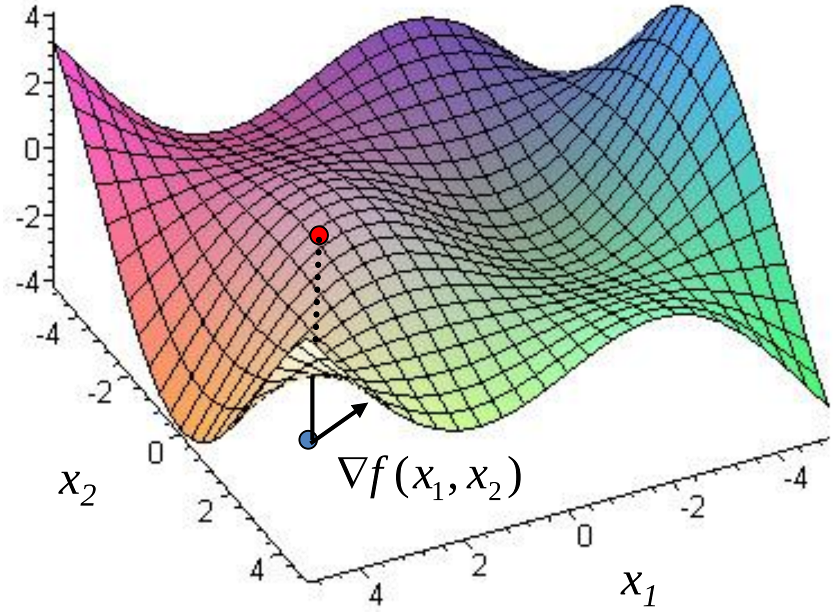


Multidimensional input functions II

- In case of a vector scalar function
- Gradient definition in 2D

$$f : R^2 \to R$$

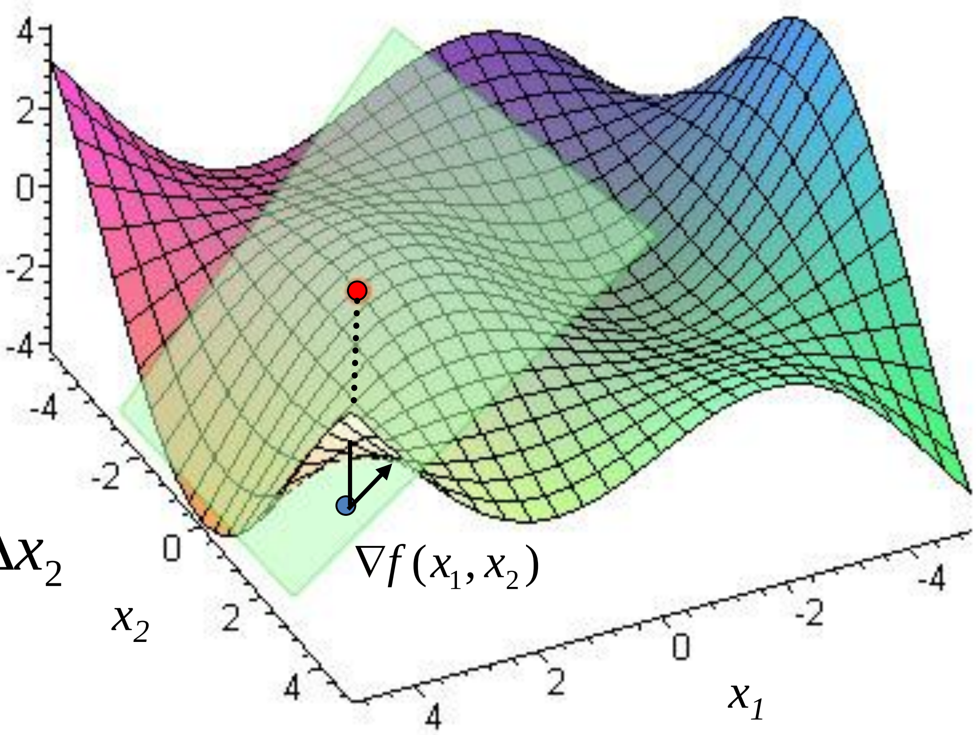
$$\nabla f(x_1, x_2) := \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} \end{pmatrix}$$



A vector in the in the x_1 - x_2 plane

Multidimensional input functions III

- The gradient defines (hyper) plane approximating the function infinitesimally at point $\mathbf{x}$ (x_1, x_2)



$$\Delta z = \frac{\partial f(x_1, x_2)}{\partial x_1} \cdot \Delta x_1 + \frac{\partial f(x_1, x_2)}{\partial x_2} \cdot \Delta x_2$$

Multidimensional input functions IV

- Directional derivative to an arbitrary direction **u** (**u** is unit vector) is the slope of *f* in that direction at point $\mathbf{x}$ (x_1, x_2):

$$\mathbf{u}^{\mathrm{T}}\nabla f(\mathbf{x})$$

- *f* decreases the fastest:

Not changing with u

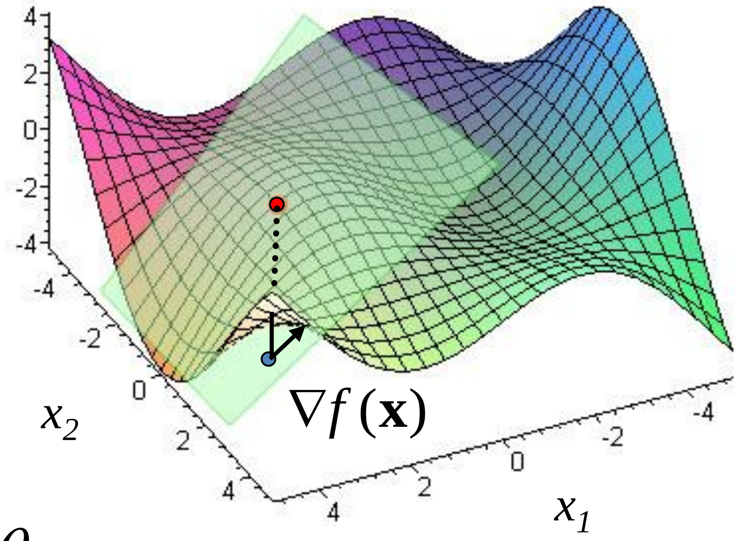
$$\min_{u,u^{\mathrm{T}}u=1} \mathbf{u}^{\mathrm{T}}\nabla f(\mathbf{x}) = \min_{u,u^{\mathrm{T}}u=1} \|\mathbf{u}\|_2 \|\nabla f(\mathbf{x})\|_2 \cos\theta$$

minimum at 180

- **u** is opposite to the gradient!!!

New points towards steepest descent:

$$\mathbf{x}' = \mathbf{x} - \varepsilon\nabla f(\mathbf{x})$$

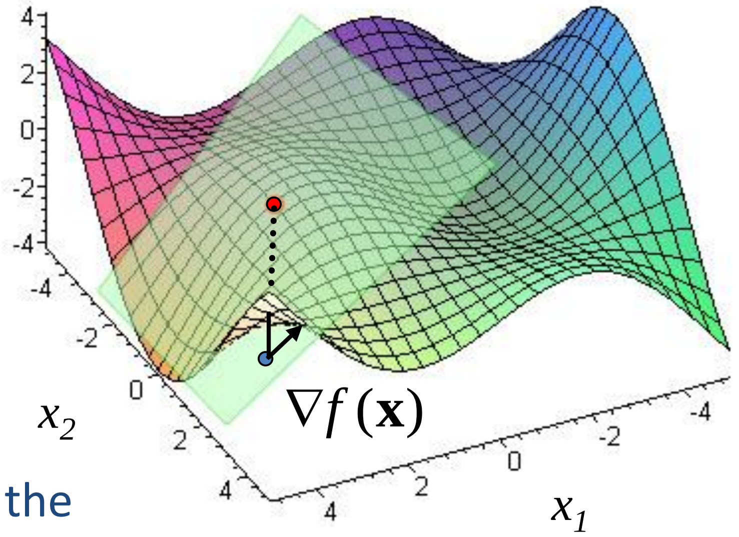


Gradient Descent in multidimensional input case

- Steepest gradient descent iteration

$$\mathbf{x}(n+1) = \mathbf{x}(n) - \varepsilon \nabla f\left(\mathbf{x}(n)\right)$$

- ε is the learning rate
- Choosing ε:
  - Small constant
  - Decreases as the iteration goes ahead
  - **Line search**: checked with several values, and the one selected, where $f(\mathbf{x})$ is the smallest
- Stopping condition of the gradient descent iteration
  - When the gradient is zero or close to zero



Jacobean Matrix

- Partial derivative of a vector → vector function
- Specifically, if we have a function $\mathbf{f} : \Re^m \to \Re^n$ then the Jacobian matrix $\mathbf{J} \in \Re^{n \times m}$ of $\mathbf{f}$ is defined such that: $\mathbf{J}_{i,j} = \frac{\partial}{\partial x_j} f(x_i)$

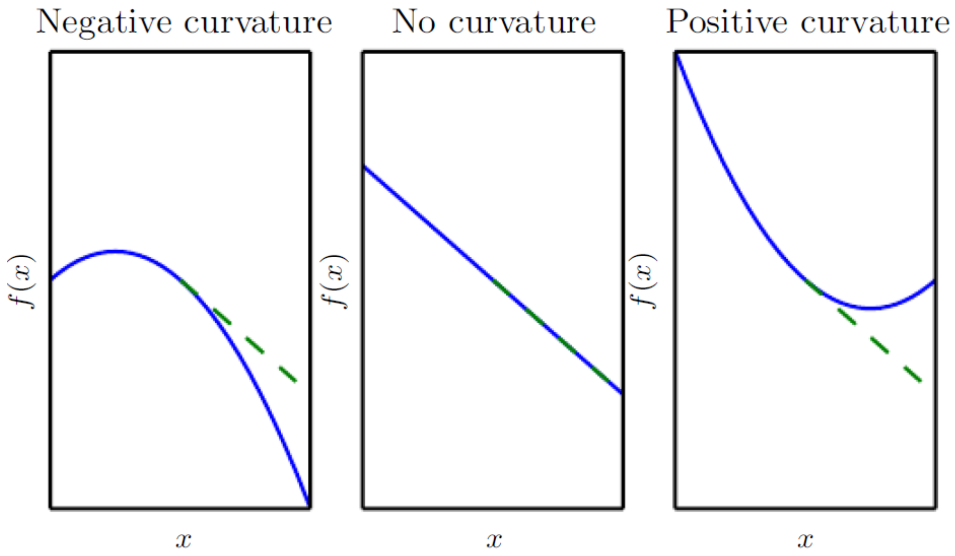
$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}$$

2nd derivatives

- 2nd derivative determines the curvature of a line in 1D
- In nD, it is described by the Hessian Matrix

$$H\left(f(x_{i,j})\right) = \frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial^2}{\partial x_j \partial x_i} f(x)$$

- The Hessian is the Jacobian of the gradient.



2nd order gradient descent method I

- 2nd derivative in a specific direction: $\mathbf{u}^{\mathrm{T}}\mathbf{H}\mathbf{u}$

- Second-order Taylor series approximation to the function *f(x) around the current point* $\mathbf{x}_0$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}}\mathbf{g} + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}}\mathbf{H}(\mathbf{x}-\mathbf{x}_0)$$

where:
g: gradient at x_0
H: Hessian at x_0

- stepping towards the largest gradient:

$$\mathbf{x}_0 - \varepsilon\,\mathbf{g} \approx \mathbf{x} \quad \rightarrow \quad \mathbf{x} - \mathbf{x}_0 \approx -\varepsilon\,\mathbf{g}$$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0 - \varepsilon\,\mathbf{g}) \approx f(\mathbf{x}_0) - \varepsilon\,\mathbf{g}^{\mathrm{T}}\mathbf{g} + \frac{1}{2}\varepsilon^2\,\mathbf{g}^{\mathrm{T}}\mathbf{H}\mathbf{g}$$

2nd order gradient descent method II

- Analyzing: $$\underset{\substack{\uparrow\\\text{Original value}}}{f(\mathbf{x}_0 - \varepsilon\,\mathbf{g}) \approx f(\mathbf{x}_0)} - \underset{\substack{\uparrow\\\text{Expected}\\\text{improvement}}}{\varepsilon\,\mathbf{g}^{\mathrm{T}}\mathbf{g}} + \underset{\substack{\uparrow\\\text{Correction due to}\\\text{curvature}}}{\frac{1}{2}\varepsilon^{2}\,\mathbf{g}^{\mathrm{T}}\mathbf{H}\mathbf{g}}$$
- When the third term is too large, the gradient descent step can actually move uphill.
- When it *is zero or negative,* the Taylor series approximation predicts that increasing ε *forever will decrease f* forever.
- In practice, the Taylor series is unlikely to remain accurate for large ε*, so* one must resort to more heuristic choices of ε *in this case.*
- *When it is positive,* solving for the optimal step

$$\varepsilon^{*} = \frac{\mathbf{g}^{\mathrm{T}}\mathbf{g}}{\mathbf{g}^{\mathrm{T}}\mathbf{H}\mathbf{g}}$$

Simplest 2nd order Gradient descent method: Newton Method

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}} \nabla f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}} \mathbf{H}\big(f(\mathbf{x}_0)\big)(\mathbf{x}-\mathbf{x}_0)$$

- Replacing $(\mathbf{x}-\mathbf{x}_0) \to \Delta\mathbf{x}$ and differentiating it with $\Delta\mathbf{x}$, assuming that we can jump to a minima, where: $\nabla f(\mathbf{x}) \approx 0$

$$0 = \frac{\partial}{\partial \Delta\mathbf{x}}\left(\underbrace{f(\mathbf{x}_0)}_{\text{Constant}\to 0} + \underbrace{\Delta\mathbf{x}^{\mathrm{T}}\nabla f(\mathbf{x}_0)}_{(\Delta x)' \to 1} + \underbrace{\frac{1}{2}\Delta\mathbf{x}^{\mathrm{T}}\mathbf{H}\big(f(\mathbf{x}_0)\big)\Delta\mathbf{x}}_{(½\,(\Delta x)^2)' \to \Delta x} \right) = \nabla f(\mathbf{x}_0) + \mathbf{H}\big(f(\mathbf{x}_0)\big)\Delta\mathbf{x}$$

Newton optimization:

$$\Delta\mathbf{x} = -\mathbf{H}\big(f(\mathbf{x}_0)\big)^{-1}\nabla f(\mathbf{x}_0) \qquad \mathbf{x}(n+1) = \mathbf{x}(n) - \eta\mathbf{H}\Big(f\big(\mathbf{x}(n)\big)\Big)^{-1}\nabla f\big(\mathbf{x}(n)\big)$$

Properties of Newton optimization method

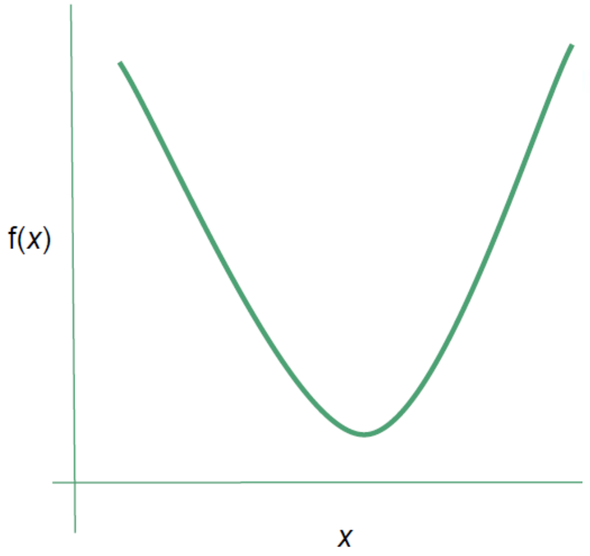
- When f *is a positive definite quadratic function, Newton's method* jumps in a single step to the minimum of the function directly.

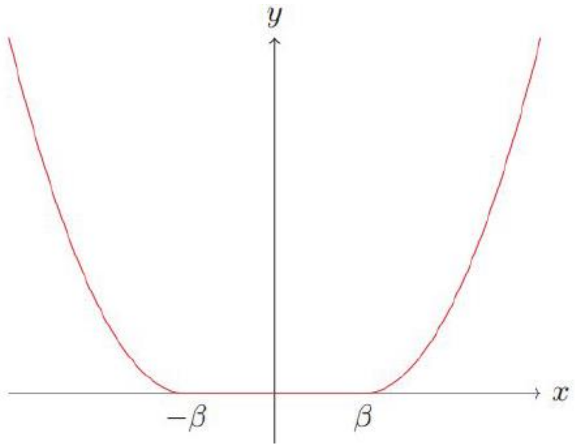
- *Newton's method can* reach the critical point much faster than 1st order gradient descent.

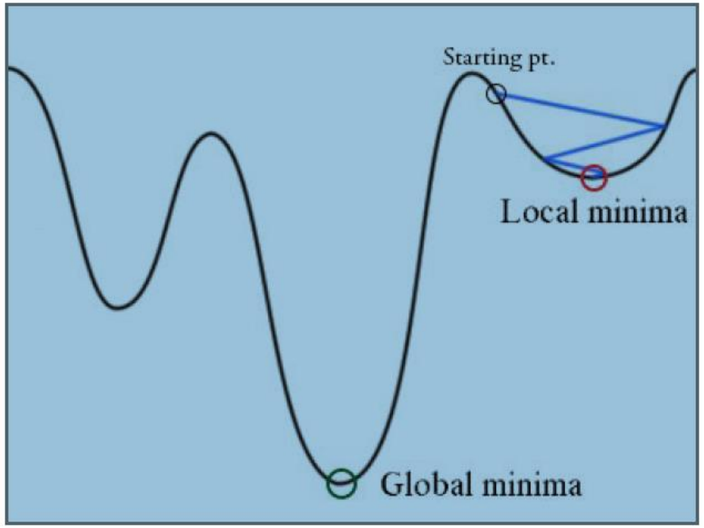
Newton optimization:

$$\Delta\mathbf{x} = -\mathbf{H}\left(f(\mathbf{x}_0)\right)^{-1}\nabla f(\mathbf{x}_0) \qquad \mathbf{x}(n+1) = \mathbf{x}(n) - \eta\mathbf{H}\left(f\left(\mathbf{x}(n)\right)\right)^{-1}\nabla f\left(\mathbf{x}(n)\right)$$

Convex and non-convex functions







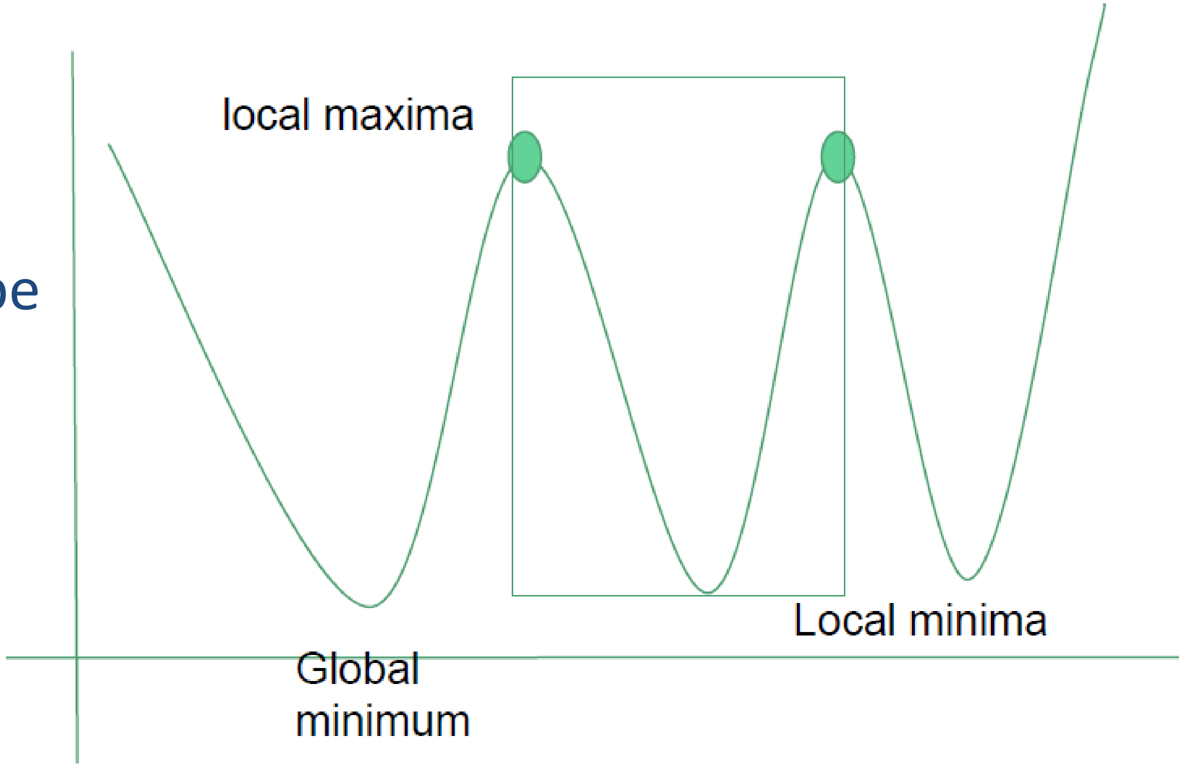
Strongly convex function:
1 local minimum

Non-Strongly convex function: infinity local touching minima with the same values

Non-convex function: multiple non-touching local minima with different values

Local optimization in non-convex case

- Optimization is done locally in a certain domain, where the function is assumed to be convex
- Multiple local optimization is used to find global minimum



Most commonly applied gradient descent methods

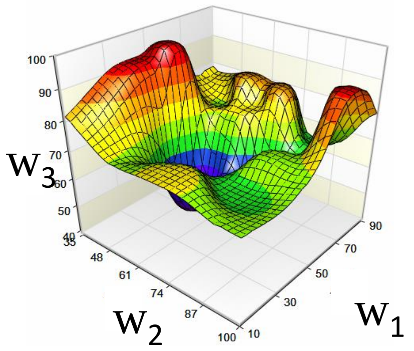
- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

What are we optimizing here?

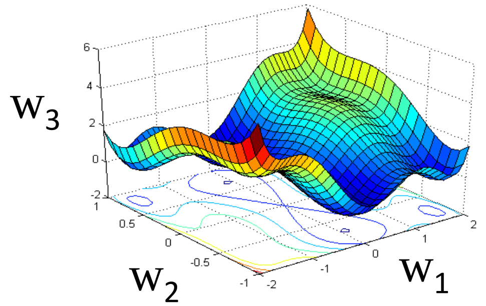
- Cost function in quadratic case for one $\mathbf{x}_i \rightarrow \mathbf{d}_i$ pair:

$$\mathcal{E}_i = \left(\mathbf{d}_i - Net(\mathbf{x}_i, \mathbf{w})\right)^2$$

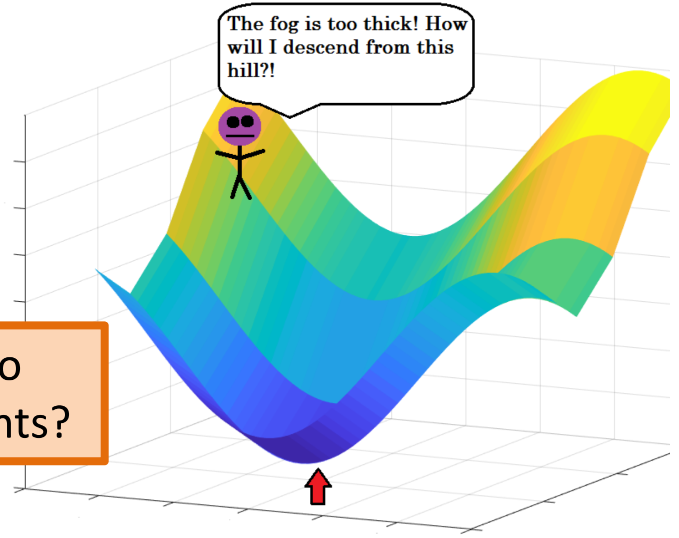
  - Error surface is in the $\mathbf{w}$ space
  - Error surface depends on the $\mathbf{x}_i \rightarrow \mathbf{d}_i$ pair
  - Moreover, we do not see the entire surface, just $\mathcal{E}$ and the gradients $\frac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}}$



Error surface for $\mathbf{x}_i \rightarrow \mathbf{d}_i$



Error surface for $\mathbf{x}_k \rightarrow \mathbf{d}_k$



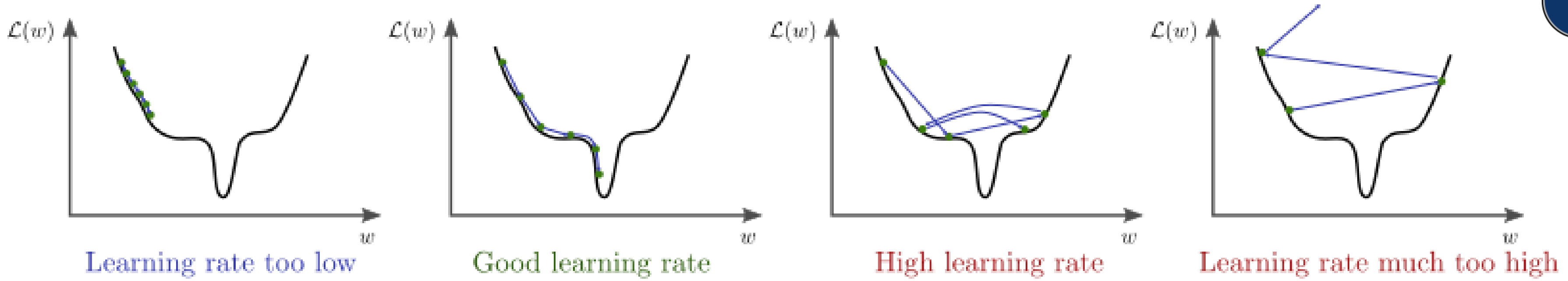
When and how to update the weights?

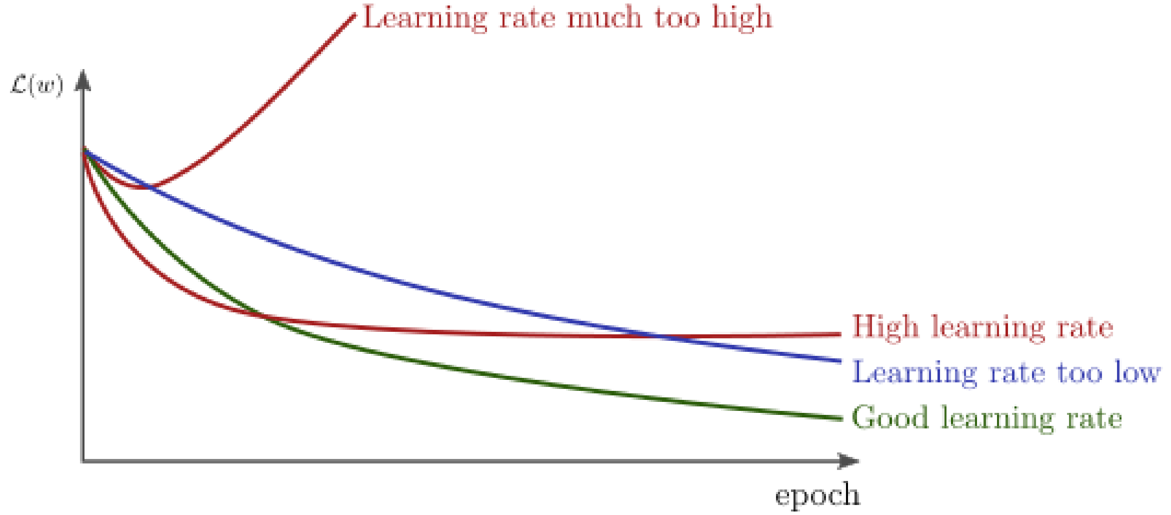
Update strategies

- Single vector update approach (instant update)
  - Weights are updated after each input vector
- Batched update approach
  - All the input vectors are applied
    - this is actually the correct entire error funtion, which is used by the original Gradient Descent Method
  - Updates (Δw_{ij}) are calculated for each vector, and averaged
  - Update is done with the averaged values (Δw_{ij}) after the entire batch is calculated
- Mini batch approach
  - When the number of inputs are very high (10^4-10^6), batch would be ineffective
  - Random selection of m input vectors (m is a few hundred)
  - Updates (Δw_{ij}) are calculated for each vector, and averaged
  - Update is done with the averaged values (Δw_{ij}) after the mini batch is calculated
  - Works efficiently when far away from minimum, but inaccurate close to minimum
  - Requires reducing learning rate

Remember, each approach optimizes different error surfaces!!!

How learning rate effects convergence?





Most commonly applied gradient descent methods

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
  - Nesterov momentum update
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

Stochastic Gradient Descent (SGD) algorithm

- Introduced in 1945
- Gradient Descent method, plus:
  - Applying mini batches
  - Changing the learning rate during the iteration

Learning rate at SGD

- Sufficient conditions to guarantee convergence of SGD:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \qquad \text{and} \qquad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

ϵ is the learning rate, also marked with η sometimes

- In practice:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha \epsilon_\tau \qquad \alpha = \frac{k}{\tau}$$

- After iteration τ *, it is common to leave* ε *constant*

Stochastic Gradient Descent algorithm

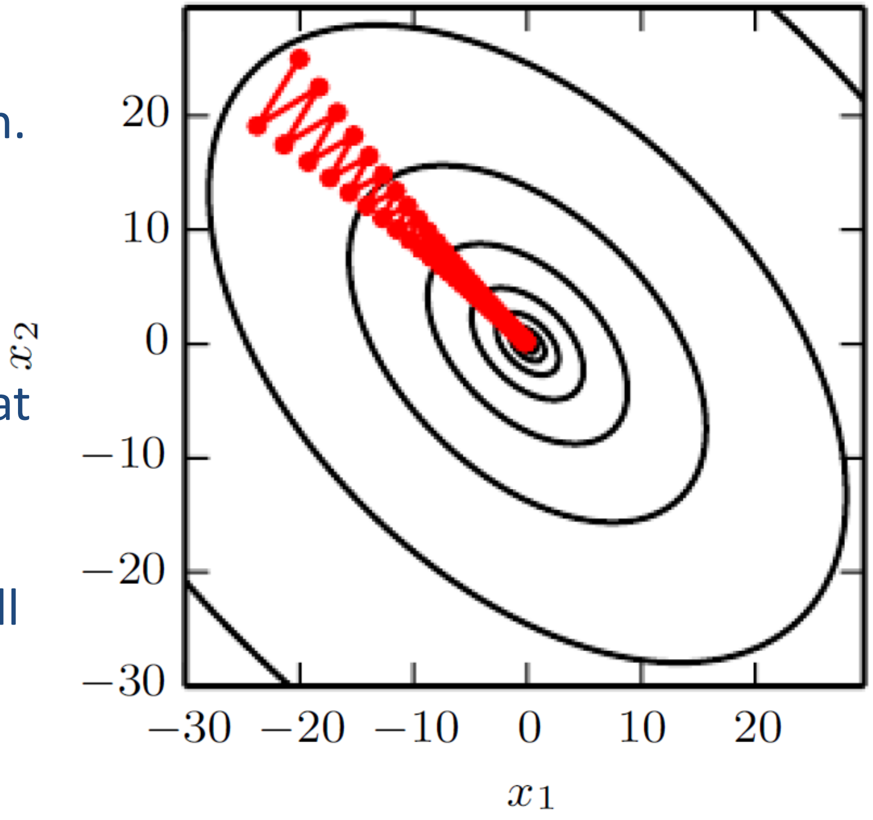
```
Algorithm Stochastic gradient descent (SGD) update at training iteration $k$
Require: Learning rate $\epsilon_k$.
Require: Initial parameter $\boldsymbol{\theta}$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  end while
```

where: L is the cost function

θ is the total set of $w_{i,j}^{(l)}$ (and all other parameters to optimize)

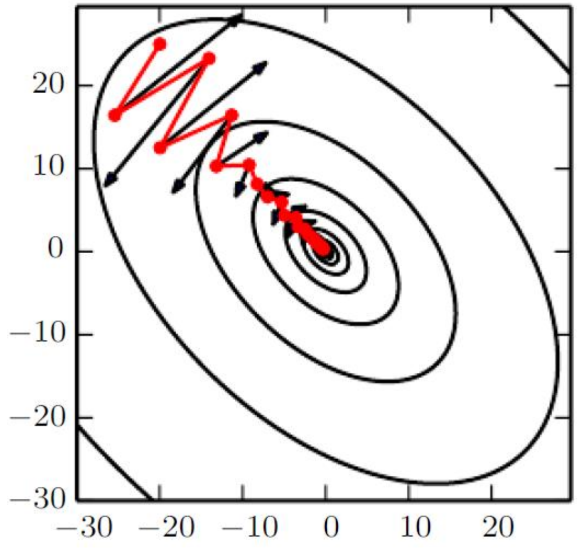
Stochastic Gradient Descent algorithm

- This very elongated quadratic function resembles a long canyon.
- Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature.
- Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration.



Momentum I

- Introduced in 1964
- Physical analogy
- The idea is to simulate a unity weight mass
- It flows through on the surface of the error function
- Follows Newton’s laws of dynamics
- Having *v* velocity
- Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.



Momentum II: velocity considerations

The update rule is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$.
The larger α is relative to ϵ, the more previous gradients affect the current direction.

Terminal velocity is applied when it finds descending gradient permanently:

$$\frac{\epsilon ||\boldsymbol{g}||}{1 - \alpha}$$

Momentum III

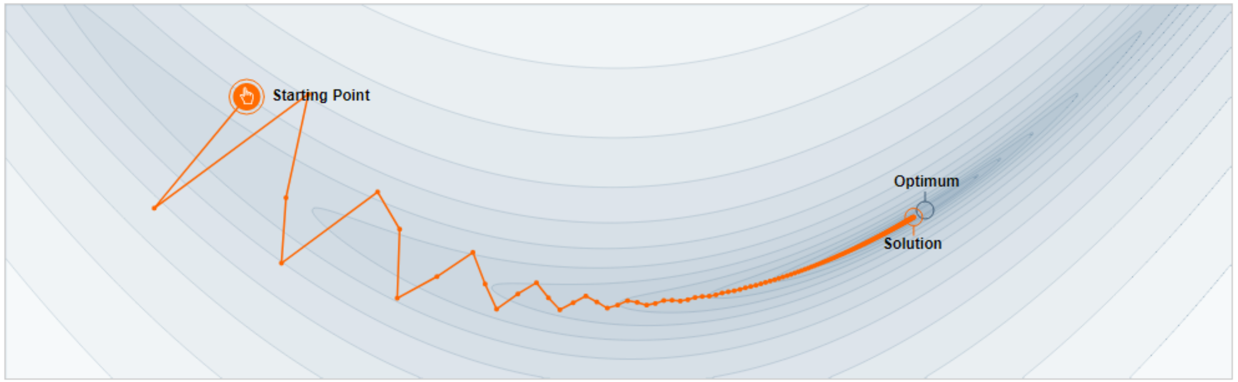
Algorithm Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\epsilon$, momentum parameter $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

Momentum demo

- What does the parameter of the momentum method means, and how to set them?
  - https://distill.pub/2017/momentum/

Why Momentum Really Works



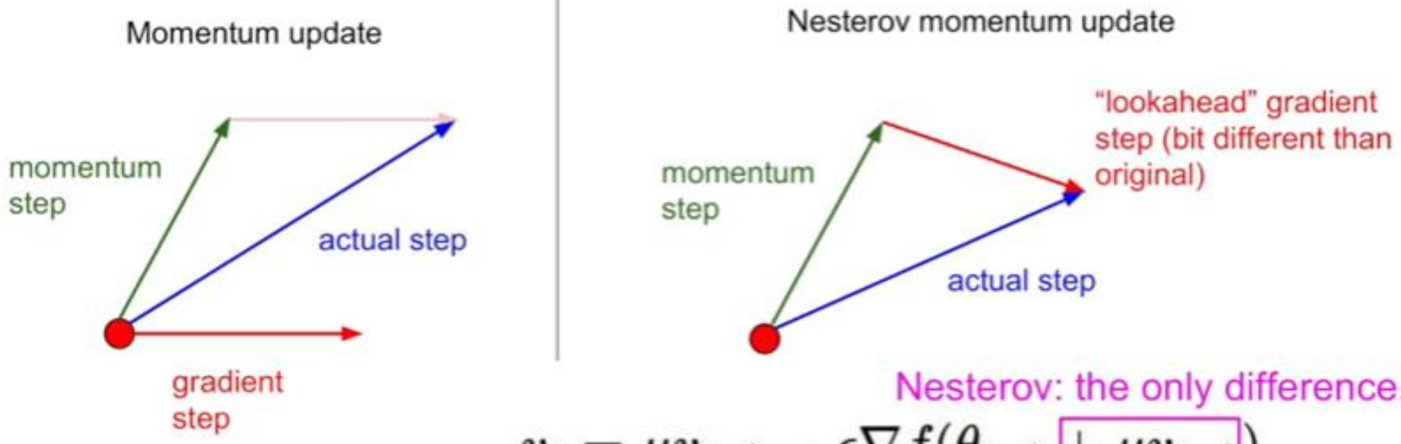
Step-size α = 0.0045

Momentum β = 0.70

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

Nesterov momentum update

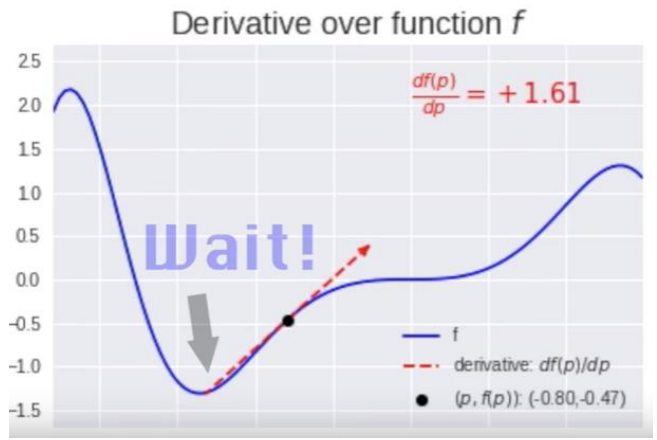
- It calculates the gradient not in the current point, but in the next point, and correct the velocity with the gradient over there (look ahead function)
- It does not runs through a minimum, because if there is a hill behind a minimum, than it starts decreasing the speed in time.



Nesterov: the only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} \boxed{+ \mu v_{t-1}})$$

$$\theta_t = \theta_{t-1} + v_t$$



What if we make the learning rate adaptive as well, not just the velocity?

Most commonly applied gradient descent methods

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
  - Nesterov momentum update
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

AdaGrad algorithm

- The AdaGrad algorithm (2011) individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values

- The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate

- The net effect is greater progress in the more gently sloped directions of parameter space

- AdaGrad performs well for some but not all deep learning models

AdaGrad algorithm

Remembers the entire history evenly

```
**Algorithm** The AdaGrad algorithm
**Require:** Global learning rate $\epsilon$
**Require:** Initial parameter $\boldsymbol{\theta}$
**Require:** Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
  Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  **while** stopping criterion not met **do**
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   (Division and square root applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  **end while**
```

RMSP algorithm

- The RMSProp algorithm (2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average

- In each step AdaGrad reduces the learning rate, therefore after a while it stops entirely!
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure

- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl

RMSP algorithm

The closer parts of the history are counted more strongly.

Algorithm The RMSProp algorithm

```
**Require:** Global learning rate $\epsilon$, decay rate $\rho$.
**Require:** Initial parameter $\boldsymbol{\theta}$
**Require:** Small constant $\delta$, usually $10^{-6}$, used to stabilize division by small
    numbers.
    Initialize accumulation variables $\boldsymbol{r} = 0$
    **while** stopping criterion not met **do**
        Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
        corresponding targets $\boldsymbol{y}^{(i)}$.
        Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
        Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
        Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$.   ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
        Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
    **end while**
```

ADAM algorithm (2014)

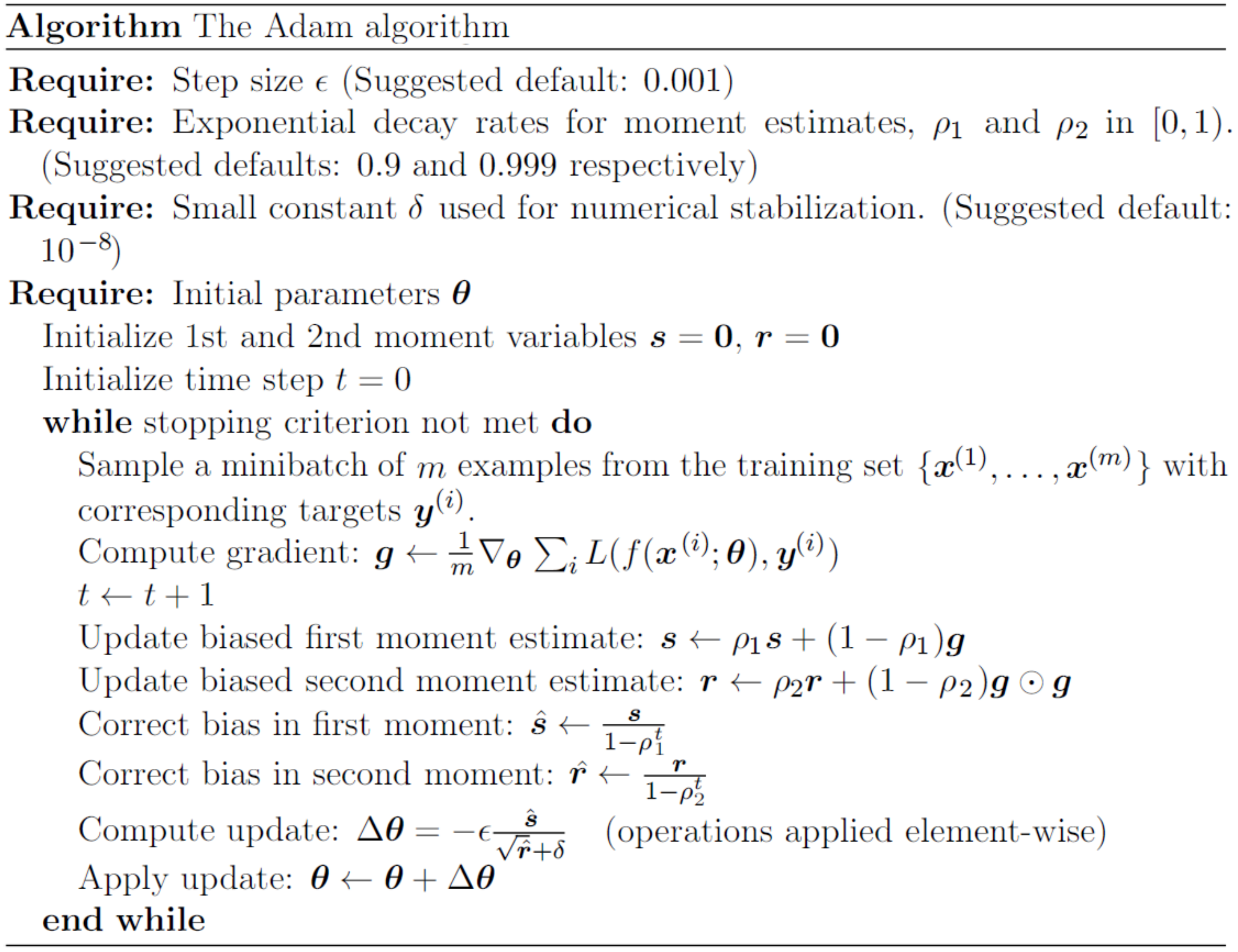
- The name “Adam” derives from the phrase “adaptive moments.”
- In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions.
- in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient.
- Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin

ADAM algorithm

s estimates the gradient from the history (moment)

r estimates the curvature of the gradient

Booth of them are biased to reduce anomalies at the initialization

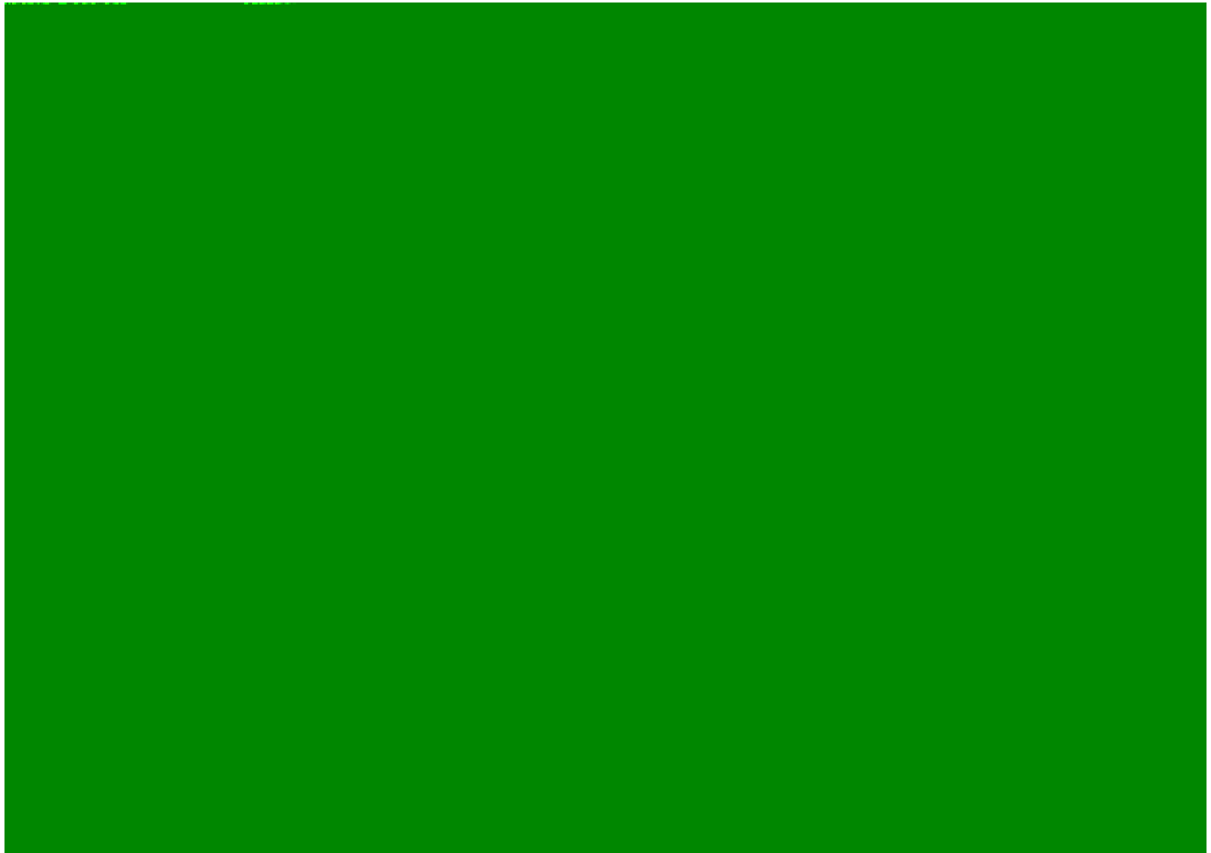


Algorithm The Adam algorithm

```
Require: Step size $\epsilon$ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, $\rho_1$ and $\rho_2$ in $[0, 1)$.
    (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant $\delta$ used for numerical stabilization. (Suggested default:
    $10^{-8}$)
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with
    corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$   (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
  end while
```

Video comparing adaptive and non-adaptive methods

- Three optimizer types are compared:
  - SGD
  - Momentum types
    - Momentum
    - Nesterov AG
  - Adaptív
    - AdaGrad
    - AdaDelta
    - RmsProp
- Adaptive ones are the fastest
- SGD is very slow (stucked into saddle point)
- https://www.youtube.com/watch?v=nhqo0u1a6fw&t=306s



Most commonly applied gradient descent methods

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
- Algorithms with adaptive learning rate
  - AdaGrad algorithm
  - RMSProp algorithm
  - ADAM algorithm
- 2nd order algorithm
  - Newton algorithm

Newton's algorithm

Algorithm Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$.

```
**Require:** Initial parameter $\boldsymbol{\theta}_0$
**Require:** Training set of $m$ examples
  **while** stopping criterion not met **do**
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}^2 \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  **end while**
```

Newton’s algorithm

- Typically not used, due to the computational complexity
- Parameter space much higher than first order (where it is already very high)

Back propagation

- We have seen last time how to calculate the gradient in a multilayer fully connected network using back propagation
  - The introduced method was based on gradient descent method
- However, being able to calculate gradient, we might select any of the above methods, which leads to orders of magnitude faster convergence