Lecture 06:

Optimization

Markus Hohle

University California, Berkeley

Machine Learning Algorithms

MSSE 277B, 3 Units

Course Map

Lecture 1: Course Overview and Introduction to Machine Learning

classic ML tools & algorithms

Lecture 2: Bayesian Methods in Machine Learning

Lecture 3: Dimensionality Reduction: Principal Component Analysis

Lecture 4: Linear and Non-linear Regression and Classification

Lecture 5: Unsupervised Learning: K-Means, GMM, Trees

Lecture 6: Adaptive Learning and Gradient Descent Optimization Algorithms

ANNs/AI/Deep Learning

Lecture 7: Introduction to Artificial Neural Networks - The Perceptron

Lecture 8: Introduction to Artificial Neural Networks - Building Multiple Dense Layers

Lecture 9: Convolutional Neural Networks (CNNs) - Part I

Lecture 10: CNNs - Part II

Lecture 11: Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs)

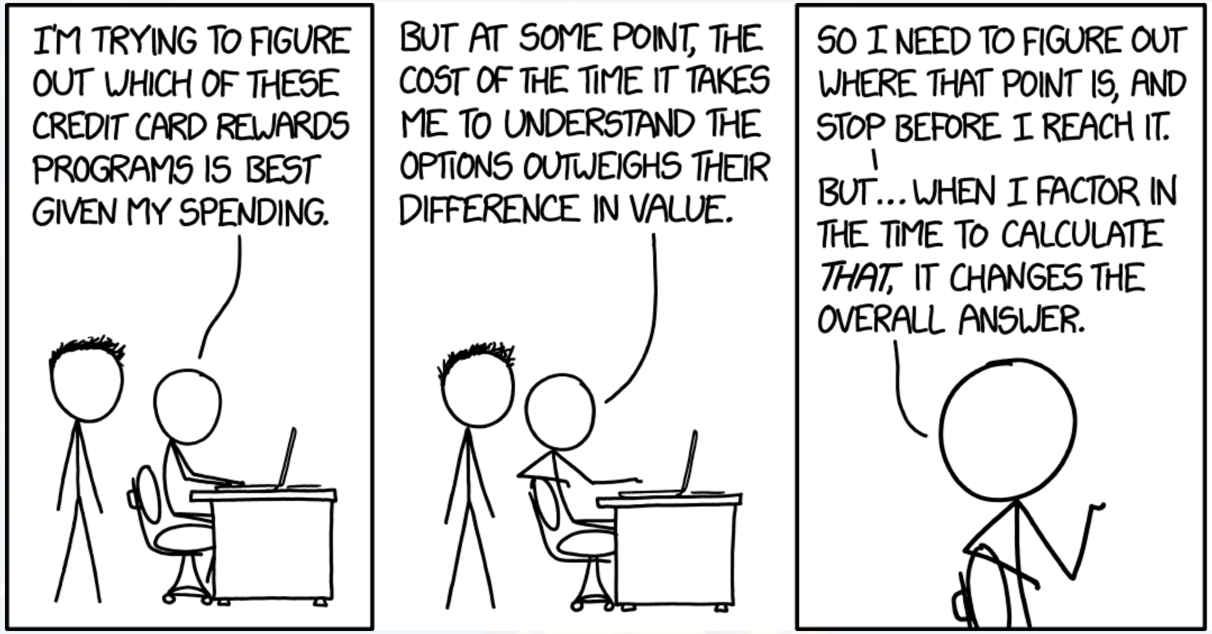
Lecture 12: Combining LSTMs and CNNs

Lecture 13: Running Models on GPUs and Parallel Processing

Lecture 14: Project Presentations

Lecture 15: Transformer

Lecture 16: GNN

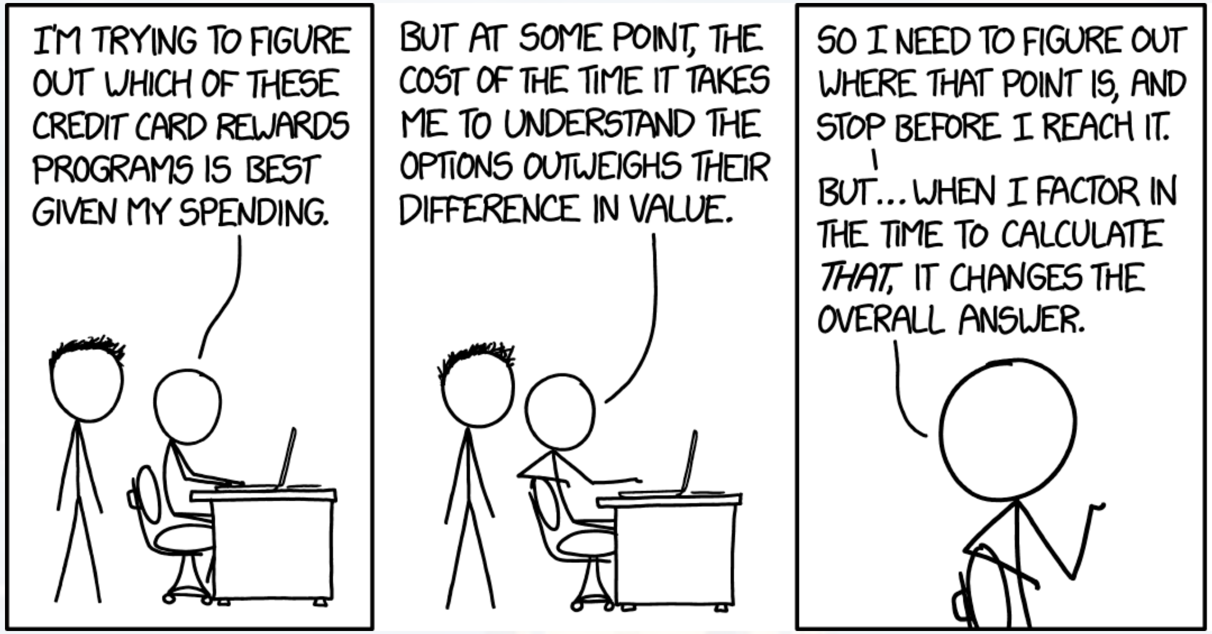


Outline

- The Problem

- Gradient Descent

- Vanilla
- Learning Rate Schedule
- Momentum
- L1 and L2
- More Finetuning



Outline

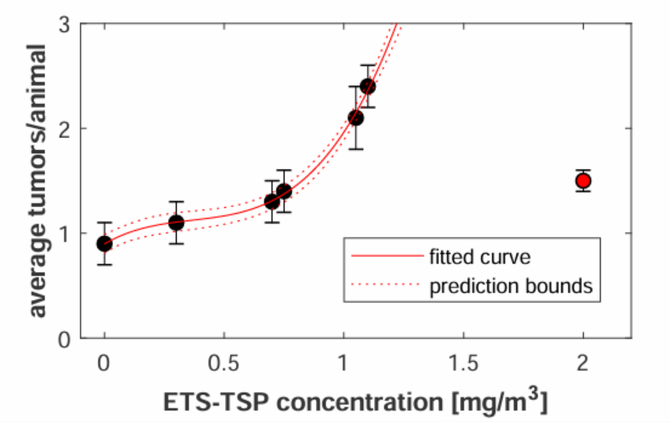
- The Problem

- Gradient Descent
  - Vanilla
  - Learning Rate Schedule
  - Momentum
  - L1 and L2
  - More Finetuning

Optimization:

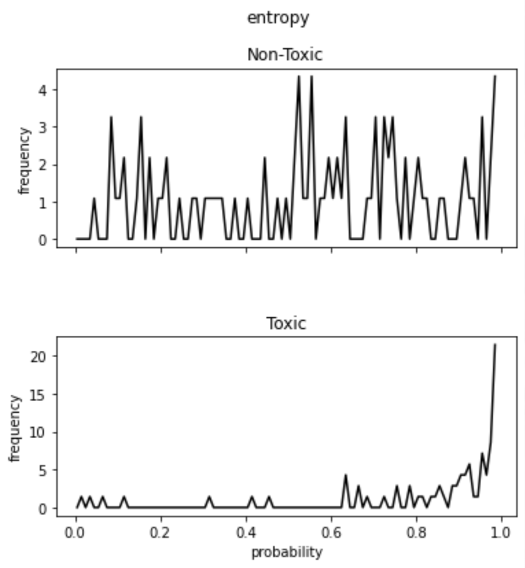
Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

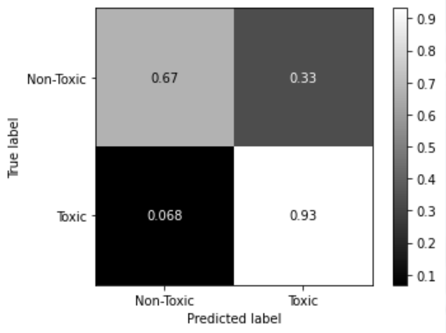
regression, e. g. curve fitting



minimize: $$\chi^2_{red} = \frac{1}{N - p - 1} \sum_{i=1}^{N} \frac{(\hat{y}(model)_i - y_i)^2}{\sigma_i^2}$$

classification





maximize: accuracy

minimize: cross entropy $$S = -\sum_i p(true)_i \cdot \ln p(model)_i$$

Optimization: The Problem

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

Often, the extreme of the objective function is subject to **constrains**

cross entropy $$S = -\sum_i p(true)_i \cdot \ln p(model)_i$$ constrain: $$\sum_i p_i = 1$$

→ Lagrangian Multipliers and variational calculus

→ mathematically: *Free Energy like term = Energy like term – Entropy term*

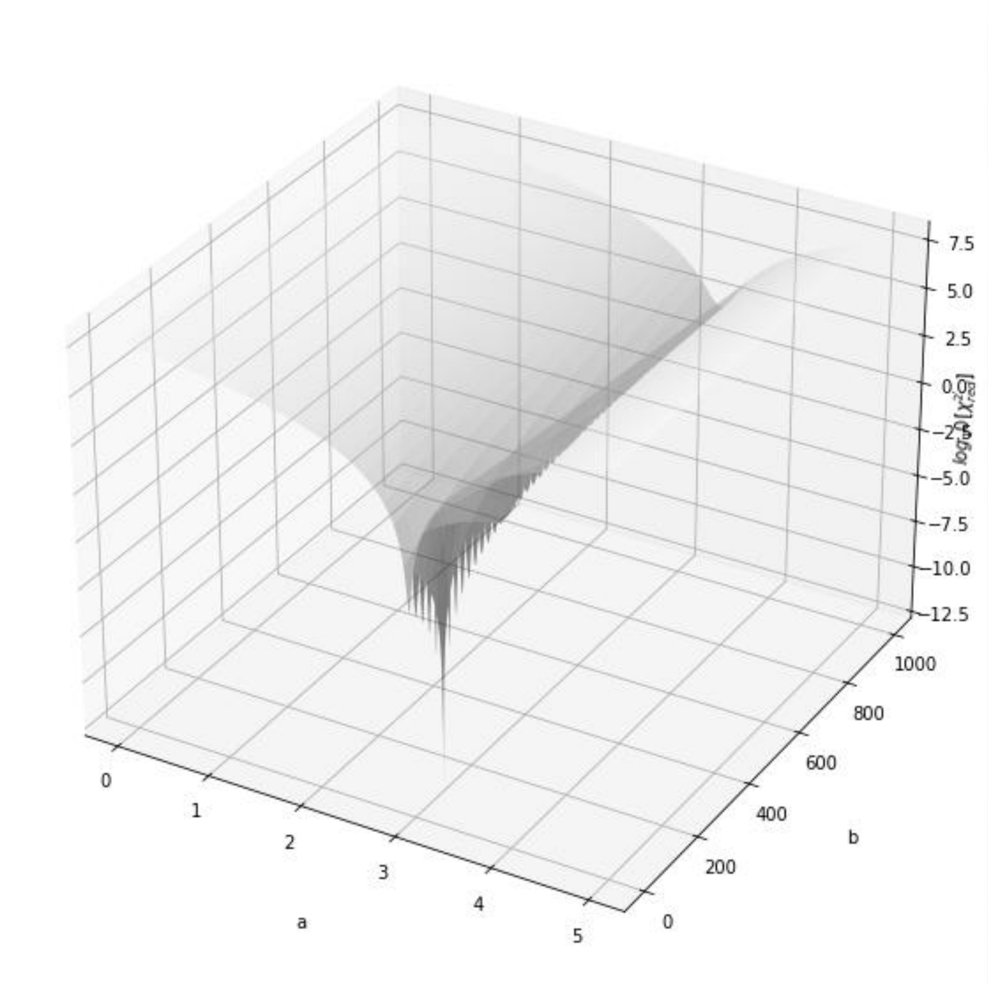
examples:
- **E**vidence **L**ower **Bo**und
- Lasso method (linear regression)
- actual energy → Boltzmann distribution

etc

Optimization:

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

These functions are very complicated, not analytical (= no mathematical equation) at all



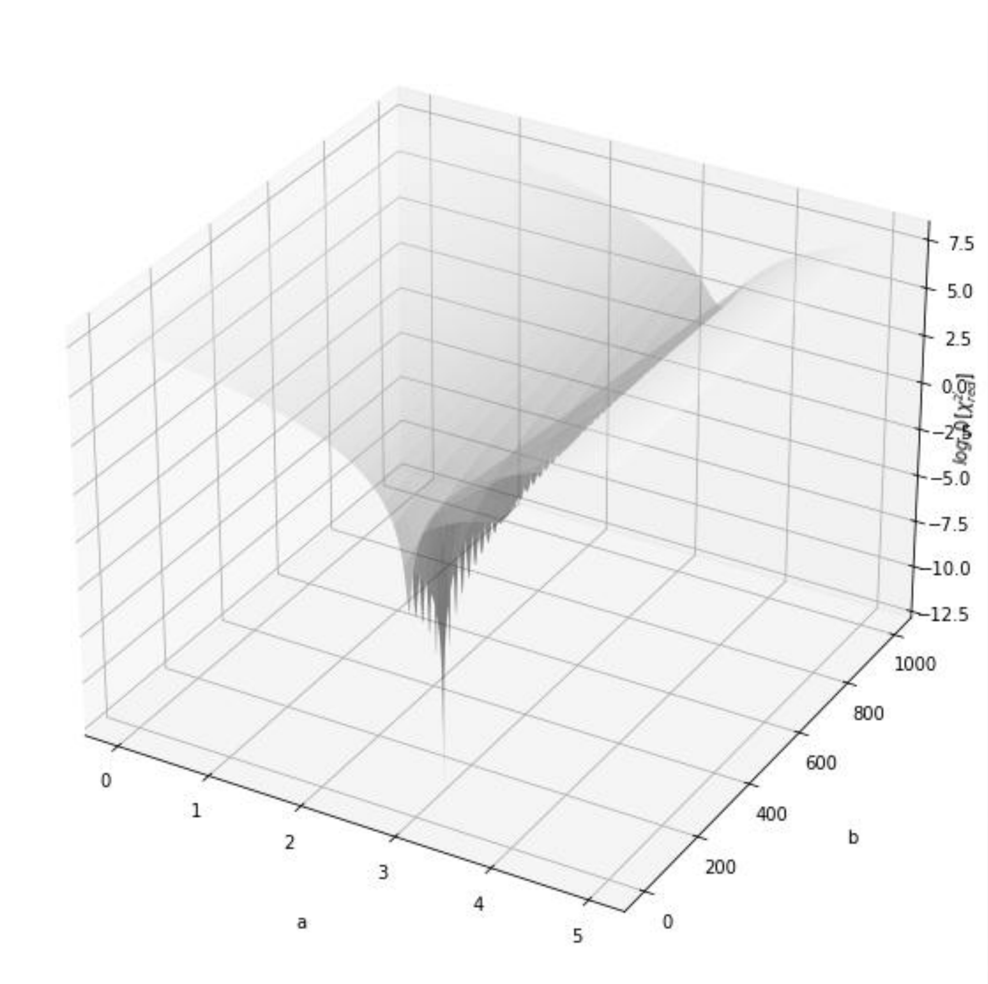
<u>two most common approaches:</u>

- gradient descent
- simulated annealing

Optimization: The Problem

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

These functions are very complicated, not analytical (= no mathematical equation) at all



<u>two most common approaches:</u>

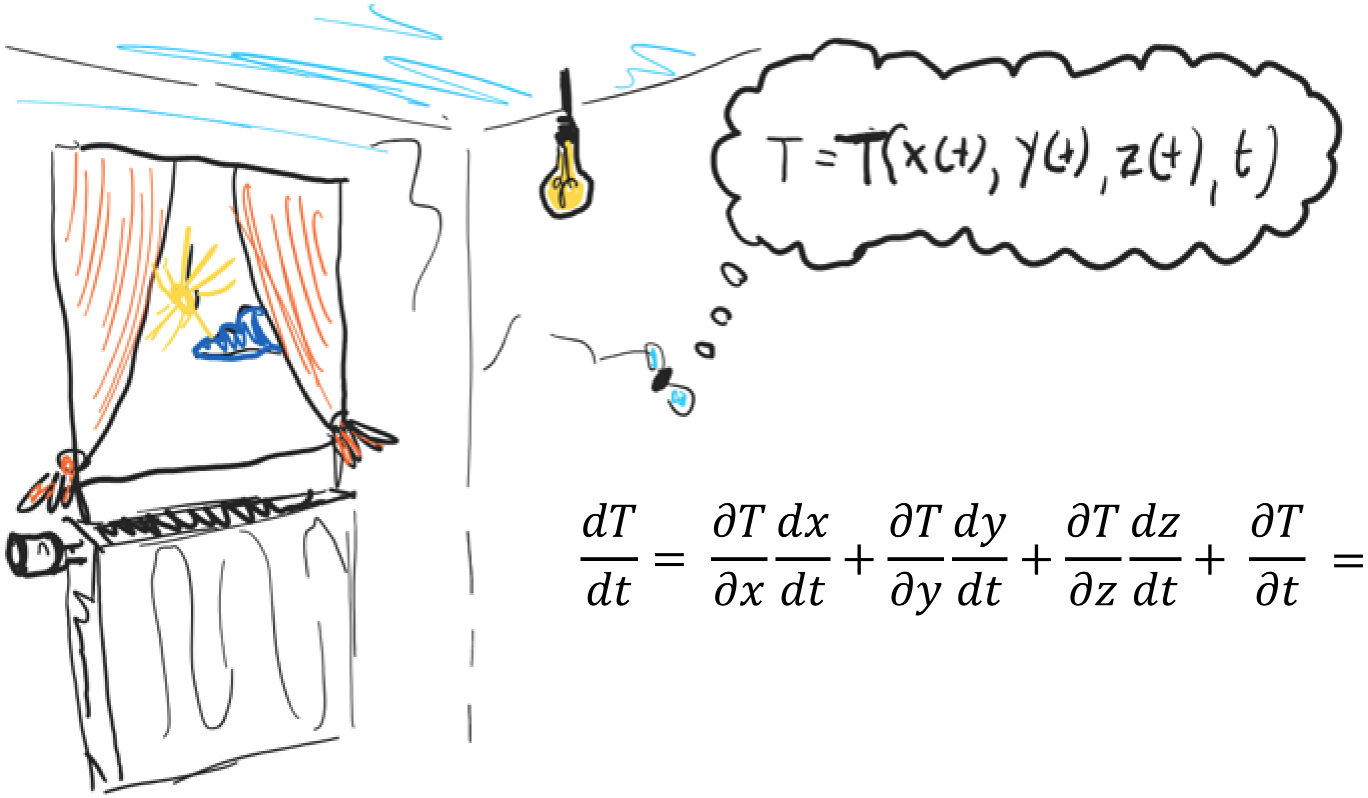
- **gradient descent**
- simulated annealing

Optimization: The Problem

gradient descent

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)
→ extreme of an objective function

temperature profile in space and time



T: temperature

$$\frac{dT}{dt} = \frac{\partial T}{\partial x}\frac{dx}{dt} + \frac{\partial T}{\partial y}\frac{dy}{dt} + \frac{\partial T}{\partial z}\frac{dz}{dt} + \frac{\partial T}{\partial t} = \begin{pmatrix} \frac{\partial T}{\partial x} \\ \frac{\partial T}{\partial y} \\ \frac{\partial T}{\partial z} \end{pmatrix} \circ \begin{pmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{pmatrix} + \frac{\partial T}{\partial t}$$

$$= grad(T) \circ \vec{v}$$

If *T* doesn’t change with time!

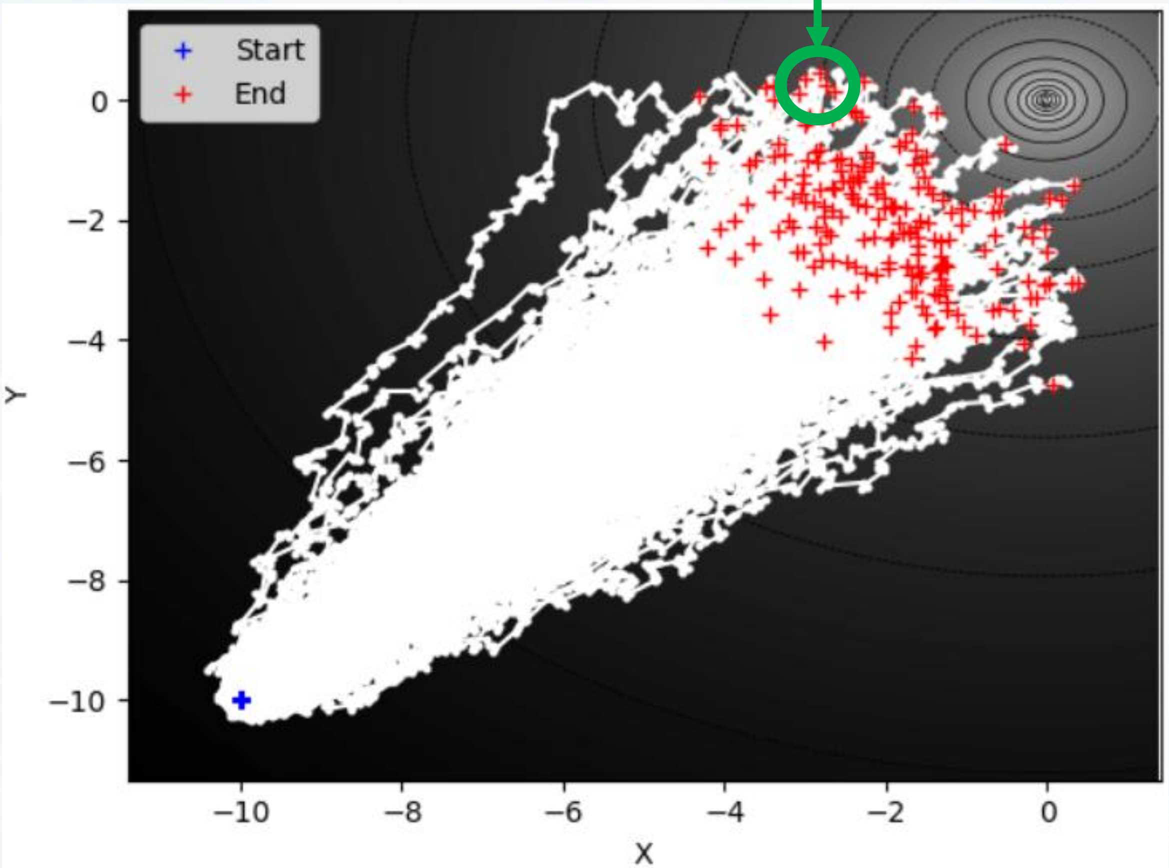
Optimization: The Problem

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)
→ extreme of an objective function

gradient descent

concentration profile in space and time

E. Coli



c: concentration

$$\frac{dc}{dt} = \frac{\partial c}{\partial x}\frac{dx}{dt} + \frac{\partial c}{\partial y}\frac{dy}{dt} + \frac{\partial c}{\partial z}\frac{dz}{dt} + \frac{\partial c}{\partial t}$$

$$= \begin{pmatrix} \frac{\partial c}{\partial x} \\ \frac{\partial c}{\partial y} \\ \frac{\partial c}{\partial z} \end{pmatrix} \circ \begin{pmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{pmatrix} + \frac{\partial c}{\partial t}$$

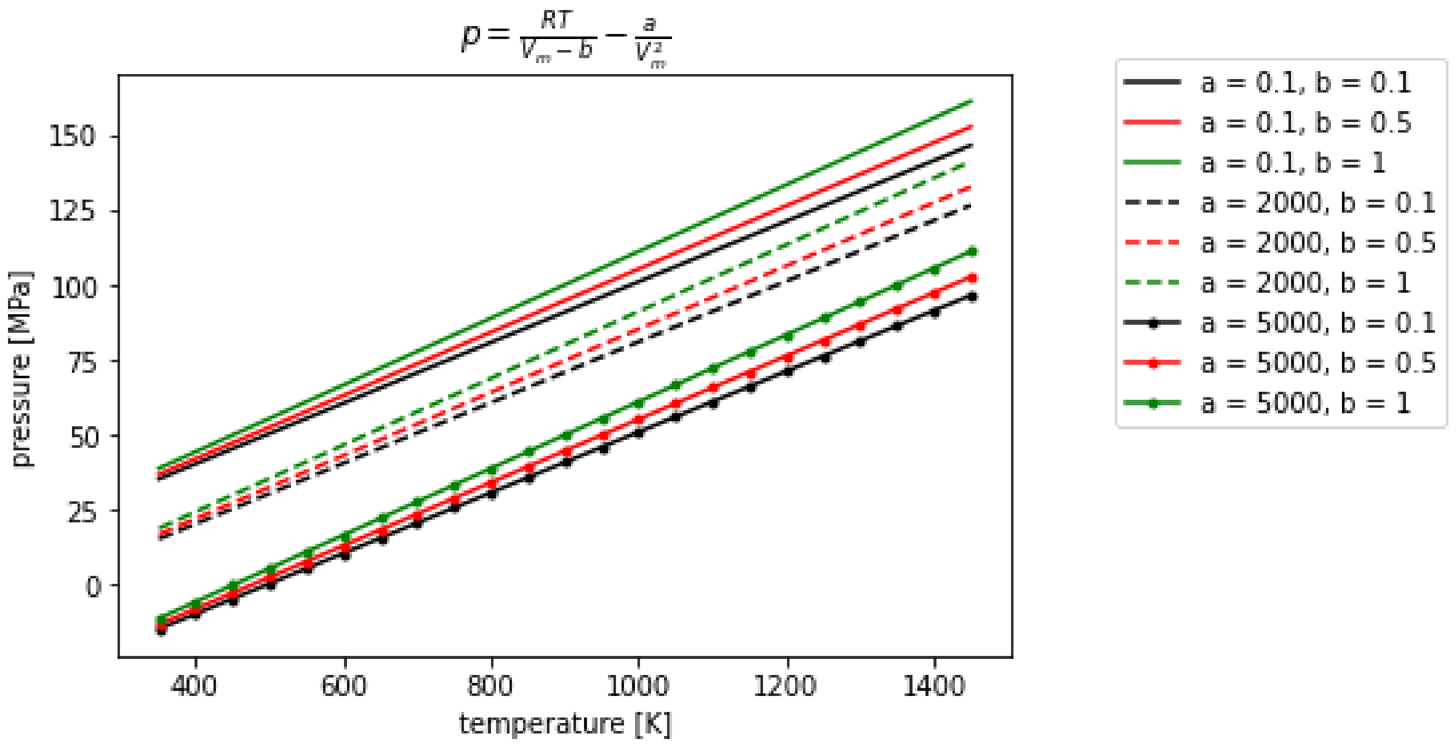
$$= grad(c) \circ \vec{v}$$

If *c* doesn’t change with time!

Optimization:

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

gradient descent



finding ***a*** and ***b*** of
a van-der-Waals gas

if critical points are not accessible
→ fitting curve, finding ***a*** and ***b***

Optimization: The Problem

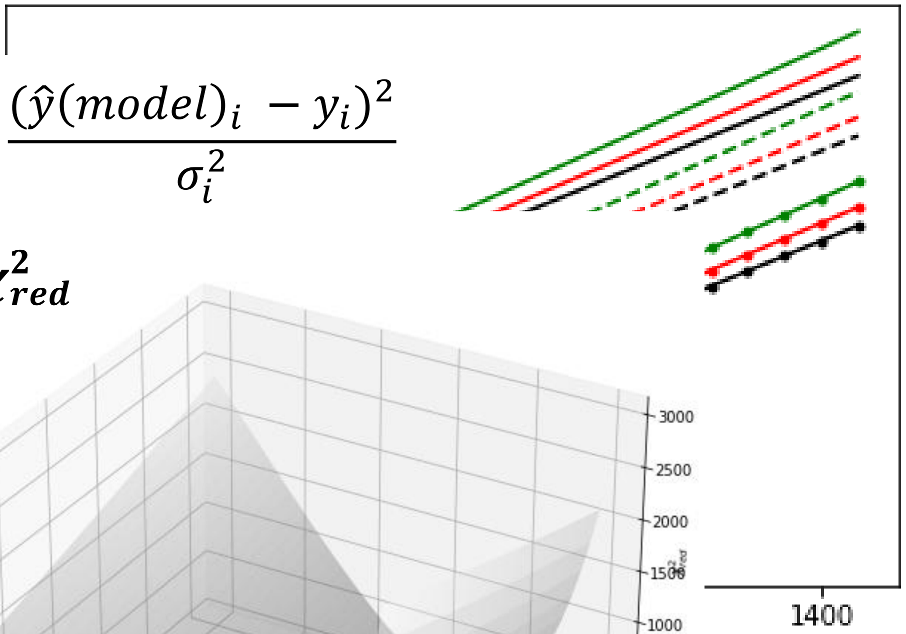
Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

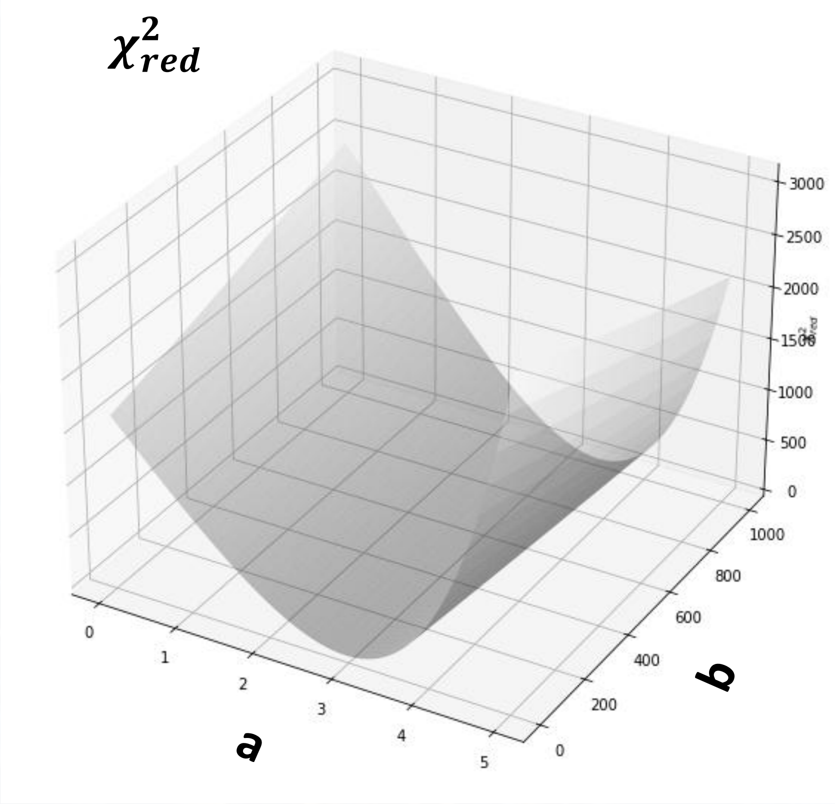
gradient descent

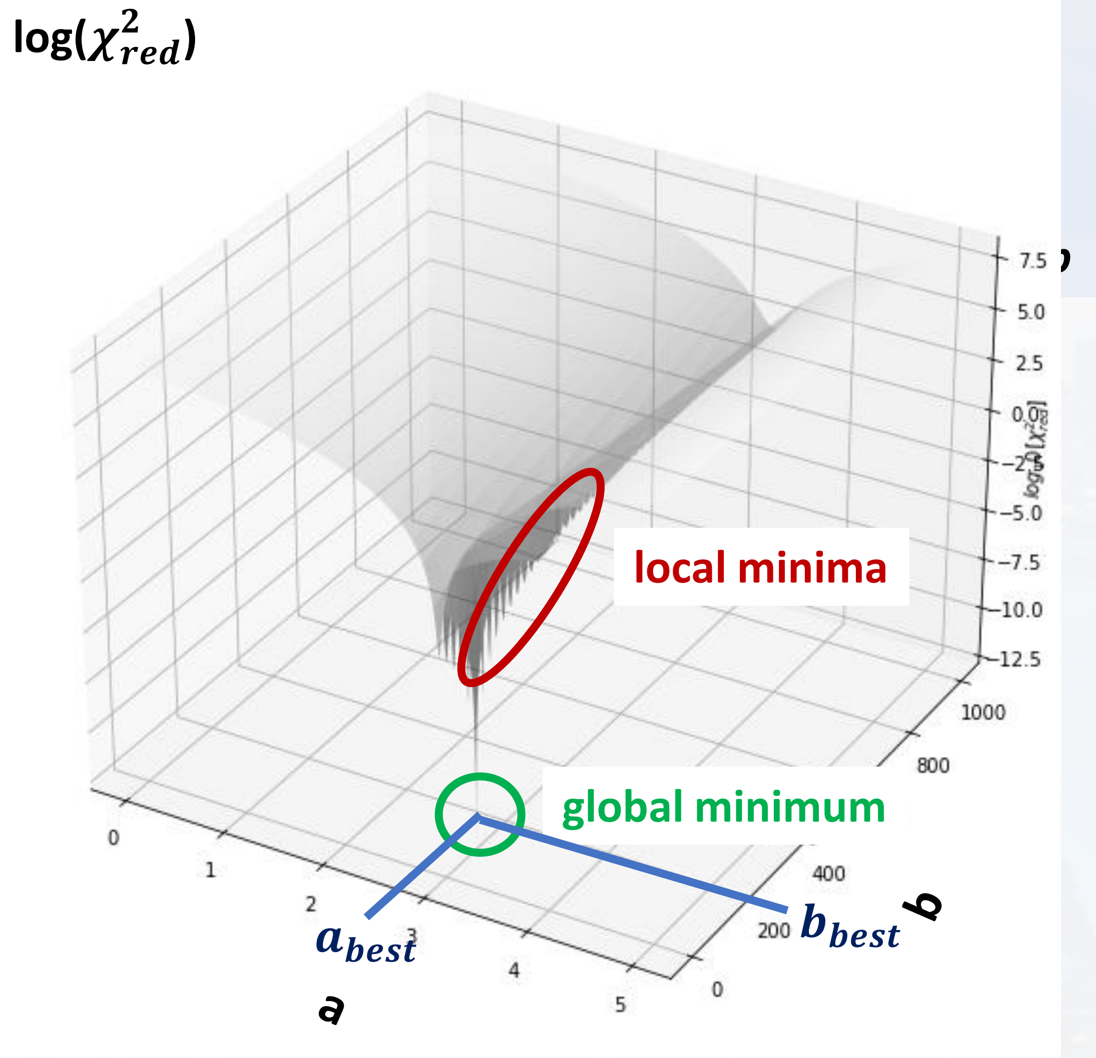
$$\chi^2_{red} = \frac{1}{N - p - 1} \sum_{i=1}^{N} \frac{(\hat{y}(model)_i - y_i)^2}{\sigma_i^2}$$

finding ***a*** and ***b*** of

$p = \frac{RT}{V_m - b} - \frac{a}{V_m^2}$



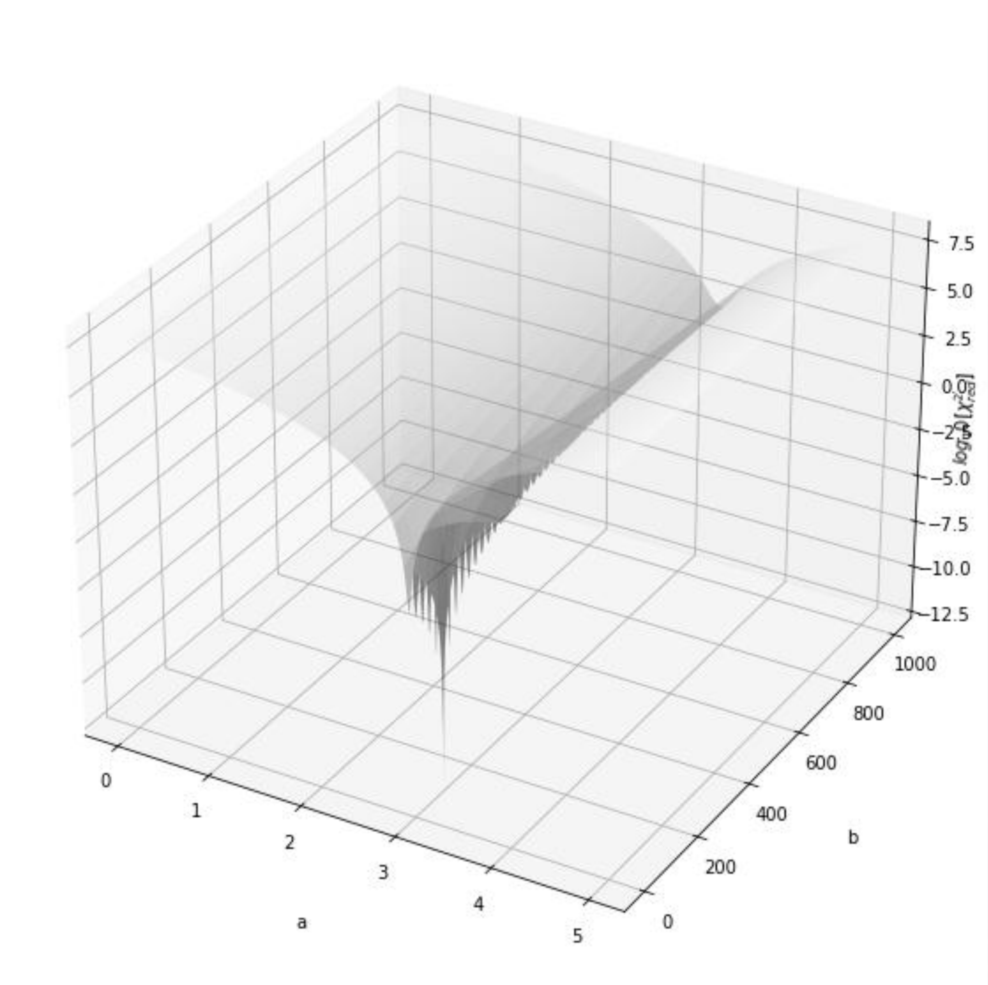




Optimization:

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

These functions are very complicated, not analytical (= no mathematical equation) at all



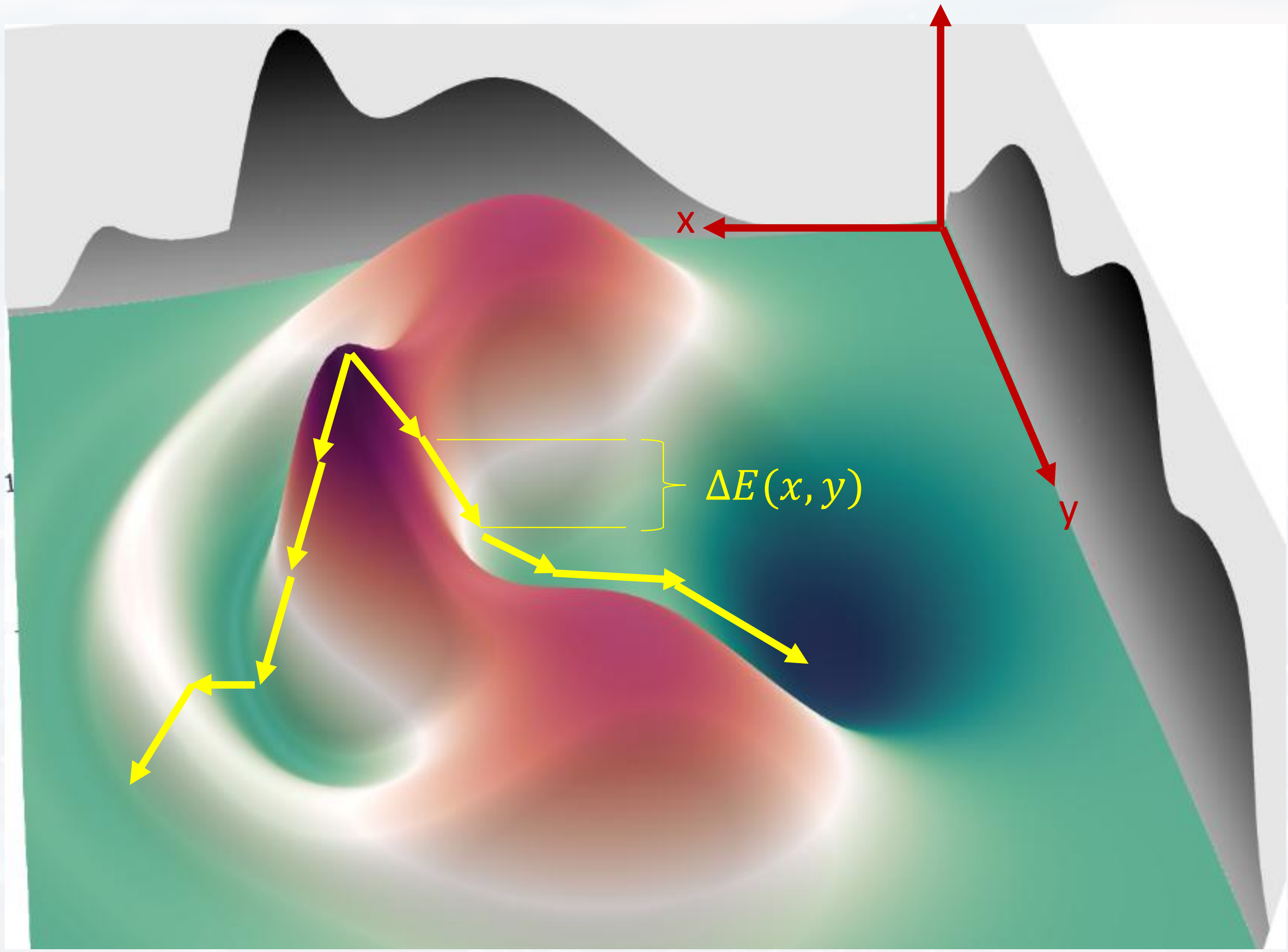
<u>two most common approaches:</u>

- gradient descent
- **simulated annealing**

Optimization:

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

simulated annealing



If $\Delta E(x,y)$ is **negative**:
→ **always move**
(a ball always rolls down the hill)

If $\Delta E(x,y)$ is **positive**:
→ calculate the **probability to move**
→ leaves some chance to escape local minimum

T: temperature

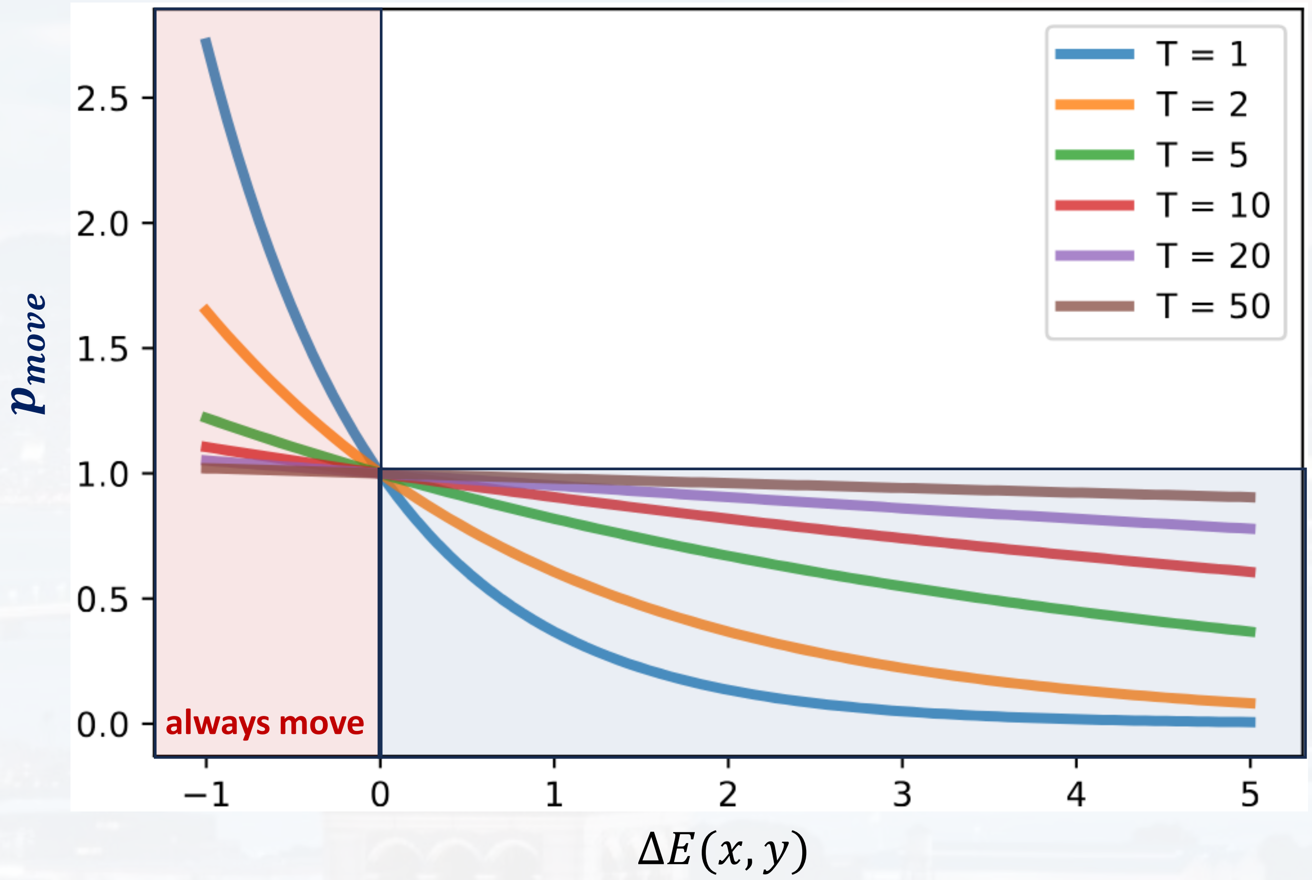
Boltzmann factor

$$p_{move} \sim \exp\left[-\frac{\Delta E(x,y)}{T}\right]$$

Optimization: The Problem

Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

simulated annealing



If $\Delta E(x,y)$ is **negative**:
→ **always move**
(a ball always rolls down the hill)

If $\Delta E(x,y)$ is **positive**:
→ calculate the **probability to move**
→ leaves some chance to escape local minimum

T: temperature

Boltzmann factor

$$p_{move} \sim \exp\left[-\frac{\Delta E(x,y)}{T}\right]$$

slowly reducing T → making larger jumps ($\Delta E(x,y)$) less likely over time

Optimization: The Problem

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

simulated annealing

Metropolis (Chem 273):

1) suggest a random move Δx and Δy

2) calculate $\Delta E(x, y)$ based on Δx and Δy

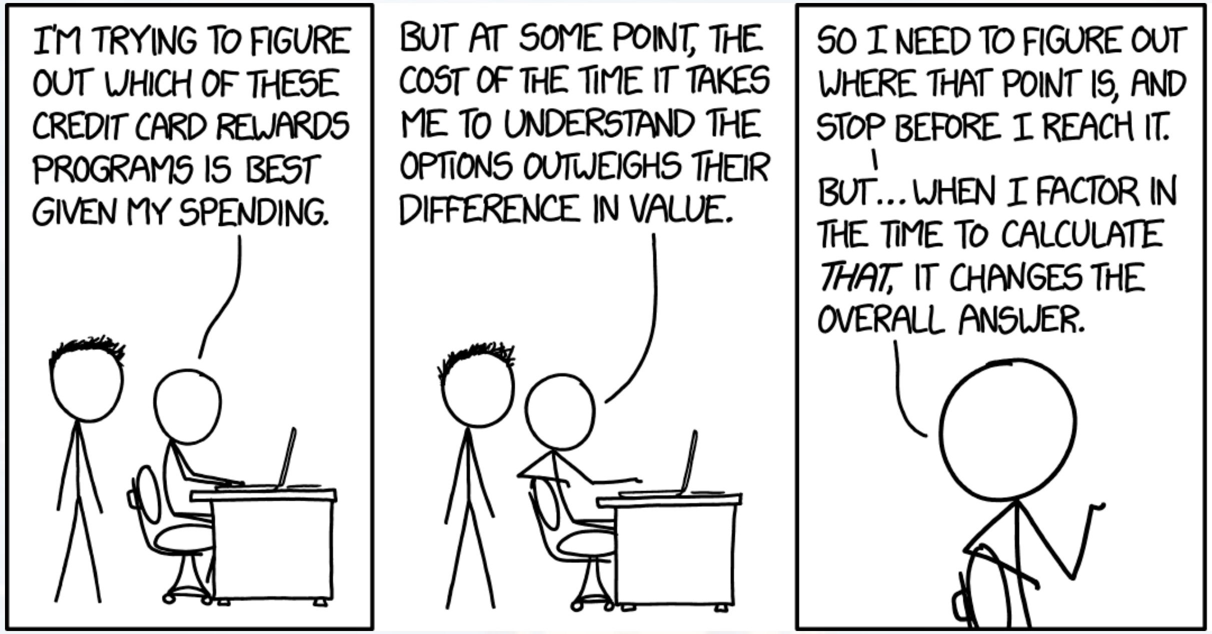
3) move or not:

a) move if $\Delta E(x, y) < 0$

b) if $\Delta E(x, y) > 0$

- draw a **random number ρ** from a **uniform distribution** in the interval $(\mathbf{0}, \mathbf{1})$
- move if $\rho < \exp\left[-\frac{\Delta E(x,y)}{T}\right]$

4) reduce T and repeat



Outline

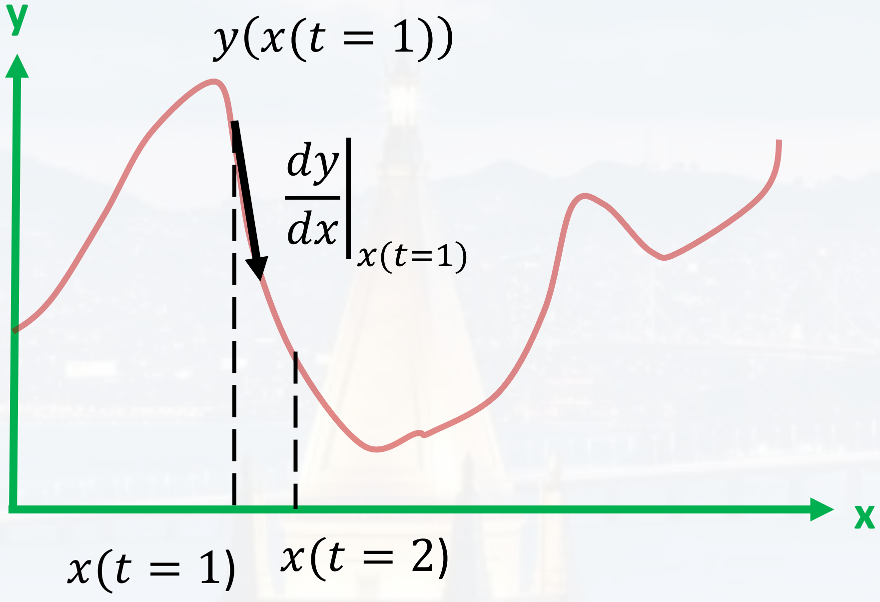
- The Problem
- **Gradient Descent**
  - Vanilla
  - Learning Rate Schedule
  - Momentum
  - L1 and L2
  - More Finetuning

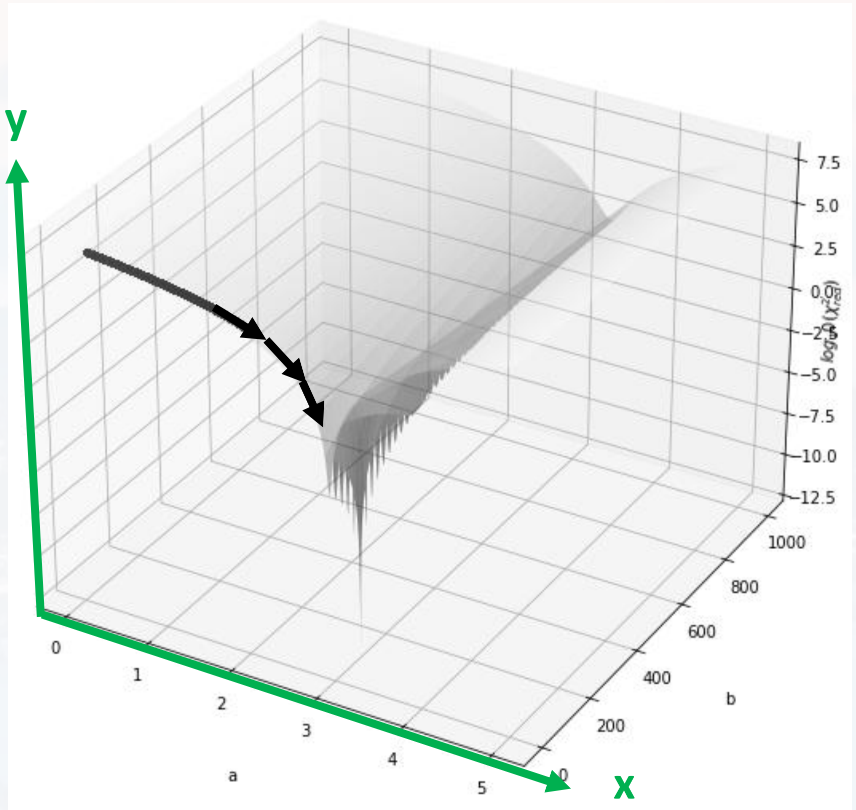
main application: **ANN!**

Optimization: Gradient Descent

Vanilla

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x)\ -y(x_0-\Delta x)}{2\Delta x}$$



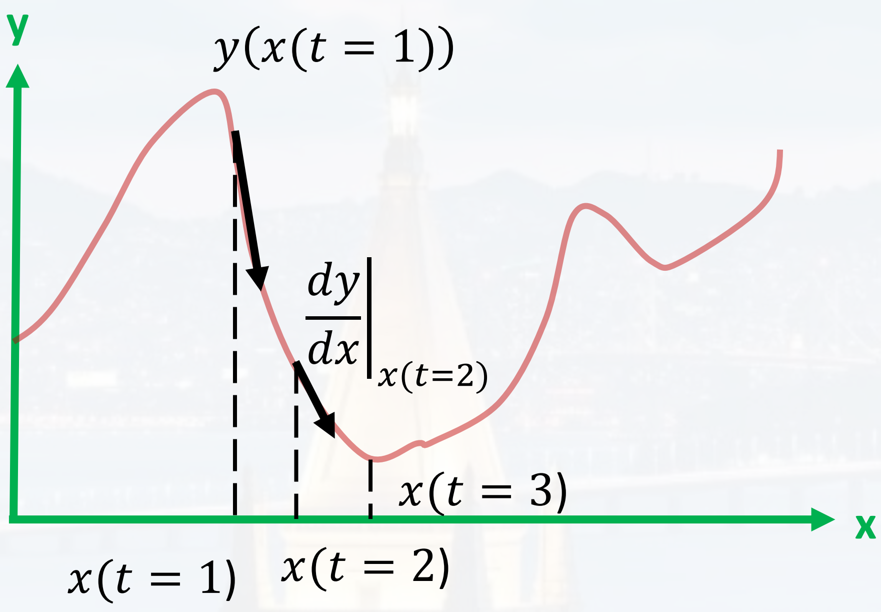


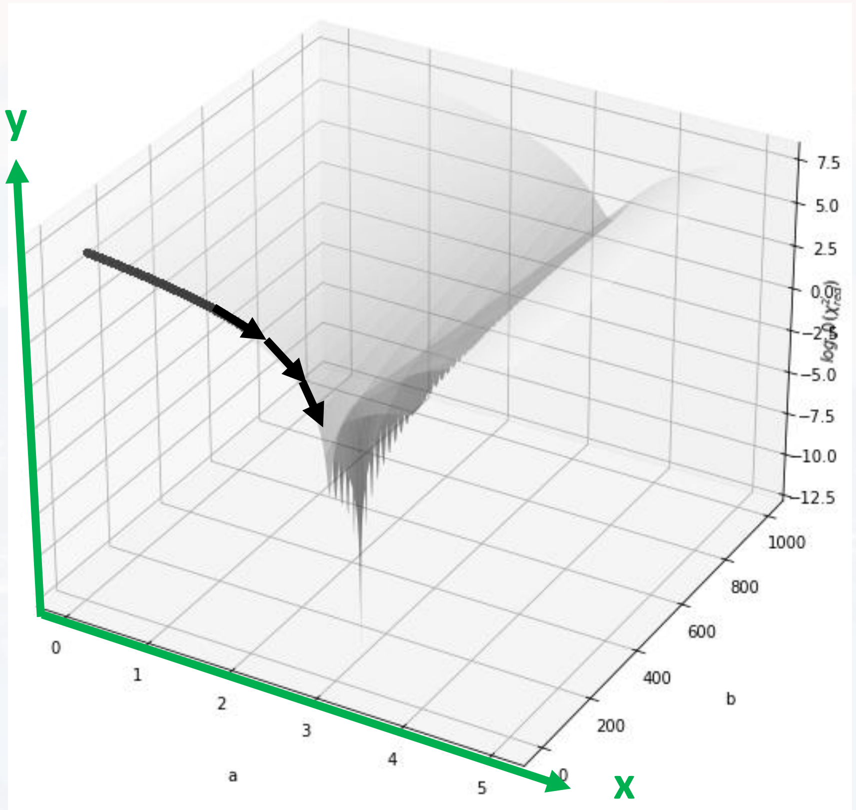
$$x(t=2) = x(t=1)\ -\ \varepsilon\left.\frac{dy}{dx}\right|_{x(t=1)}$$

$\varepsilon > 0$

Vanilla

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$





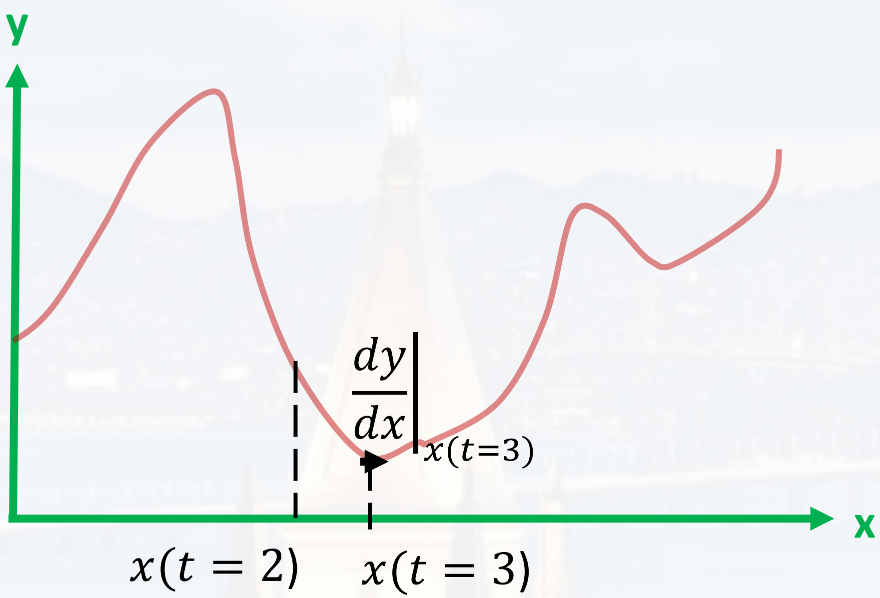
$$x(t=3) = x(t=2) - \varepsilon \left.\frac{dy}{dx}\right|_{x(t=2)}$$

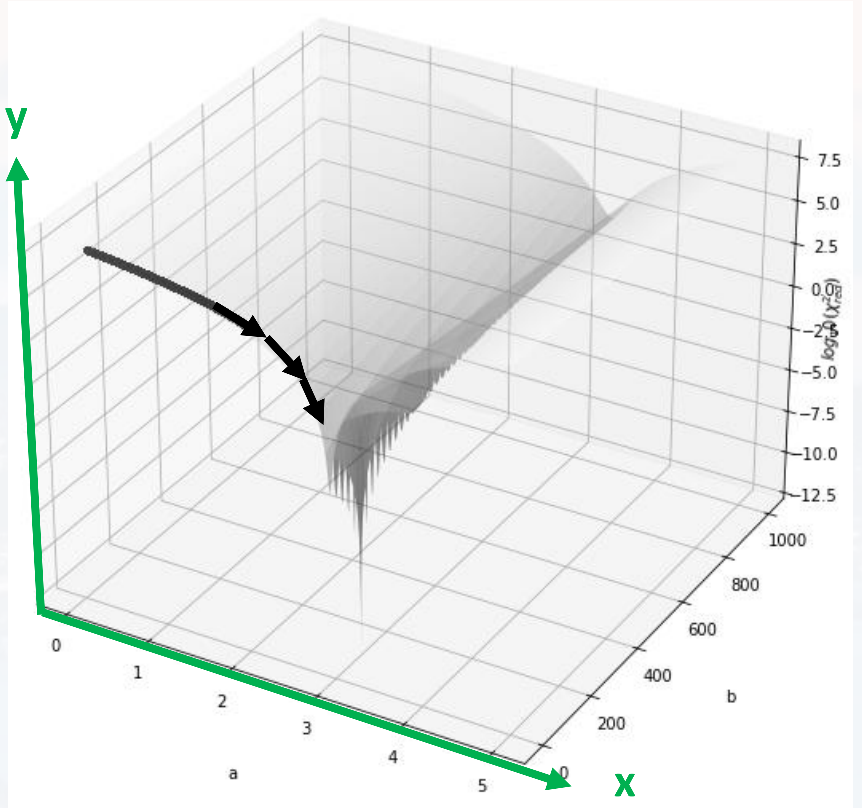
ε > 0

Optimization: Gradient Descent

Vanilla

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$



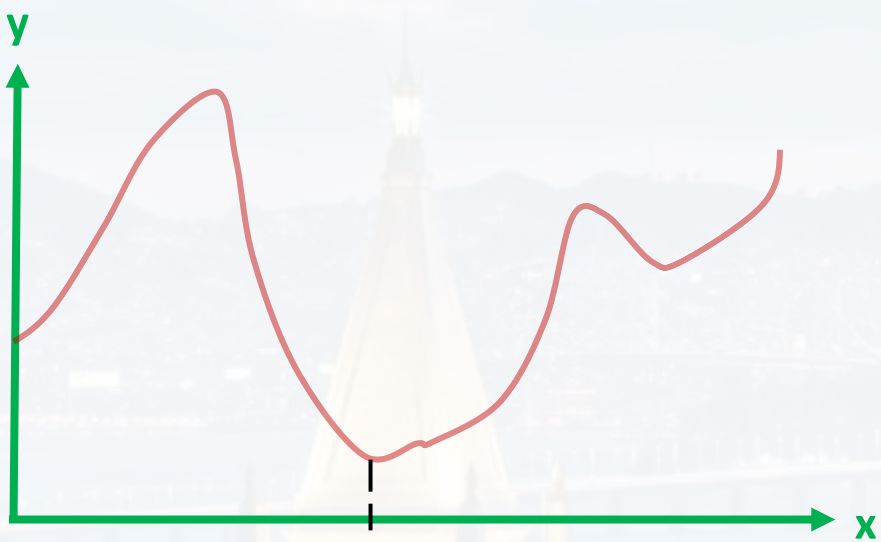


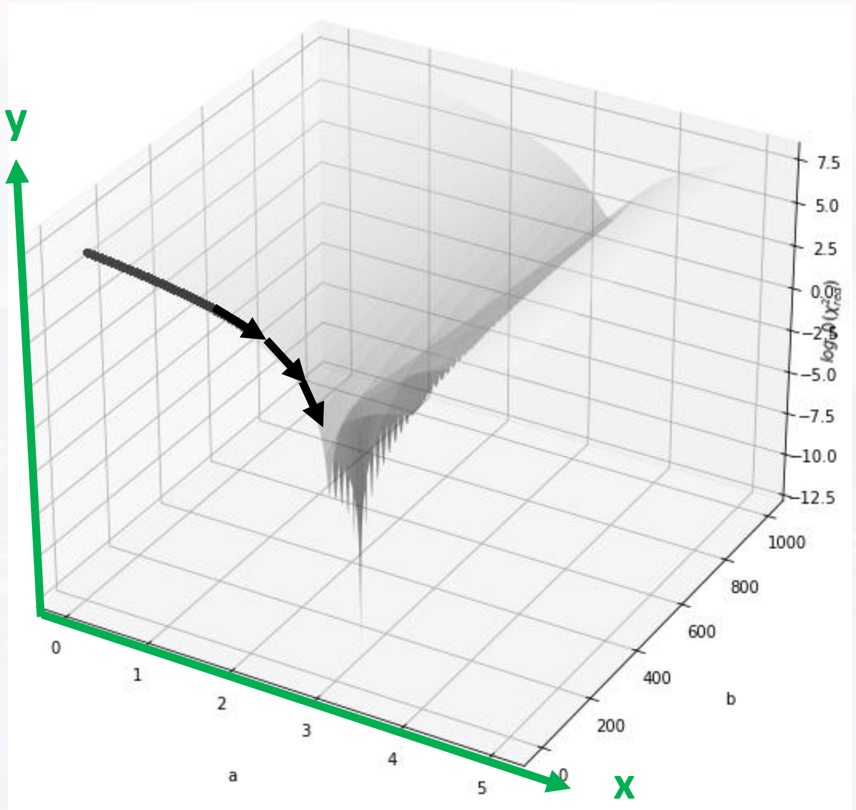
$$x(t=4) = x(t=3) - \varepsilon \left.\frac{dy}{dx}\right|_{x(t=3)}$$

ε > 0

Vanilla

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$



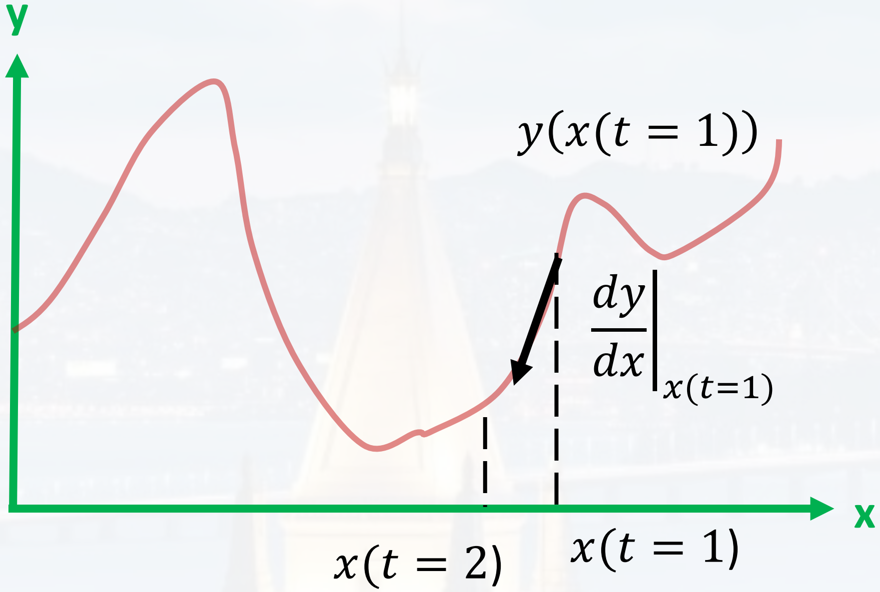


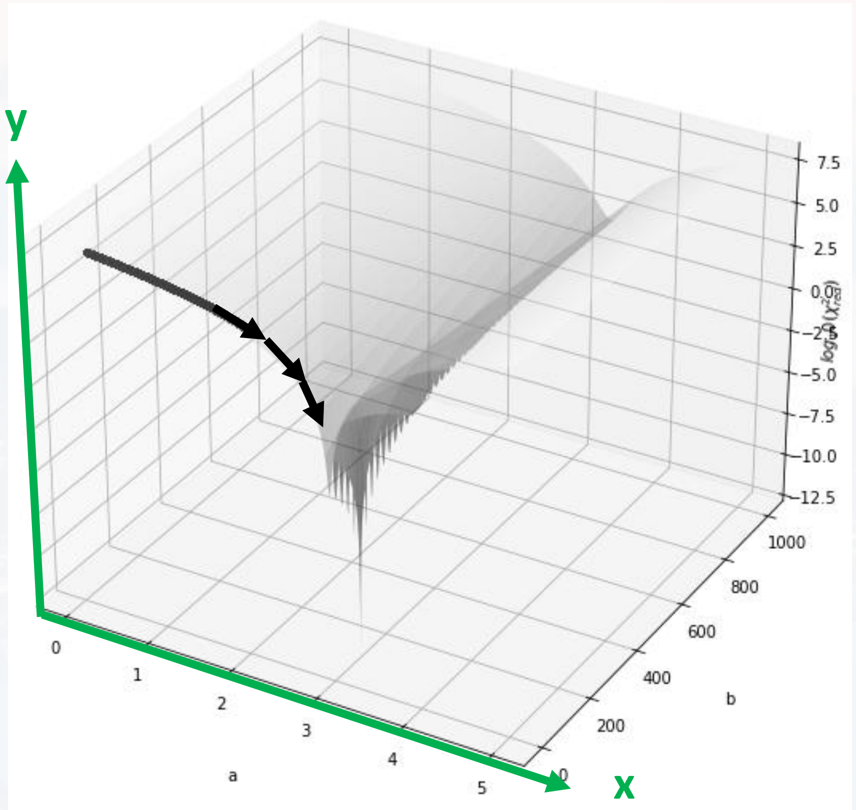
$$x(t=4) = x(t=3) - \varepsilon \left.\frac{dy}{dx}\right|_{x(t=3)}$$

ε > 0

Vanilla

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0 + \Delta x) - y(x_0 - \Delta x)}{2\Delta x}$$





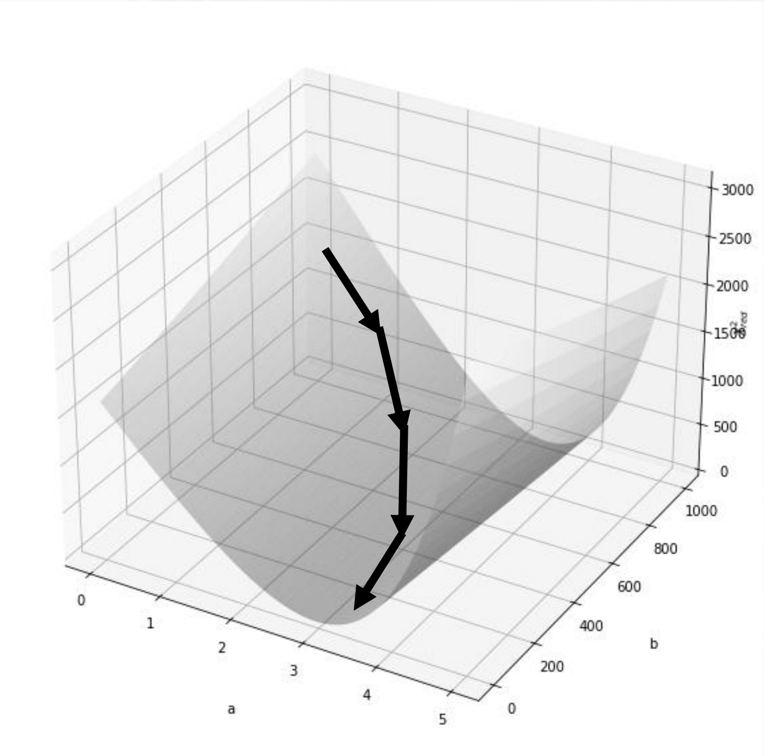
$$x(t = 2) = x(t = 1) - \varepsilon \left.\frac{dy}{dx}\right|_{x(t=1)}$$

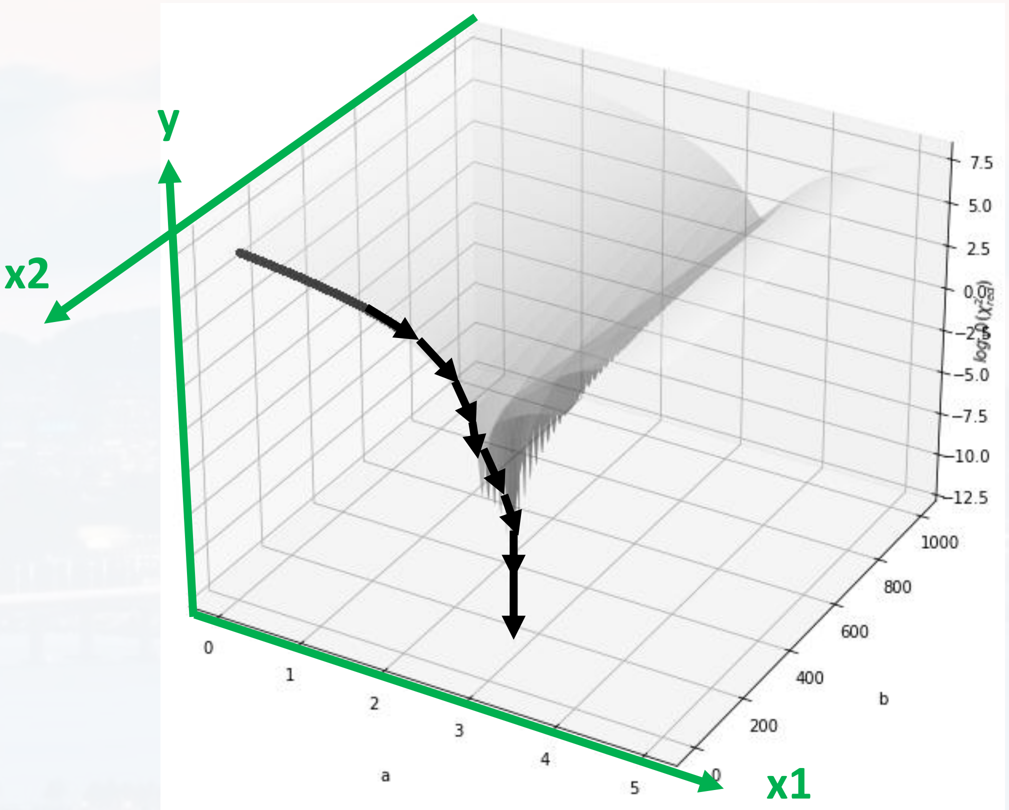
$\varepsilon > 0$

Vanilla

$$\left.\frac{\partial y}{\partial x_1}\right|_{x_1^*;\, x_2^*} \approx \frac{y(x_1^* + \Delta x_1,\ x_2^*) - y(x_1^* - \Delta x_1,\ x_2^*)}{2\Delta x_1}$$

$$\left.\frac{\partial y}{\partial x_2}\right|_{x_1^*;\, x_2^*} \approx \frac{y(x_1^*,\ x_2^* + \Delta x_2)\ - y(x_1^*,\ x_2^* - \Delta x_2)}{2\Delta x_2}$$





Optimization: Gradient Descent

Vanilla

$$\left.\frac{\partial y}{\partial x_1}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*} \approx \frac{y(x_1^* + \Delta x_1,\ x_2^*, \ldots, x_N^*) - y(x_1^* - \Delta x_1,\ x_2^*, \ldots, x_N^*)}{2\Delta x_1}$$

$$\left.\frac{\partial y}{\partial x_2}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*} \approx \frac{y(x_1^*, x_2^* + \Delta x_2, \ldots, x_N^*) - y(x_1^*, x_2^* - \Delta x_2, \ldots, x_N^*)}{2\Delta x_2}$$

$$\vdots$$

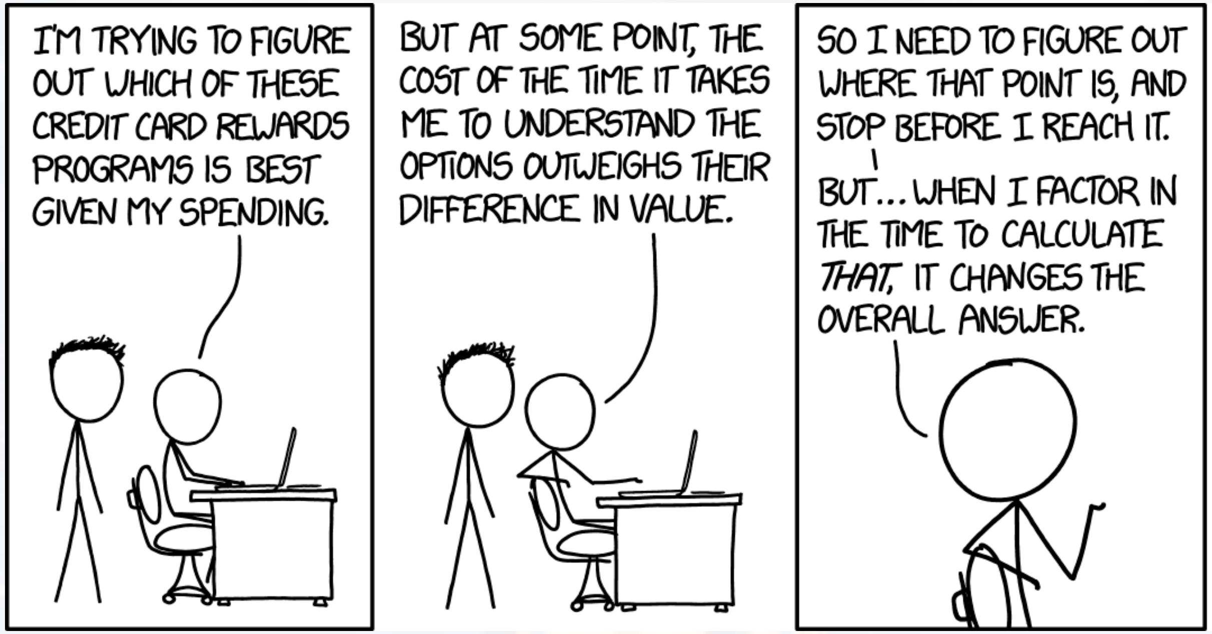
$$\left.\frac{\partial y}{\partial x_i}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*} \approx \frac{y(\ldots, x_i^* + \Delta x_i, \ldots, x_N^*)\ - y(\ldots, x_i^* - \Delta x_i, \ldots, x_N^*)}{2\Delta x_i}$$

$$\vdots$$

$$\left.\frac{\partial y}{\partial x_N}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*} \approx \frac{y(x_1^*, x_2^*, \ldots, x_N^* + \Delta x_N) - y(x_1^*, x_2^*, \ldots, x_N^* - \Delta x_N)}{2\Delta x_N}$$

$$\begin{pmatrix} \left.\frac{\partial y}{\partial x_1}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*)} \\ \ldots \\ \left.\frac{\partial y}{\partial x_i}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*} \\ \ldots \\ \left.\frac{\partial y}{\partial x_N}\right|_{x_1^*;\, x_2^*;\ldots;\, x_N^*} \end{pmatrix} = grad(y)_x$$

gradient of *y* wrt *x*



Outline

- The Problem

- Gradient Descent

- Vanilla
- Learning Rate Schedule
- Momentum
- L1 and L2
- More Finetuning

Optimization: Gradient Descent

Learning Rate Schedule

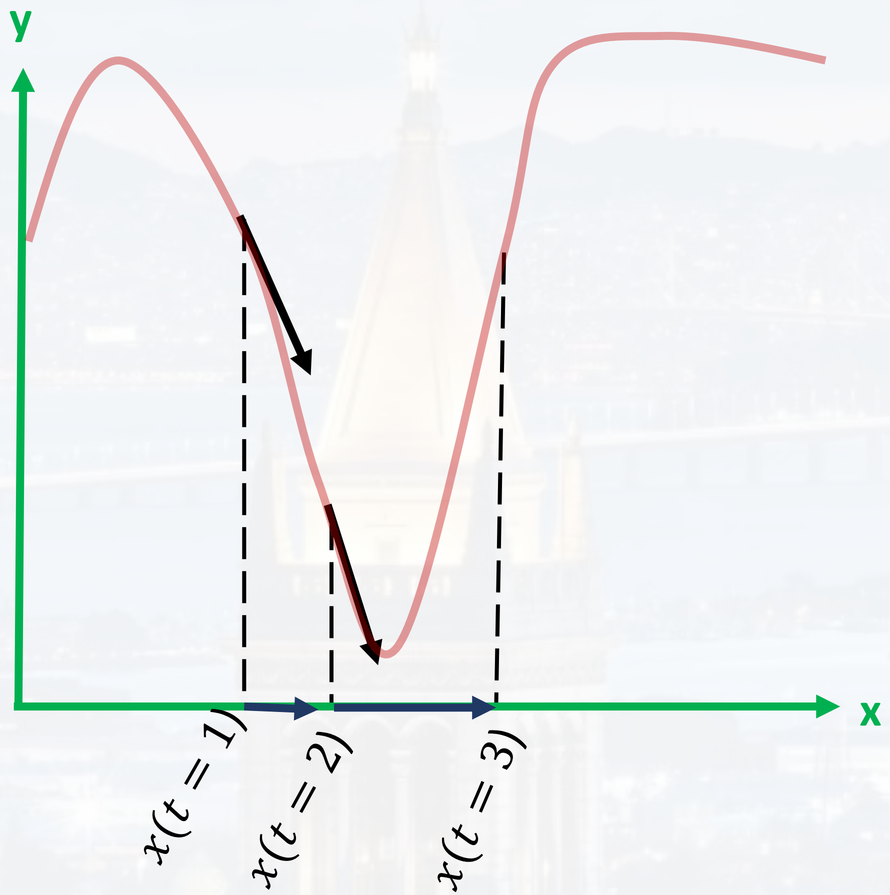
$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$

$$x(t+1) = x(t) - \boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

$\boldsymbol{\varepsilon} > 0$ called *learning rate*

$$\Delta x = -\boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

defines how large the leap Δx is

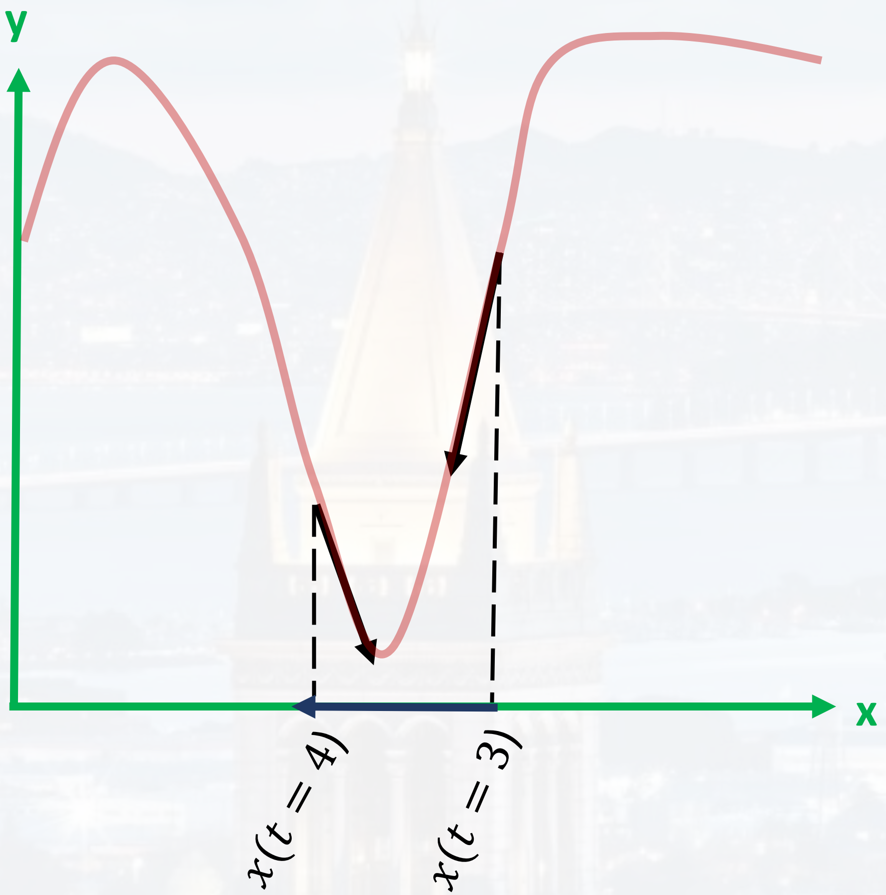


Optimization: Gradient Descent

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$

$$x(t+1) = x(t) - \boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

Learning Rate Schedule



$\boldsymbol{\varepsilon} > 0$ called *learning rate*

$$\Delta x = -\boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

defines how large the leap Δx is

Learning Rate Schedule

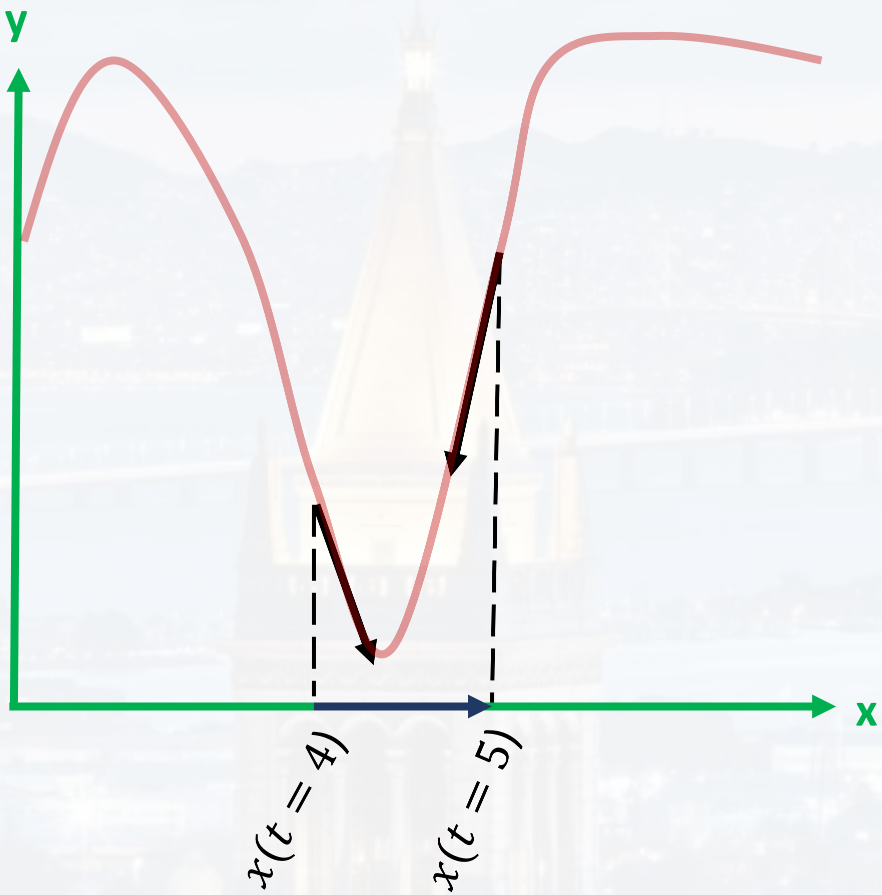
$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$

$$x(t+1) = x(t) - \boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

$\boldsymbol{\varepsilon} > 0$ called *learning rate*

$$\Delta x = -\boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

defines how large the leap Δx is



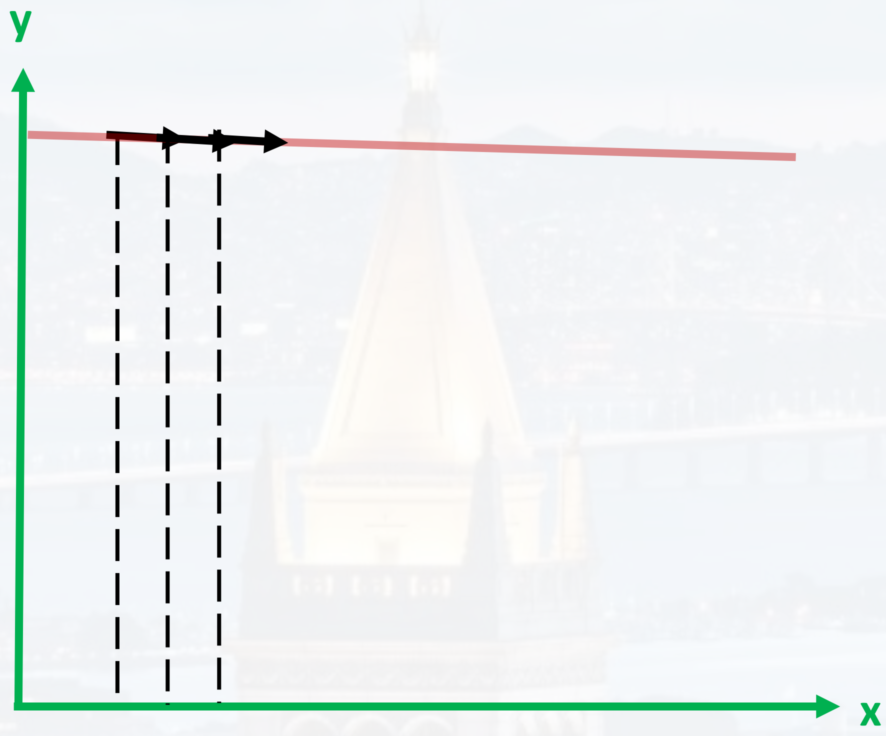
… and so on…

→ smaller $\boldsymbol{\varepsilon}$?

Learning Rate Schedule

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x)\ - y(x_0-\Delta x)}{2\Delta x}$$

$$x(t+1) = x(t)\ -\ \boldsymbol{\varepsilon}\left.\frac{dy}{dx}\right|_{x(t)}$$



$\boldsymbol{\varepsilon} > 0$ called *learning rate*

$$\Delta x = -\ \boldsymbol{\varepsilon}\left.\frac{dy}{dx}\right|_{x(t)}$$

defines how large the leap Δx is

… and so on…

→ smaller $\boldsymbol{\varepsilon}$? Takes too long!

Learning Rate Schedule

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$

$$x(t+1) = x(t) - \boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

$\boldsymbol{\varepsilon} > 0$ called *learning rate*

$$\Delta x = -\boldsymbol{\varepsilon}\left.\frac{dy}{dx}\right|_{x(t)}$$

defines how large the leap Δx is

learning rate as function of t:

$$\boldsymbol{\varepsilon}(t) = \frac{\boldsymbol{\varepsilon}_0}{1+\kappa t}$$

decay rate κ

Learning Rate Schedule

$$\left.\frac{dy}{dx}\right|_{x_0} \approx \frac{y(x_0+\Delta x) - y(x_0-\Delta x)}{2\Delta x}$$

$$x(t+1) = x(t) - \boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

$\boldsymbol{\varepsilon} > 0$ called *learning rate*

$$\Delta x = -\boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$

defines how large the leap Δx is

<u>learning rate as function of t:</u>

$$\boldsymbol{\varepsilon}(t) = \frac{\boldsymbol{\varepsilon}_0}{1+\kappa t}$$

decay rate κ

can also be a stepwise function (learning rate schedule)

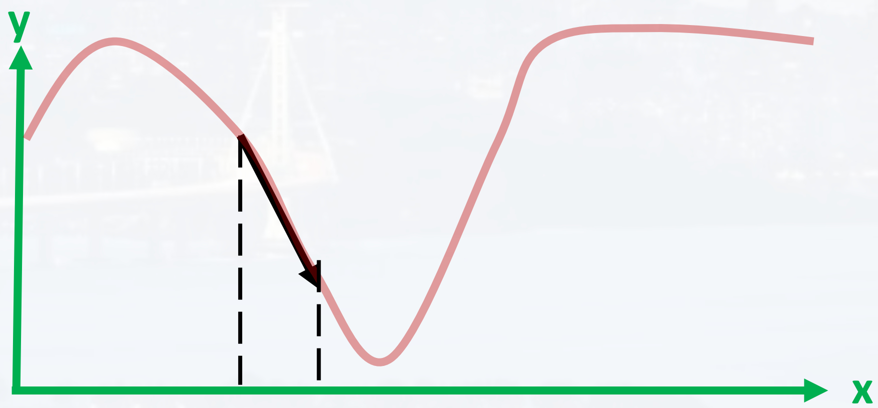
Learning Rate Schedule

learning rate as function of t:

$$\boldsymbol{\varepsilon}(t) = \frac{\boldsymbol{\varepsilon}_0}{1 + \kappa\, t}$$ decay rate κ

$$\Delta x = -\,\boldsymbol{\varepsilon} \left.\frac{dy}{dx}\right|_{x(t)}$$ defines how large the leap Δx is

can also be a stepwise function (learning rate schedule)

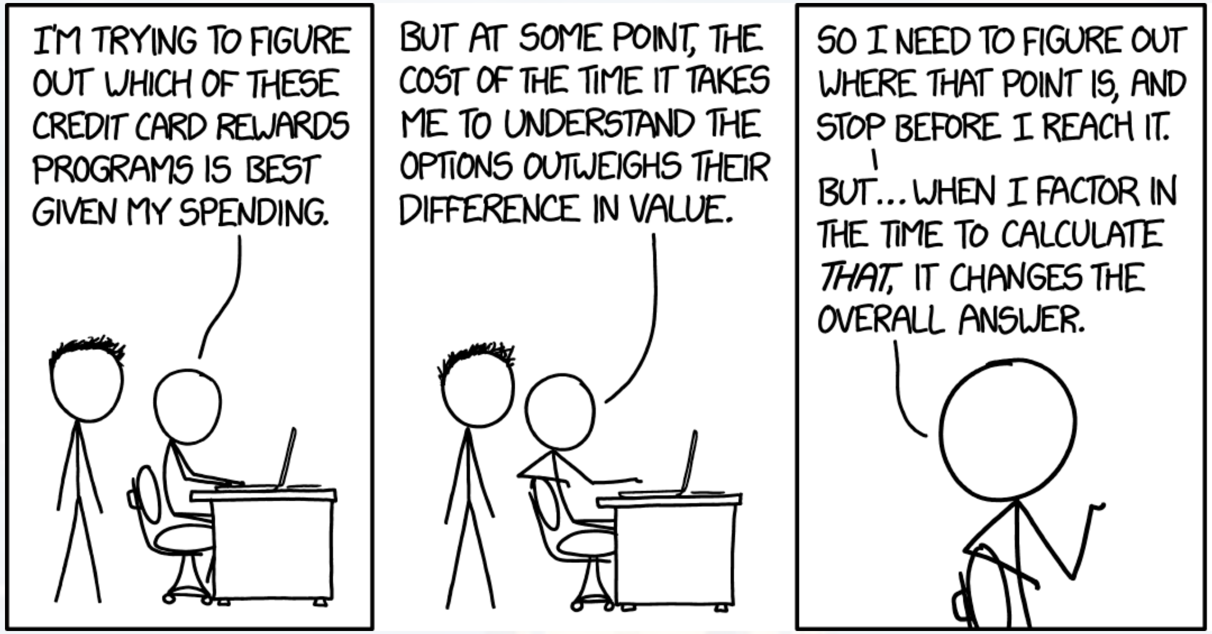


better:
adaptive gradient, aka **AdaGrad**

$$r_{t+1} = r_t + grad(y)_x \oplus grad(y)_x$$

$\oplus$: outer product
δ: small, >0

$$\boldsymbol{\varepsilon} \rightarrow \frac{\boldsymbol{\varepsilon}}{\sqrt{r_{t+1}} + \delta}$$



Outline

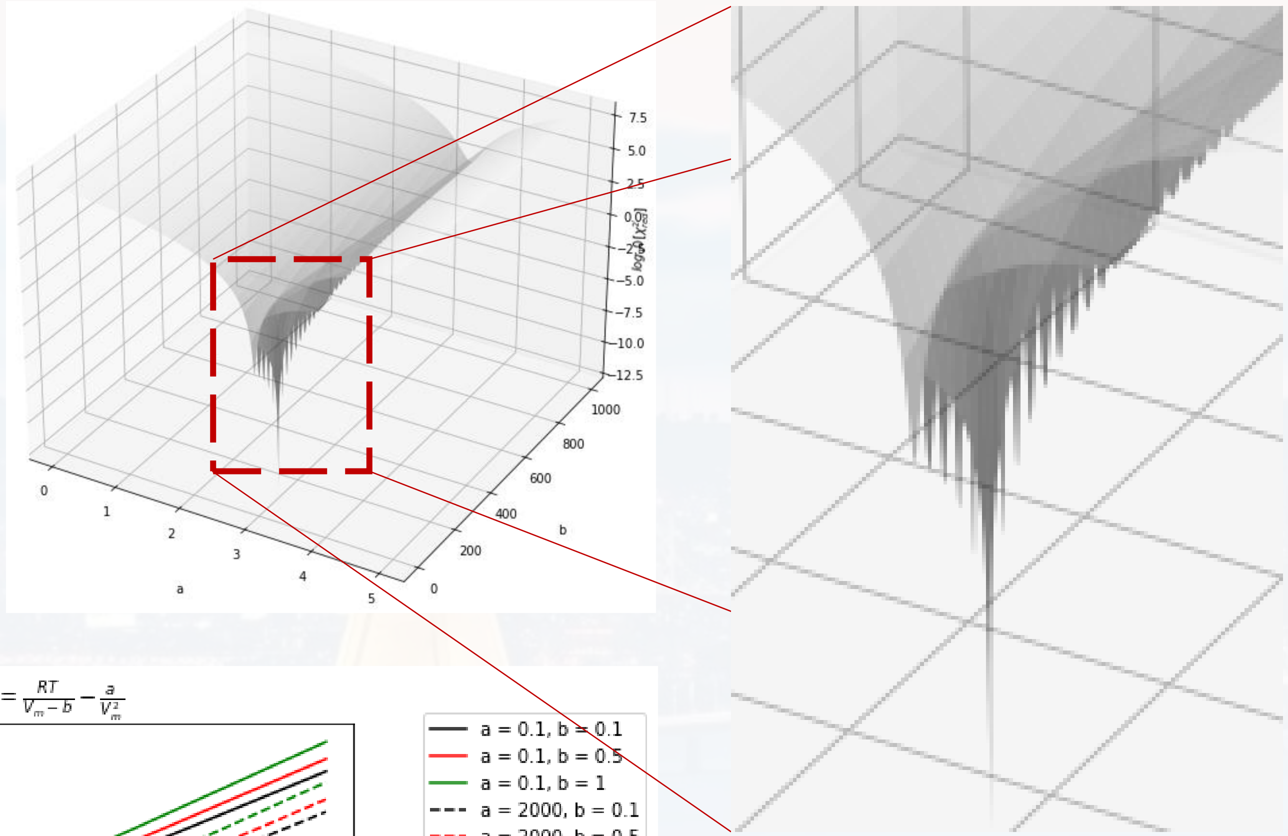
- The Problem

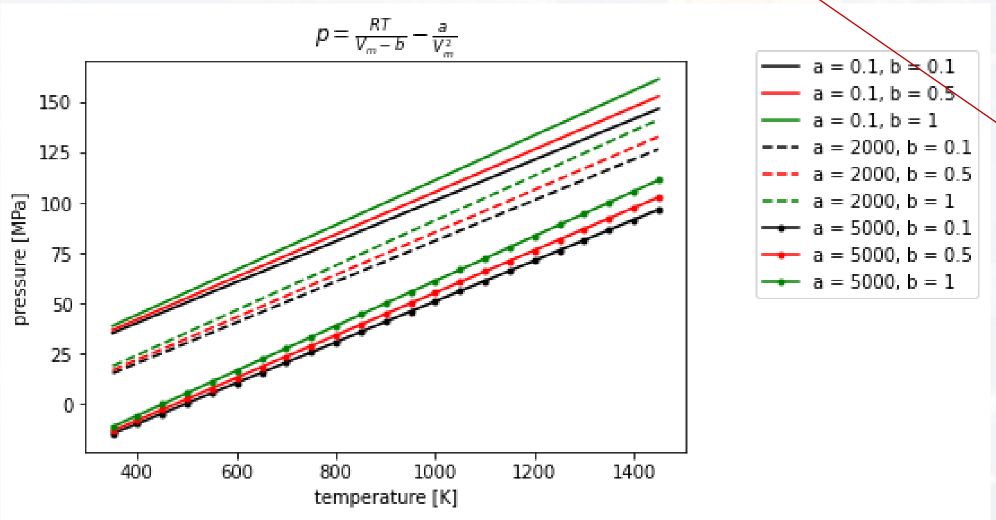
- Gradient Descent

- Vanilla
- Learning Rate Schedule
- Momentum
- L1 and L2
- More Finetuning

Optimization: Gradient Descent

Momentum

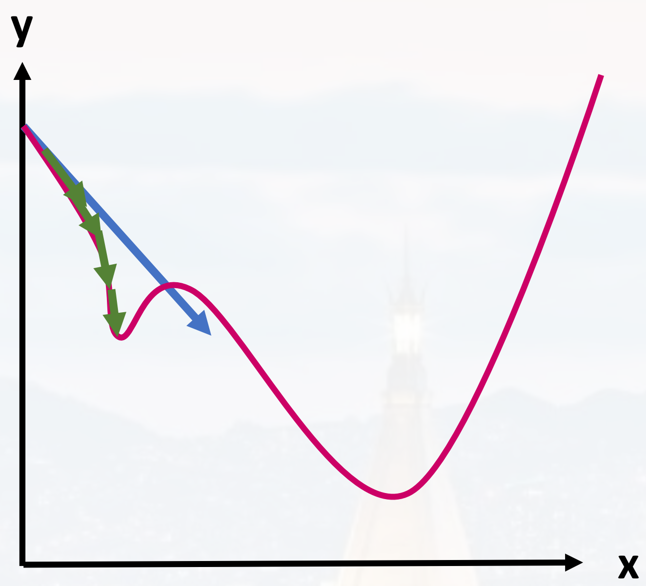




even with AdaGrad and learning rate schedule
→ would get stuck in local minimum

need to roll over → **momentum**

Momentum



taking the **average** of ***N*** previous gradients

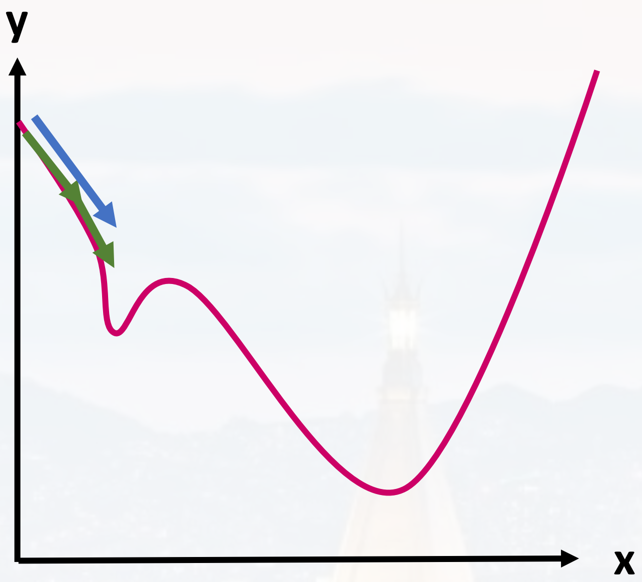
$$\langle grad(y)_{x(t)} \rangle = \frac{1}{N}[grad(y)_{x(t-1)} + grad(y)_{x(t-2)} + \ldots + grad(y)_{x(t-N)}]$$

but we want more recent gradients to contribute more than older gradients

→ **weighted average** with weighting factor $\boldsymbol{\mu_k}$

$$\langle grad(y)_{x(t)} \rangle = \sum_{k=t-N}^{t-1} \mu_k \cdot grad(y)_{x(k)}$$

Finding a clever way to adjust $\boldsymbol{\mu_k}$ during every iteration *t*



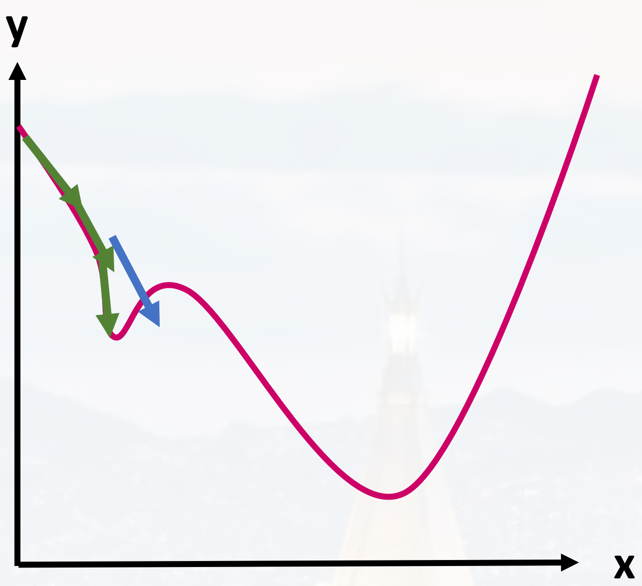
Momentum

weighted average with weighting factor $\boldsymbol{\mu_k}$

Finding a clever way to adjust $\boldsymbol{\mu_k}$ during every iteration t

$$\langle grad(y)_{x(0)} \rangle = grad(y)_{x(0)} \qquad \mu_0 = (0,1)$$

$$\langle \boldsymbol{grad(y)_{x(1)}} \rangle = grad(y)_{x(1)} + \mu_0 \cdot grad(y)_{x(0)}$$



Momentum

weighted average with weighting factor $\boldsymbol{\mu_k}$

Finding a clever way to adjust $\boldsymbol{\mu_k}$ during every iteration t

$$\langle grad(y)_{x(0)} \rangle = grad(y)_{x(0)} \qquad \mu_0 = (0,1)$$

$$\langle \boldsymbol{grad(y)_{x(1)}} \rangle = grad(y)_{x(1)} + \mu_0 \cdot grad(y)_{x(0)}$$

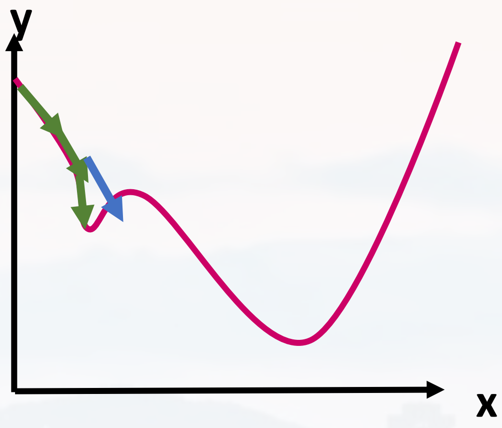
$$\langle \boldsymbol{grad(y)_{x(2)}} \rangle = grad(y)_{x(2)} + \boxed{\mu_0} \, [grad(y)_{x(1)} + \boxed{\mu_0} \, grad(y)_{x(0)}]$$

$$\boldsymbol{\mu_{k=2}} = \mu_0 \, \mu_0 = \mu_0^2$$

$$\langle \boldsymbol{grad(y)_{x(3)}} \rangle = grad(y)_{x(3)} + \boxed{\mu_0} \Big[grad(y)_{x(2)} + \boxed{\mu_0} [grad(y)_{x(1)} + \boxed{\mu_0} \cdot grad(y)_{x(0)}] \Big]$$

… and so on…

Momentum



weighted average with weighting factor $\boldsymbol{\mu_k}$

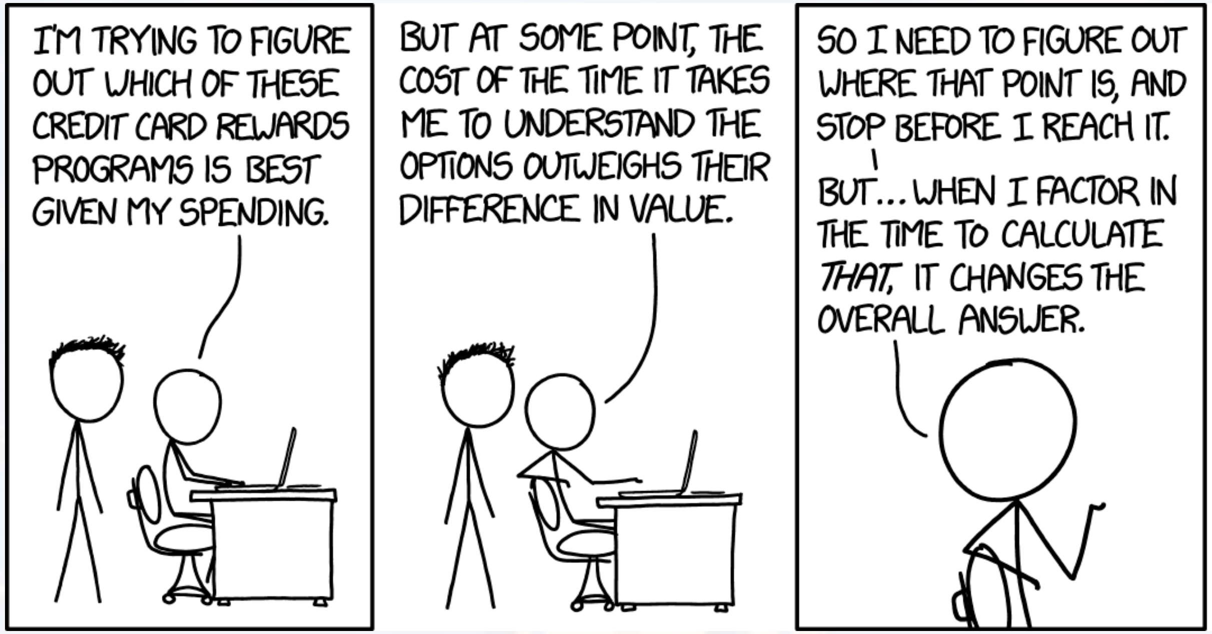
$\mu_0 = (0{,}1)$ called ''momentum'

$$\langle \boldsymbol{grad(y)_{x(3)}} \rangle = grad(y)_{x(3)} \; +$$

$$\mu_0 \left[grad(y)_{x(2)} + \mu_0 \left[grad(y)_{x(1)} + \mu_0 \cdot grad(y)_{x(0)} \right] \right]$$ ... and so on...

```
class Optimizer:

    def __init__(self, learning_rate = 0.1, decay = 0, momentum = 0):
      self.learning_rate         = learning_rate
      self.decay                 = decay
      self.current_learning_rate = learning_rate
      self.iterations            = 0
      self.momentum              = momentum
```



Outline

- The Problem

- Gradient Descent

- Vanilla
- Learning Rate Schedule
- Momentum
- L1 and L2
- More Finetuning

L1 and L2

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

Often, the extreme of the objective function is subject to **constrains**

sometimes we have some **prior knowledge** about the **independent variables**

recall: linear regression

finding best β by $$\min_{\beta}\left\{\frac{1}{N}\|Y - X\beta\|^2\right\}$$

now:
constrain: **encourages sparsity of $\boldsymbol{\beta}$** $$\min_{\beta}\left\{\frac{1}{N}\|Y - X\beta\|^2 + \boldsymbol{\lambda\|\beta\|^1}\right\}$$

λ *Lagrangian Multiplier*

called **L1 regularization**, or LASSO

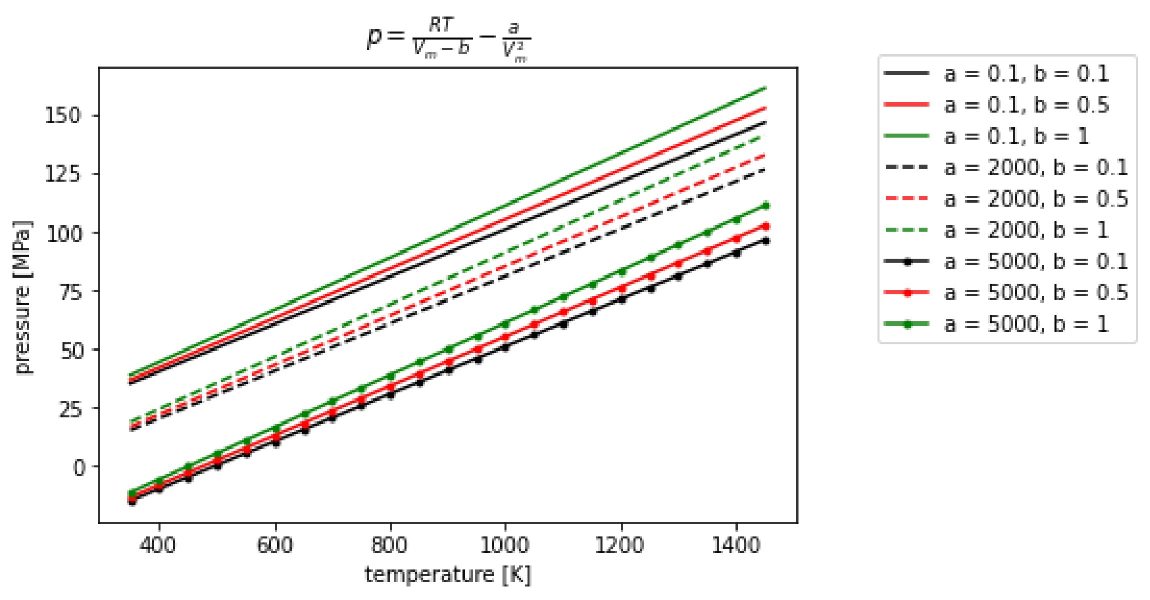
L1 and L2

Any algorithm needs a "goal" aka **objective function** that has to be ***optimized*** (finding an **extreme**)

Often, the extreme of the objective function is subject to **constrains**

sometimes we have some **prior knowledge** about the **independent variables**

L1 regularization



We often have even hard constrains based on the laws of physics!

L1 and L2

Any algorithm needs a “goal” aka **objective function** that has to be ***optimized*** (finding an **extreme**)

Often, the extreme of the objective function is subject to **constrains**

sometimes we have some **prior knowledge** about the **independent variables**

recall: linear regression

finding best β by

$$\min_{\beta} \left\{ \frac{1}{N} \| Y - X\beta \|^2 \right\}$$

now:

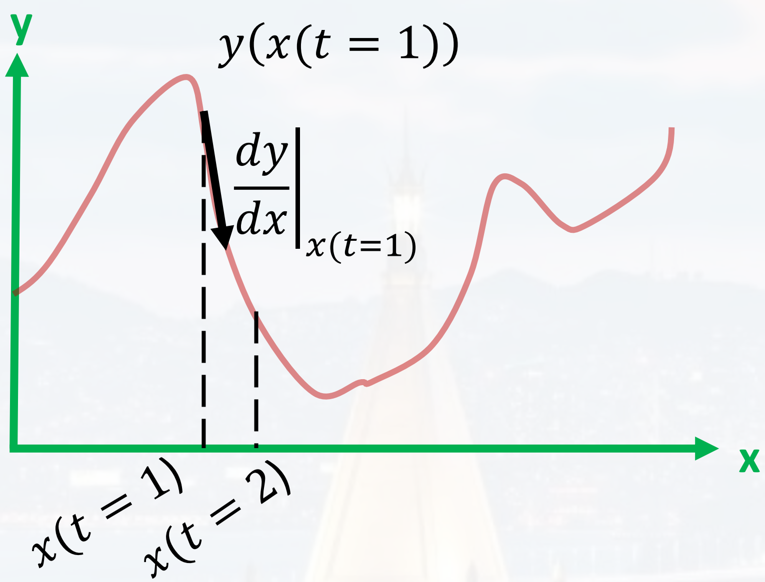
$$\hat{\beta} = (X^T X + \boldsymbol{\lambda I})^{-1} X^T Y \longrightarrow \min_{\beta} \left\{ \frac{1}{N} \| Y - X\beta \|^2 + \boldsymbol{\lambda} \| \boldsymbol{\beta} \|^2 \right\}$$

λ *Lagrangian Multiplier*

called **L2 regularization**, or RIDGE penalizes large β

L1 and L2 regularization

L1 and L2

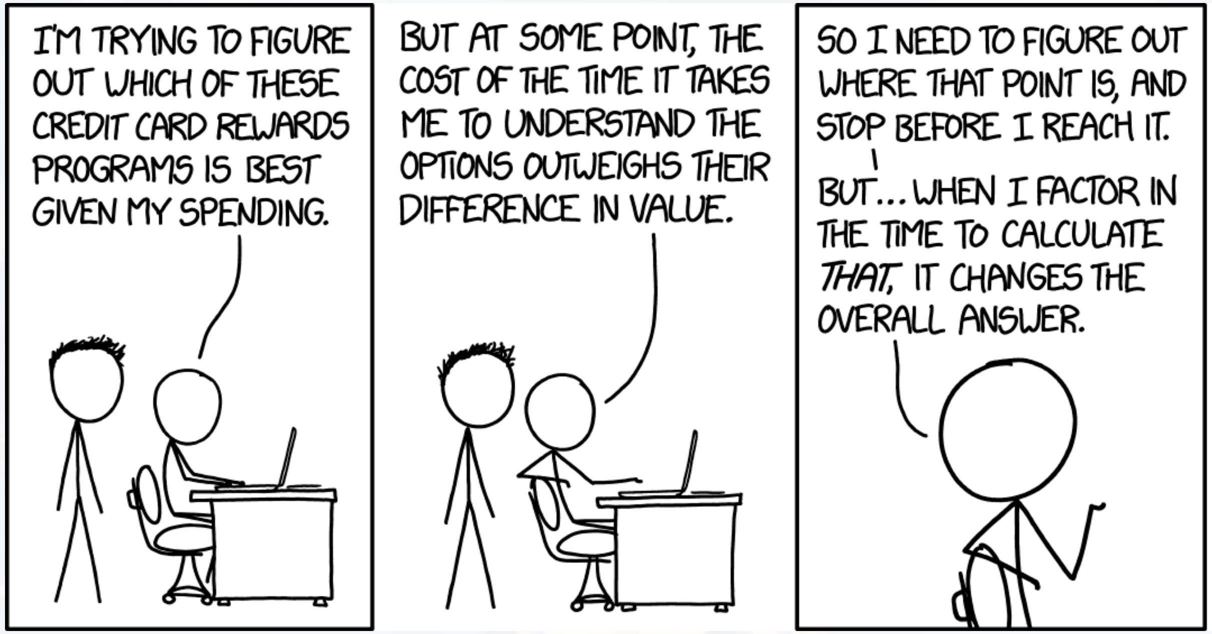


$$x(t=2) = x(t=1) - \varepsilon \left.\frac{d[y + \lambda_1 \|x\|^1 + \lambda_2 \|x\|^2]}{dx}\right|_{x(t=1)}$$

$$x(t=2) = x(t=1) - \varepsilon \left.\frac{dy}{dx}\right|_{x(t=1)}$$

$$- \varepsilon \left.\frac{\lambda_1 d\|x\|^1}{dx}\right|_{x(t=1)} \quad - \varepsilon \left.\frac{\lambda_2 d\|x\|^2}{dx}\right|_{x(t=1)}$$

- gradient descent does not stop if values for x are too large and prefers sparsity
- note: the derivative of $\|x\|^1$ returns the sign (i. e. direction)
- usually $\lambda \ll \|x\|^n$
- will be important for ANNs later



Outline

- The Problem

- Gradient Descent

- Vanilla
- Learning Rate Schedule
- Momentum
- L1 and L2
- More Finetuning

Optimization: Gradient Descent

More Fine Tuning

Vanilla Gradient Descent → **S**tochastic **G**radient **D**escent

Learning Rate Schedule, L1, L2

different scaling for all different direction

momentum

$\varepsilon \rightarrow \frac{\varepsilon}{\sqrt{r_{t+1}}+\delta}$ adaptive gradient, aka **AdaGrad**

Multiplying a decay factor to the sum of gradient squared (similar to momentum), aka **R**oot **M**ean **S**quare **Prop**agation **RMSProp**

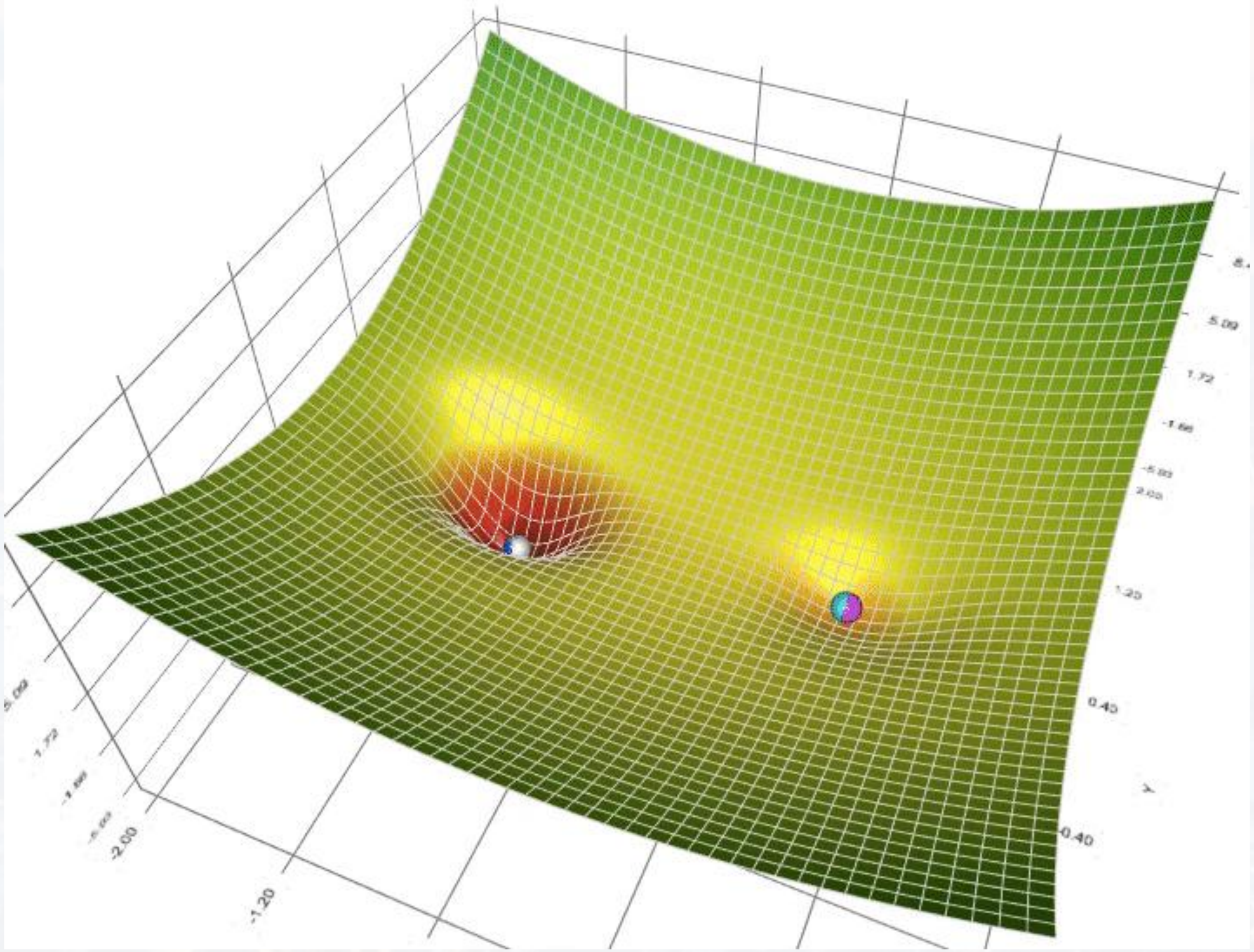
all combined:
Adaptive **M**oment Estimation
aka **Adam**

Optimization:

Lili Jiang

[TowardsDataScience]

More Fine Tuning



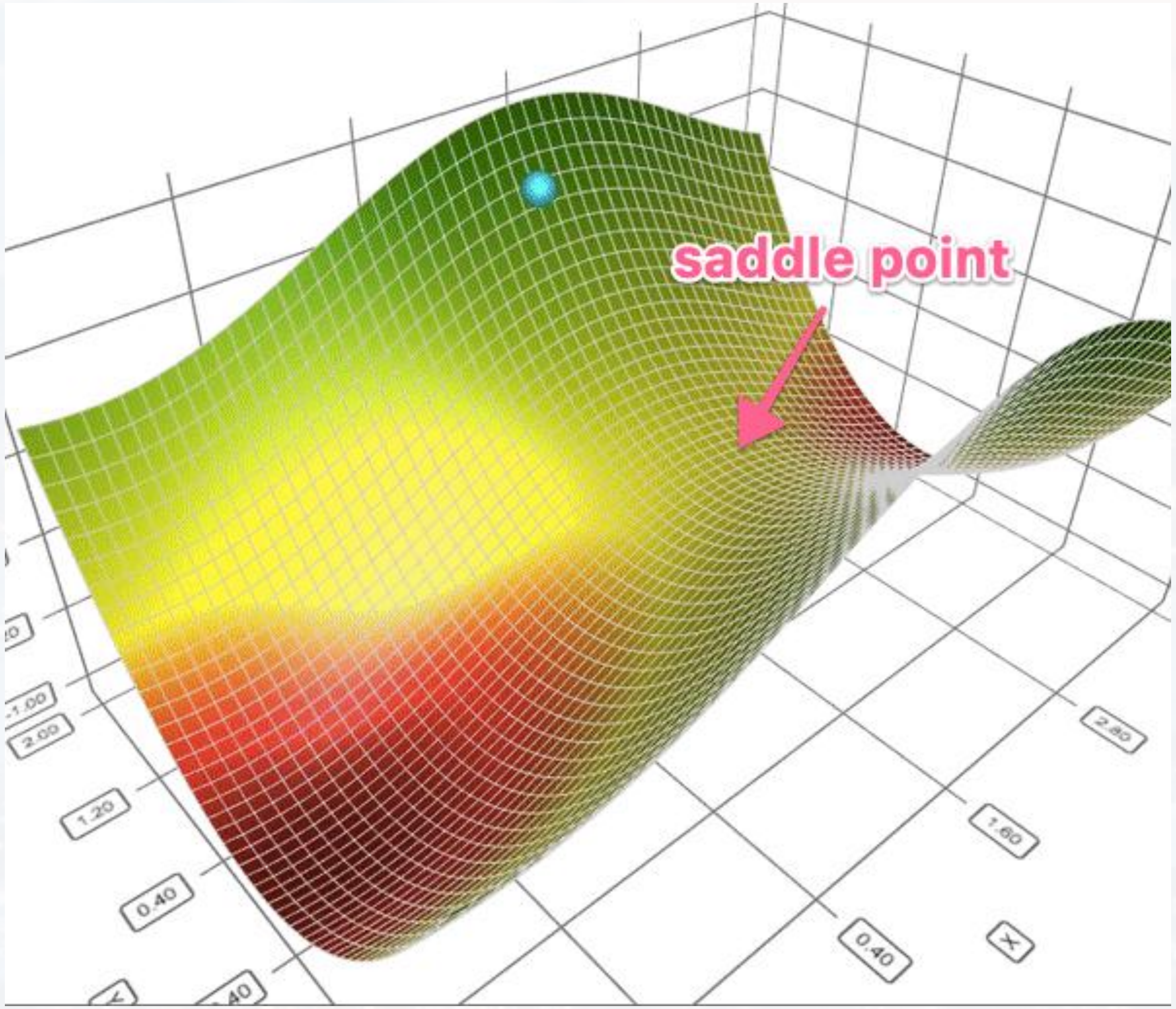
gradient descent (cyan),
momentum (magenta),
AdaGrad (white),
RMSProp (green),
Adam (blue)

Optimization: Gradient Descent

Lili Jiang

TowardsDataScience

More Fine Tuning



gradient descent (cyan),
momentum (magenta),
AdaGrad (white),
RMSProp (green),
Adam (blue)

Thank you very much for your attention!