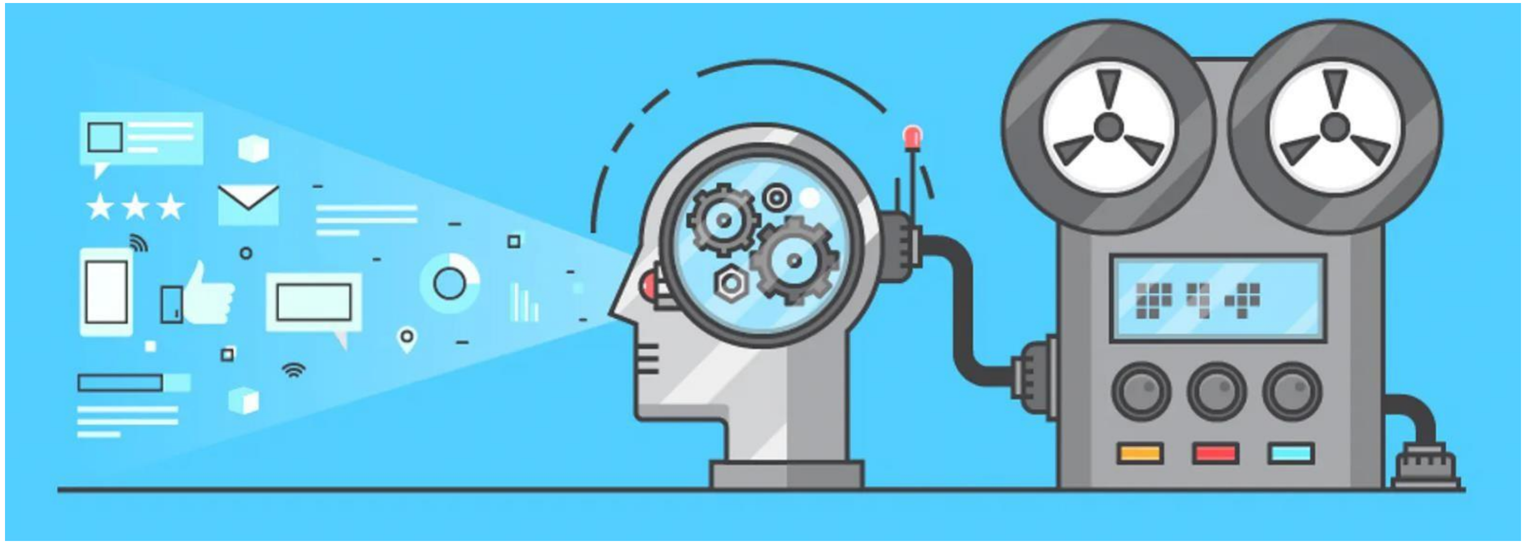
Deep Learning and Neural Networks

Topic 4: Optimization



Ricardo Abel Espinosa Loera, McS

Researcher in DL & Computer Vision

Recap

Training neural networks

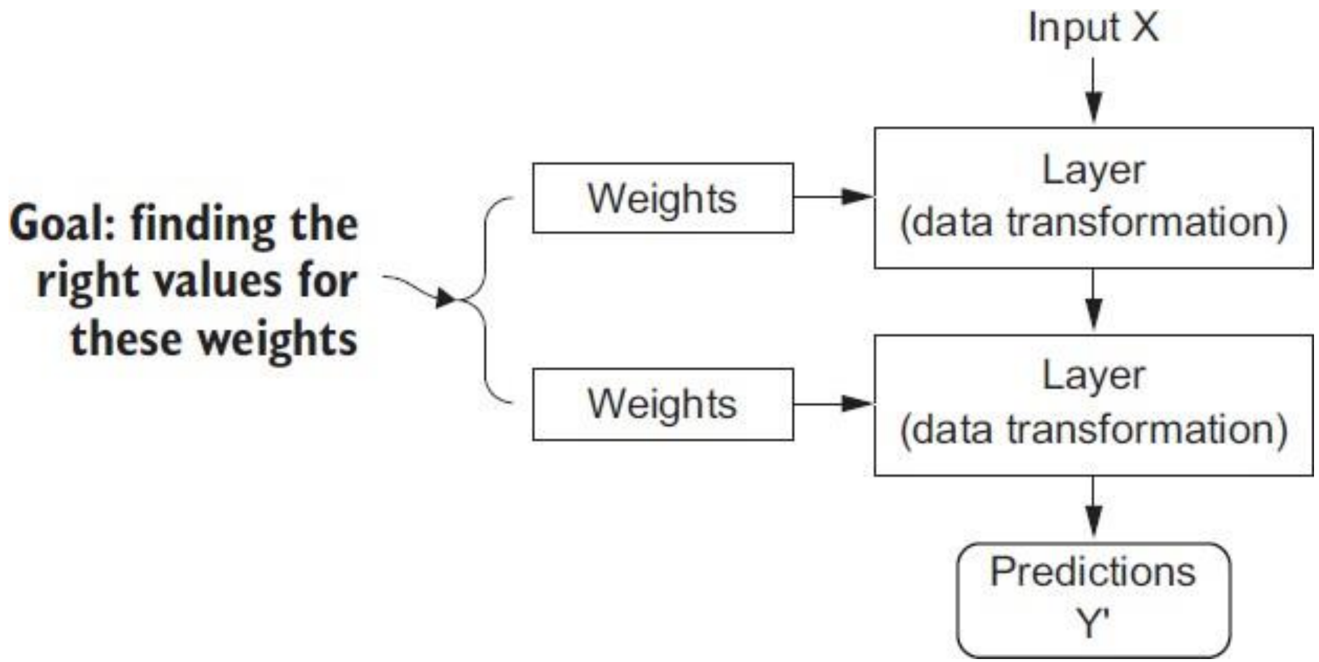
- Optimization algorithms are the basic engine behind deep learning methods that enable models to learn from data by adapting their parameters

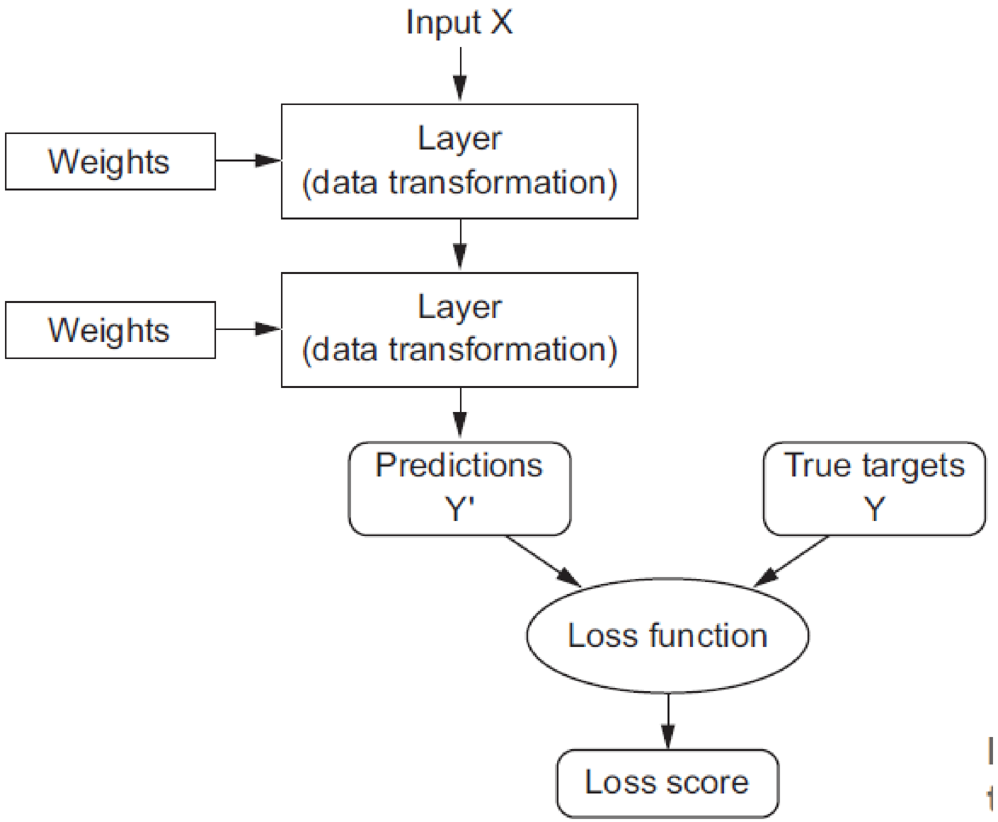
- They solve the problem of the minimization of an objective function that measures the mistakes made by the model

  - e.g. prediction error (classification), negative reward (reinforcement learning)

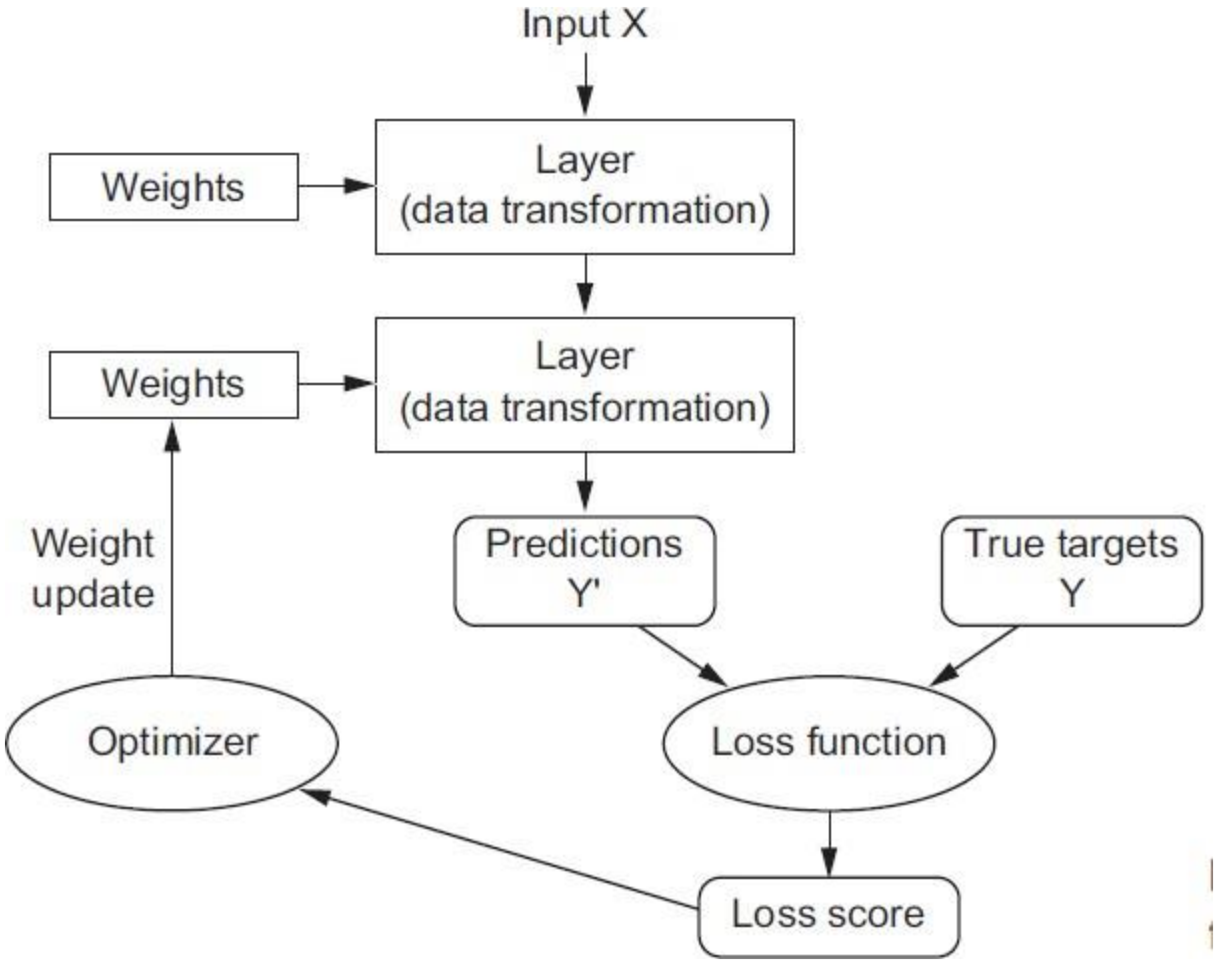
- Work by making a sequence of small incremental changes to model parameters that are each guaranteed to reduce the objective by some small amount

Recap
Training neural networks



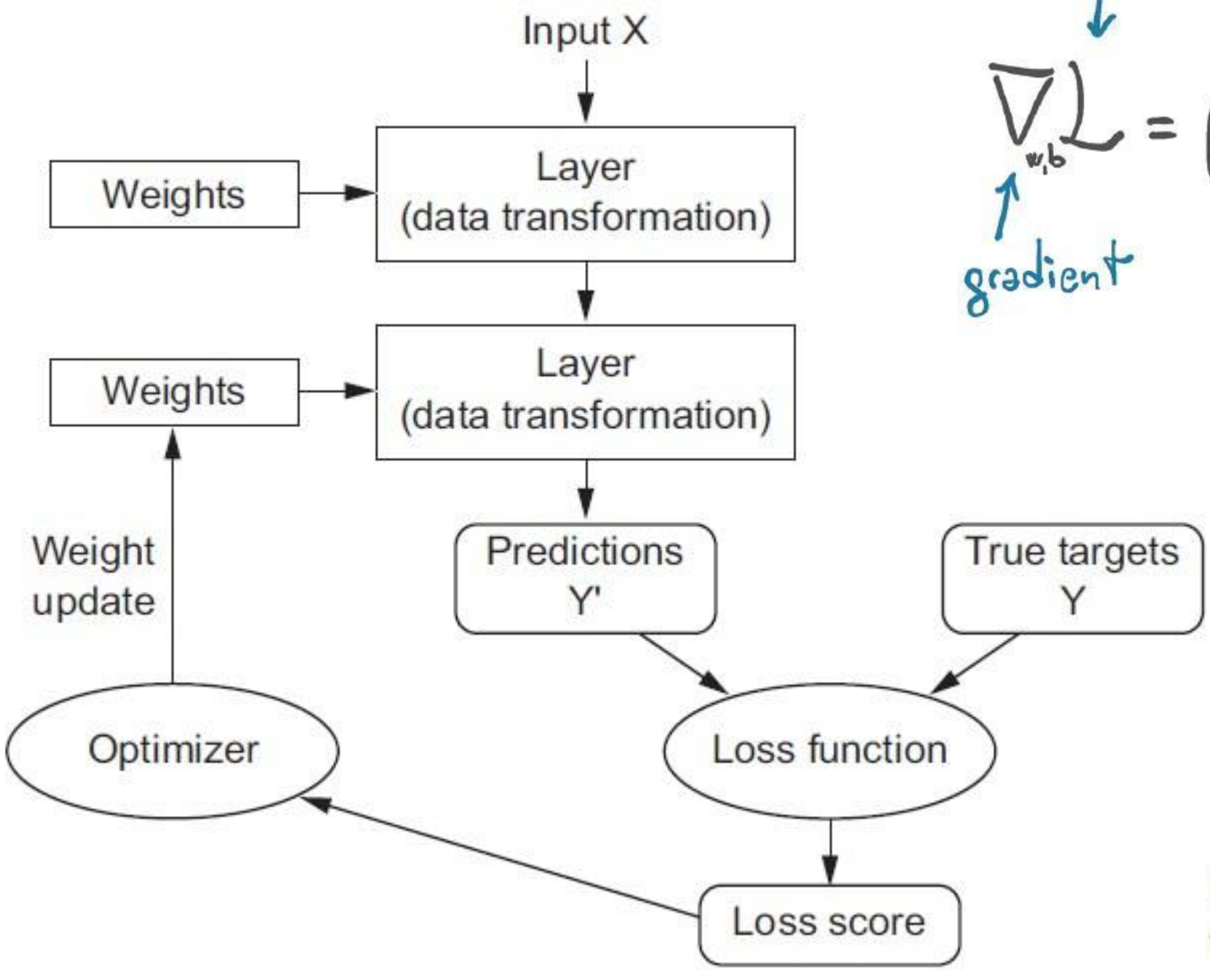


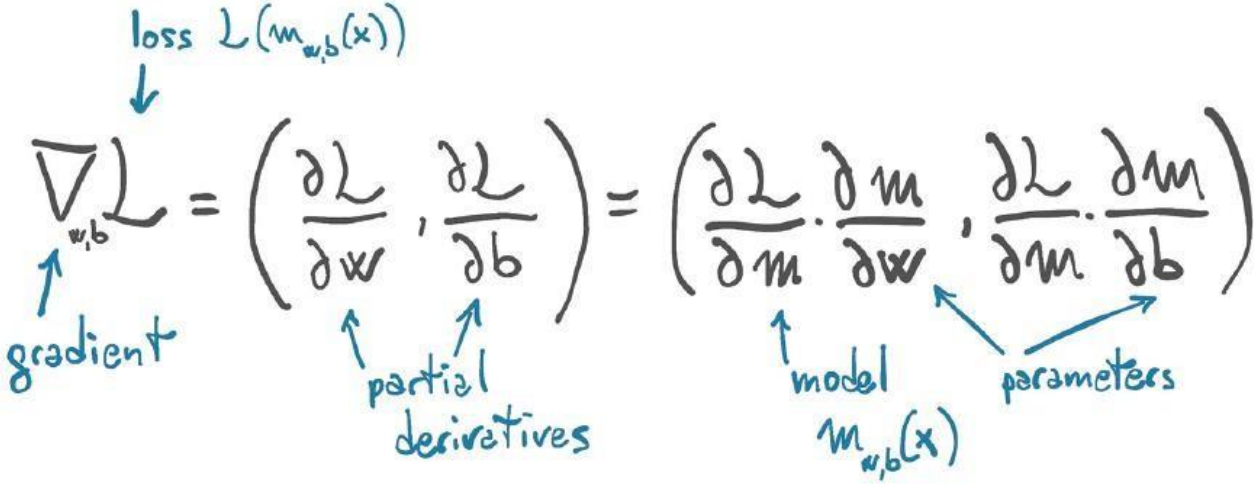
Recap
Training neural networks

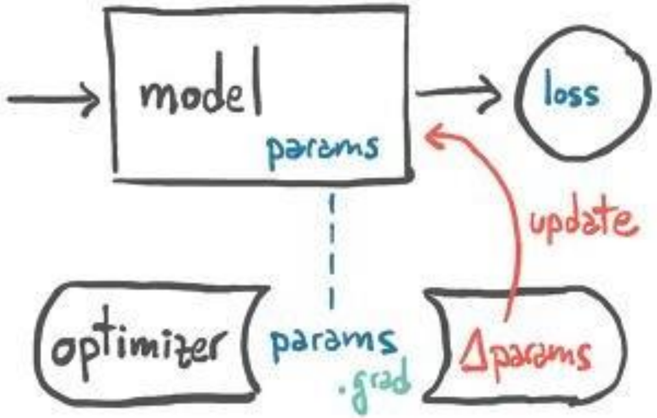


Recap

Training neural networks

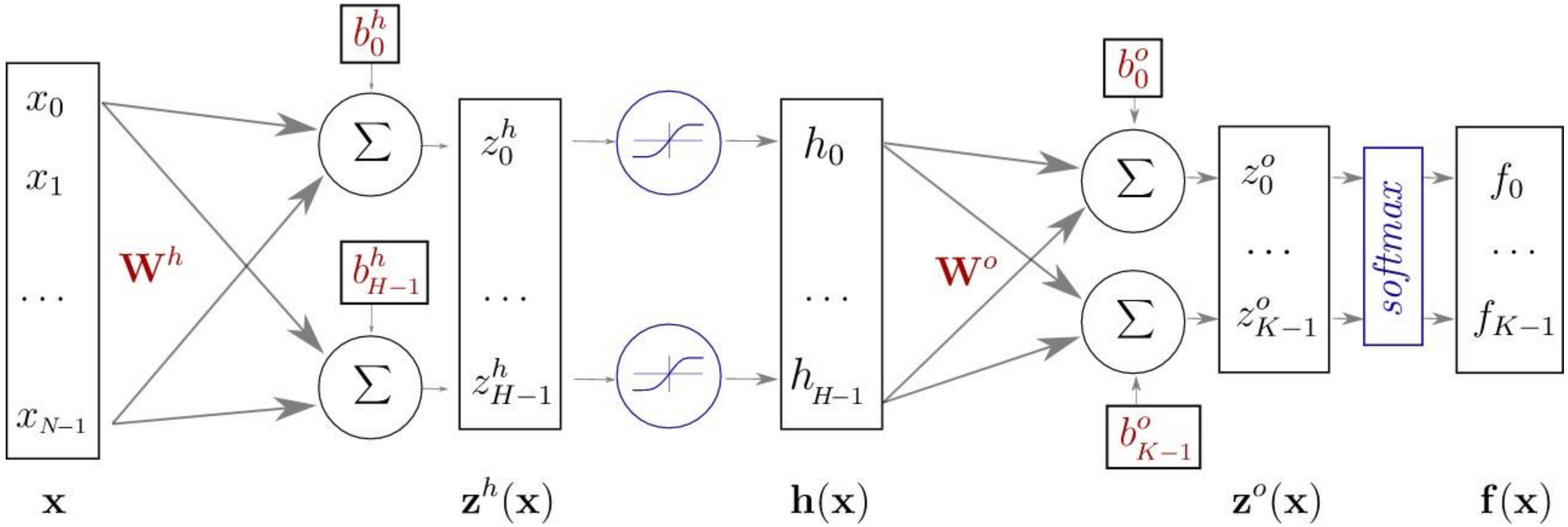






Recap

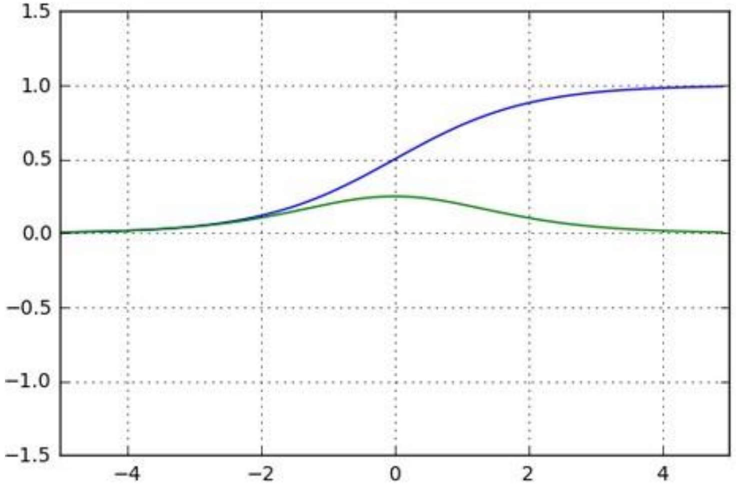
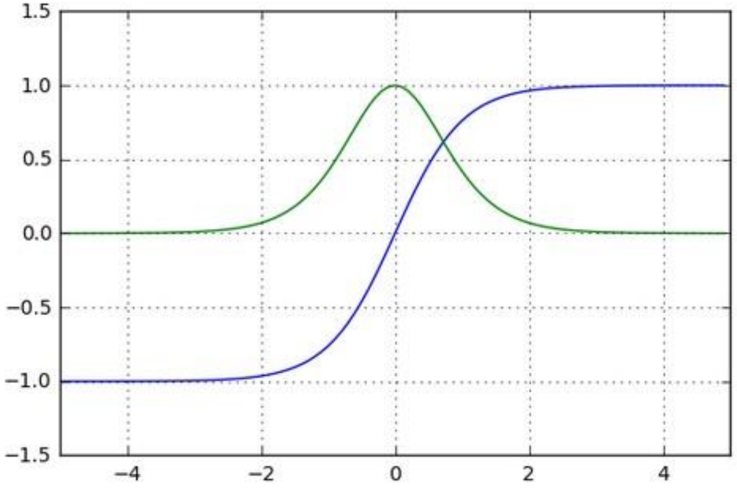
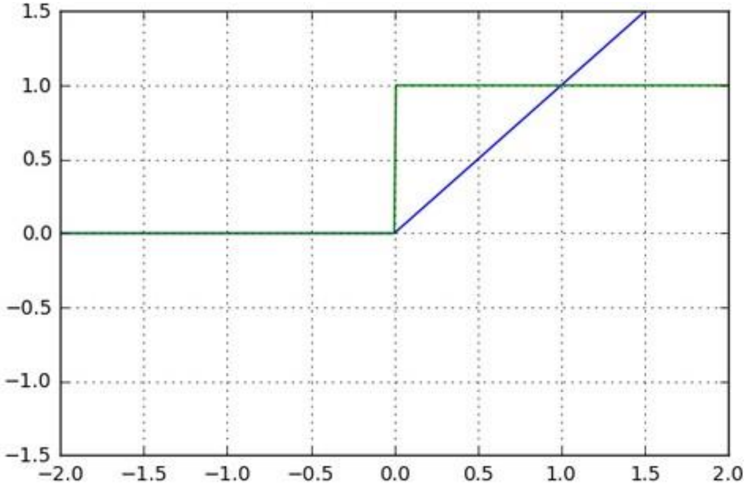
Training neural networks



Keras Implementation

```
model = Sequential()
model.add(Dense(H, input_dim=N))  # weight matrix dim [N * H]
model.add(Activation("tanh"))
model.add(Dense(K))               # weight matrix dim [H x K]
model.add(Activation("softmax"))
```

Recap
Training neural networks

$$\mathrm{sigm}(x) = \frac{1}{1+e^{-x}}$$

$$\mathrm{sigm}'(x) = \mathrm{sigm}(x)(1-\mathrm{sigm}(x))$$

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

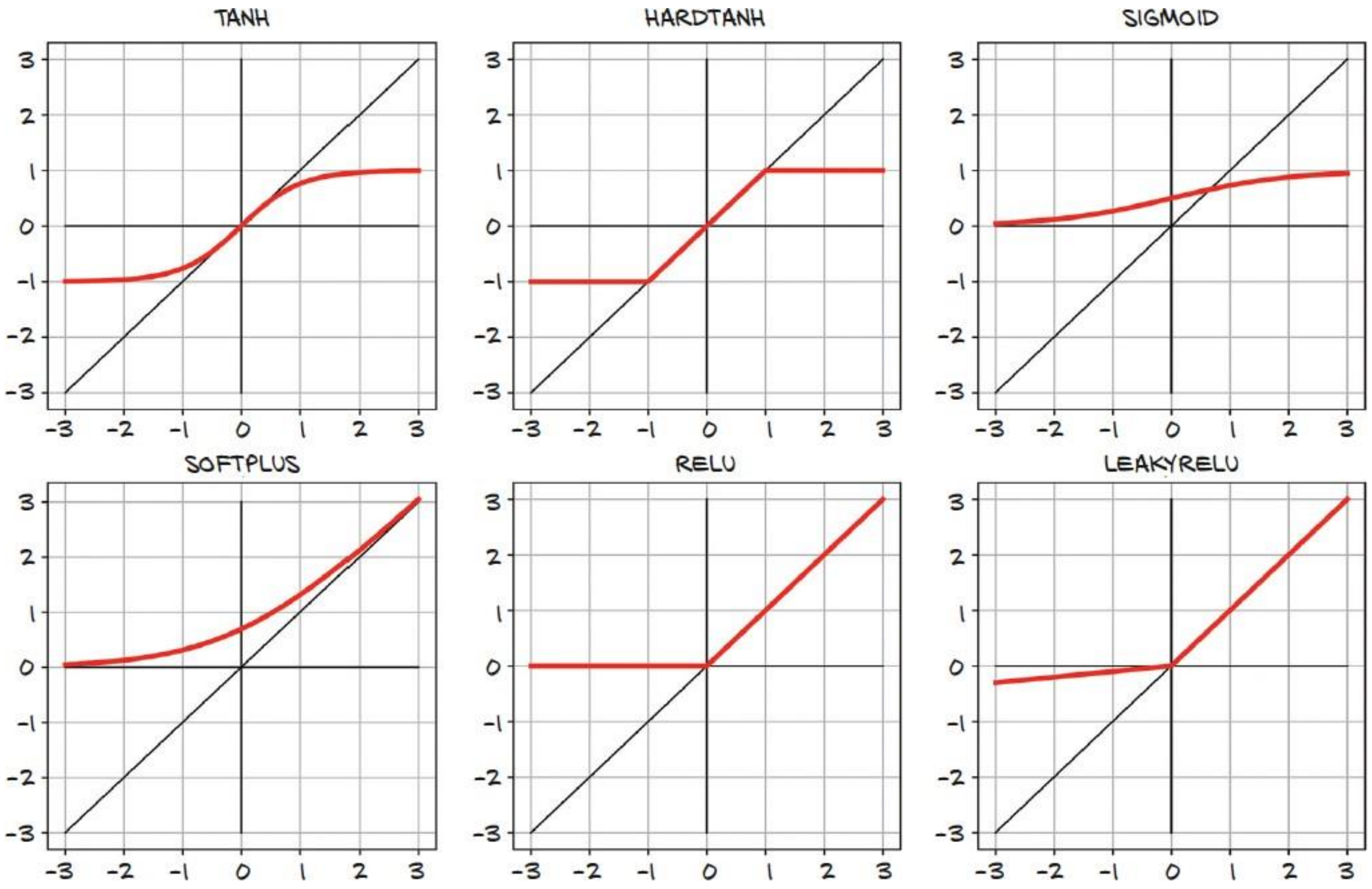
$$\tanh'(x) = 1-\tanh(x)^2$$

$$\mathrm{relu}(x) = \max(0,x)$$

$$\mathrm{relu}'(x) = 1_{x>0}$$

Recap

Training neural networks



Recap

Training neural networks

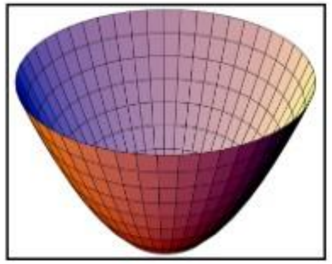
Largest difference between simple ML models and neural networks is:

– Nonlinearity of neural network causes interesting loss functions to be non-convex

Logistic Regression Loss:

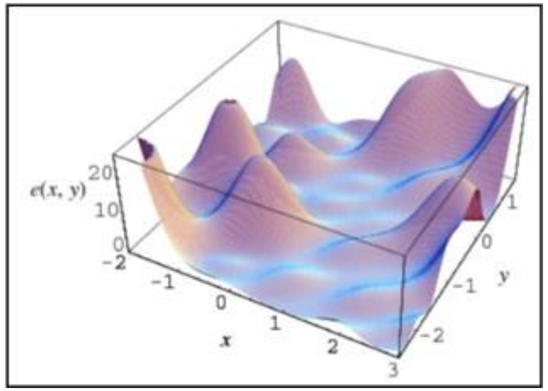
Linear Regression with Basis Functions:

$$E_D(\mathrm{w}) = \frac{1}{2}\sum_{n=1}^{N}\{ t_n - \mathrm{w}^T\varphi(x_n) \}^2$$



Neural Network Loss:

$$J(\boldsymbol{\theta}) = -E_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$



Use iterative gradient-based optimizers that merely drives cost to low value, rather than
- Exact linear equation solvers used for linear regression or
- convex optimization algorithms used for logistic regression or SVMs

Motivation

Notation

- Parameters:

$$\theta \in \mathrm{R}^{n}$$

dimension

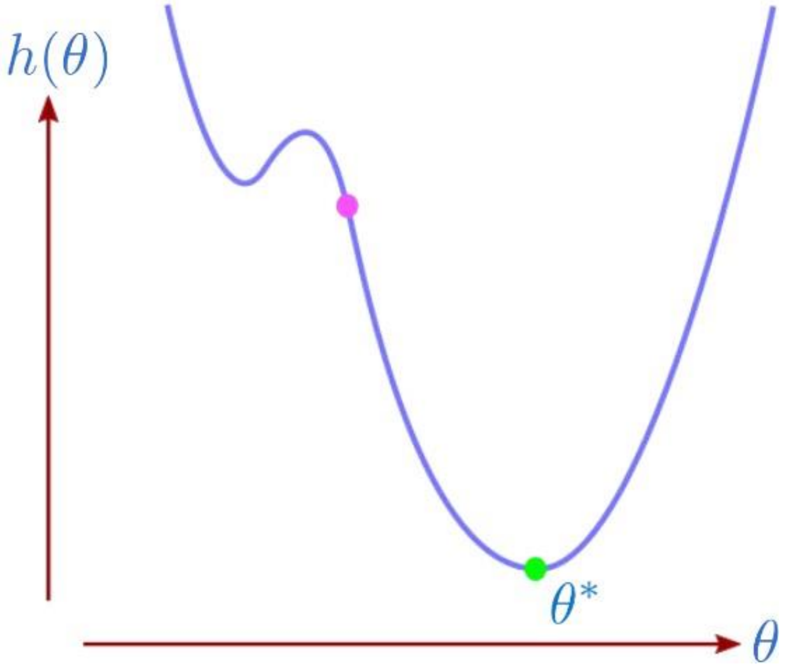
- Real-valued objective function :

$$h(\theta)$$

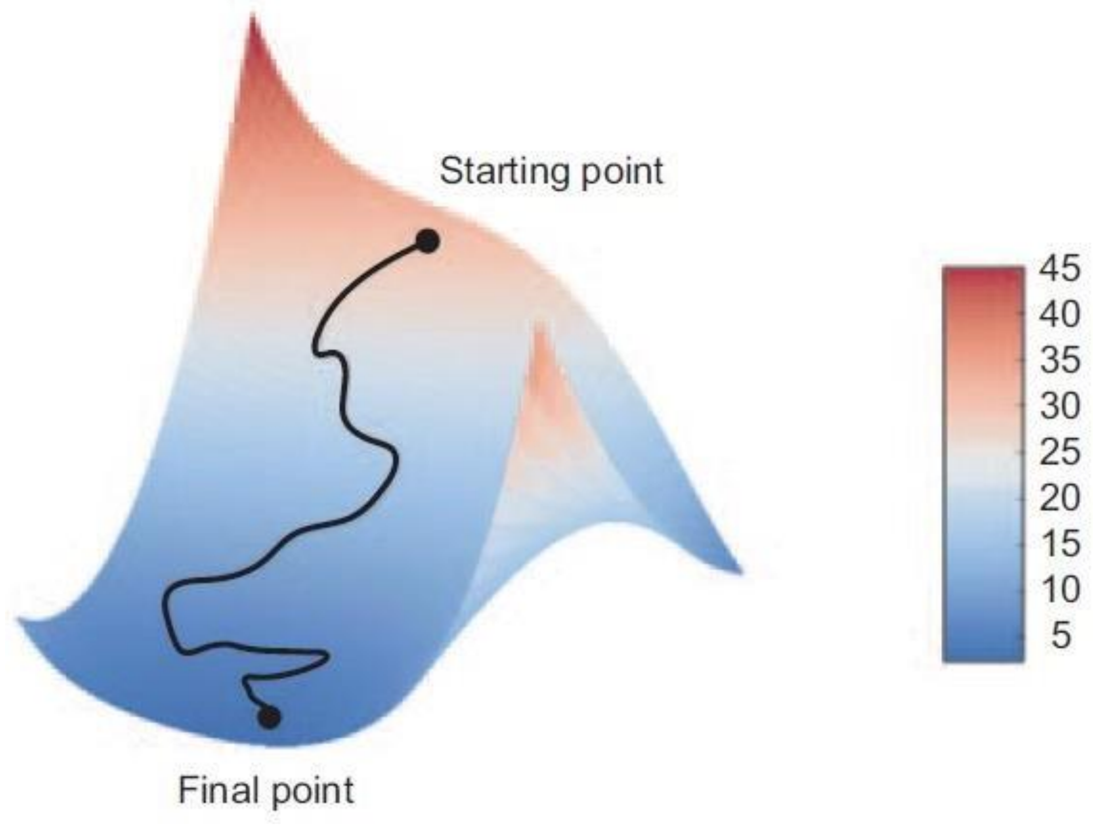
- Goal of optimization:

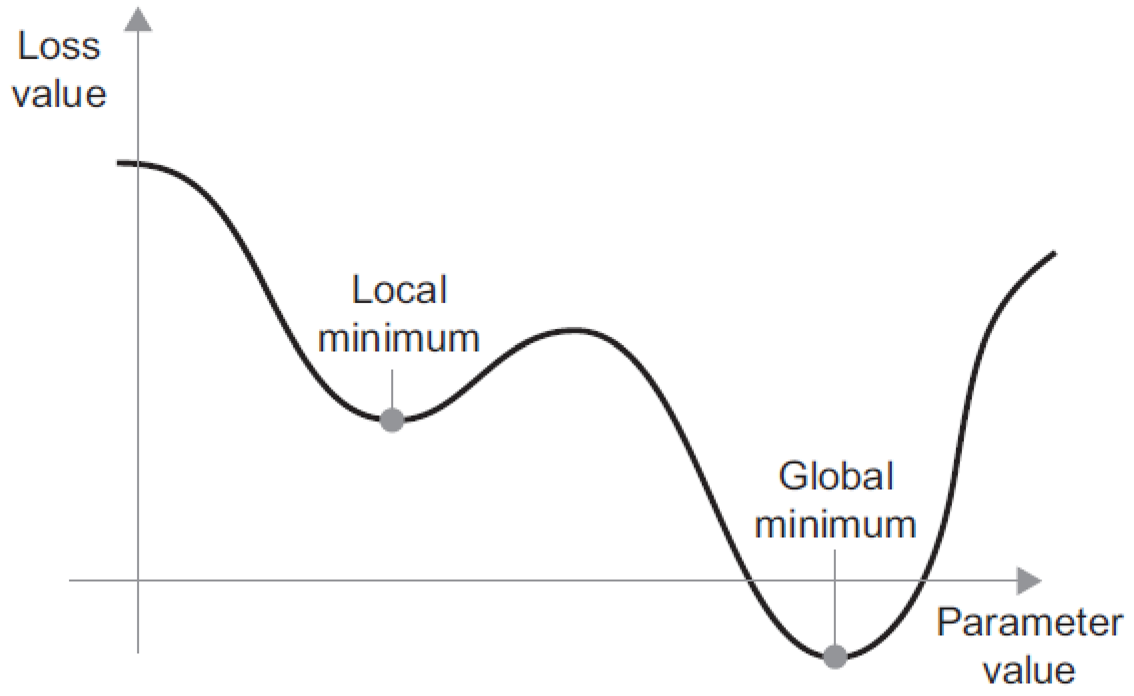
$$\theta^* = \arg\min_{\theta} h(\theta)$$

1D example objective function



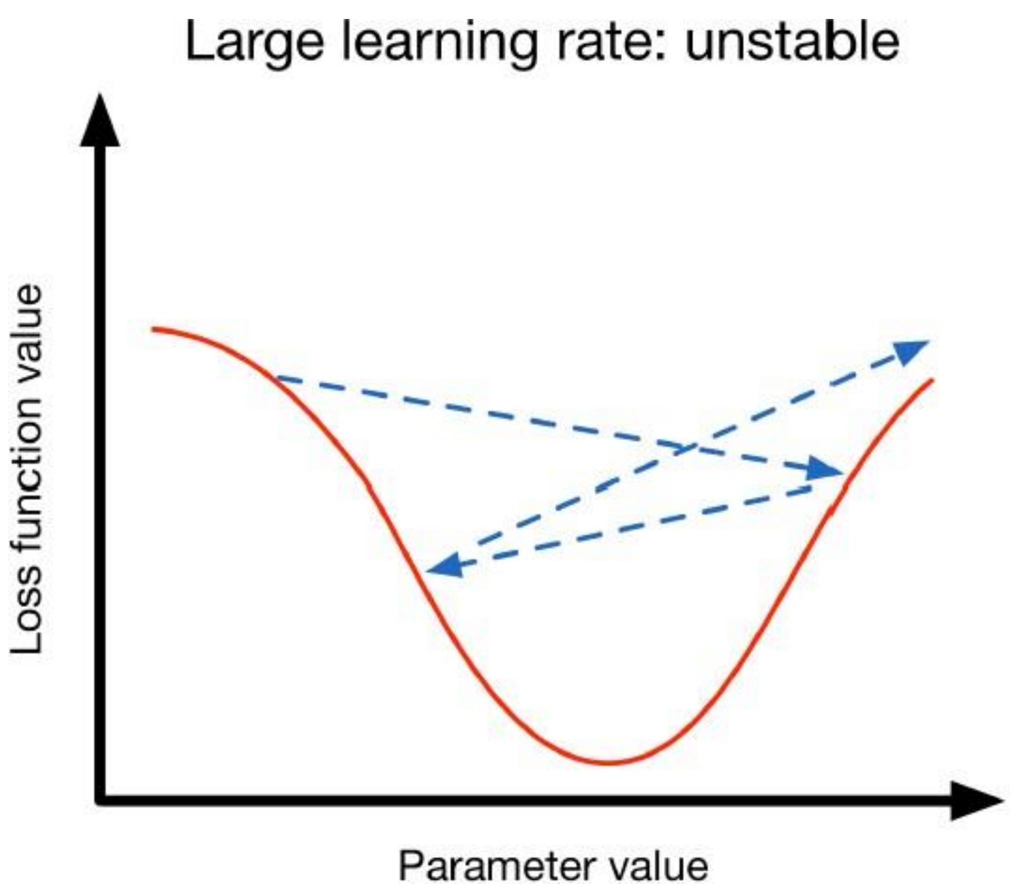
Motivation
Notation

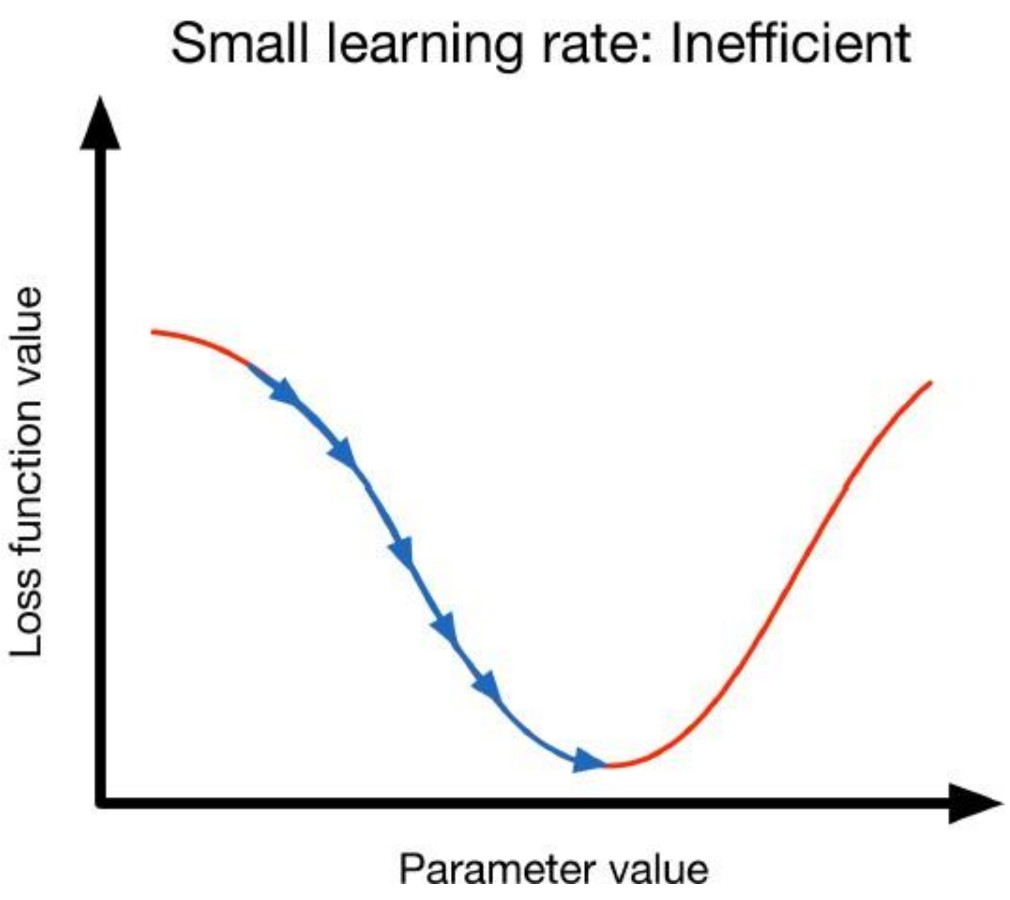




Motivation

Notation





Motivation

NN training objective

- The standard neural network training objective is given by:

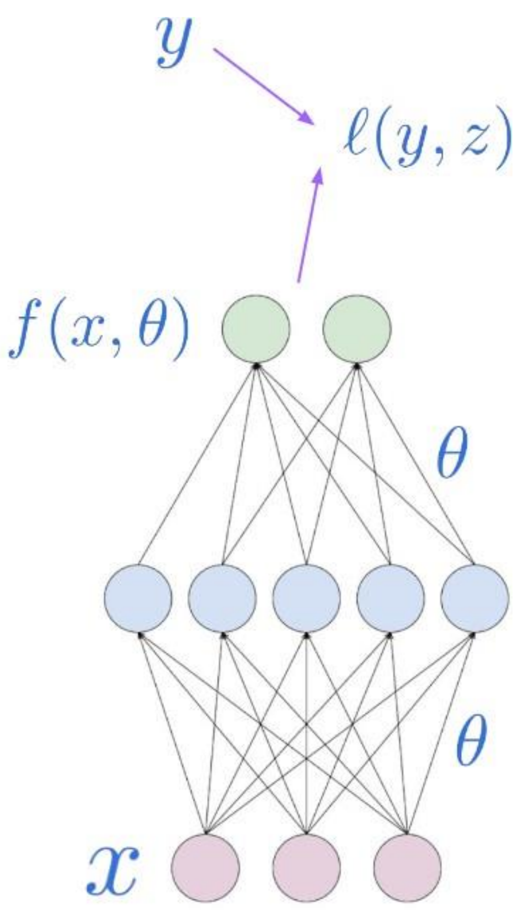
$$h(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(y_i, f(x_i, \theta))$$

where:

$\ell(y, z)$ is a loss function measuring disagreement between y and z

and

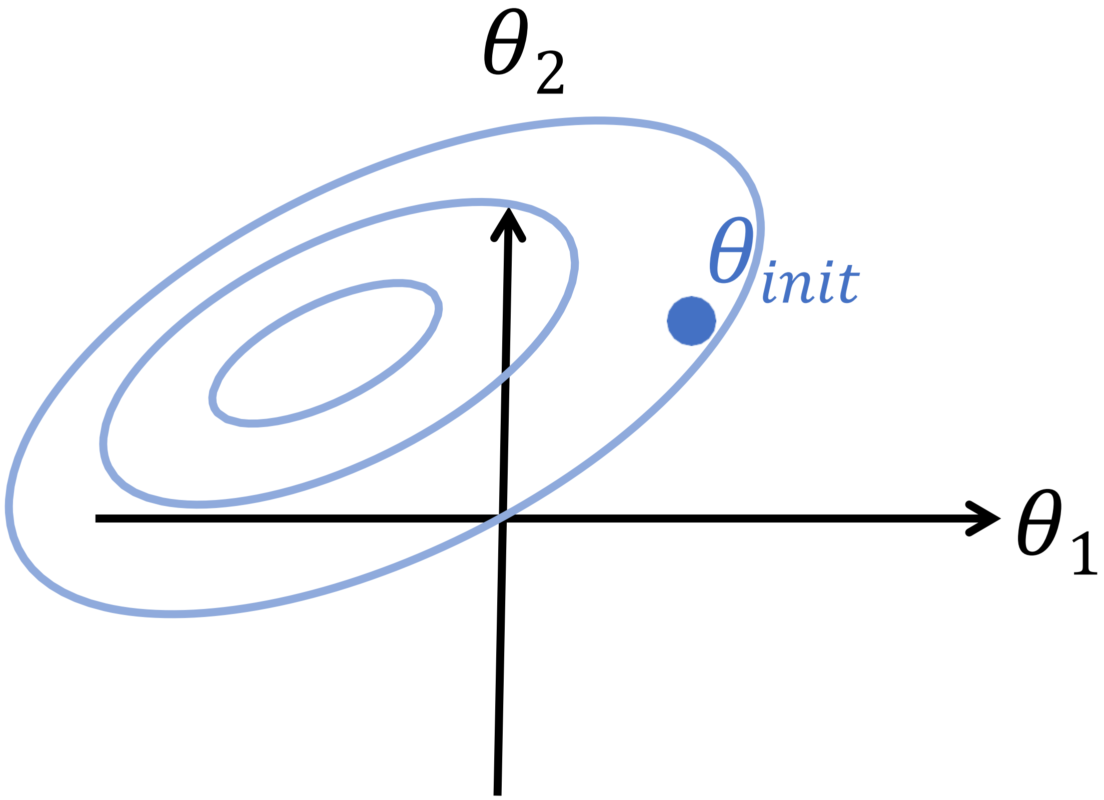
$f(x, \theta)$ is a neural network function taking input x and outputing some prediction



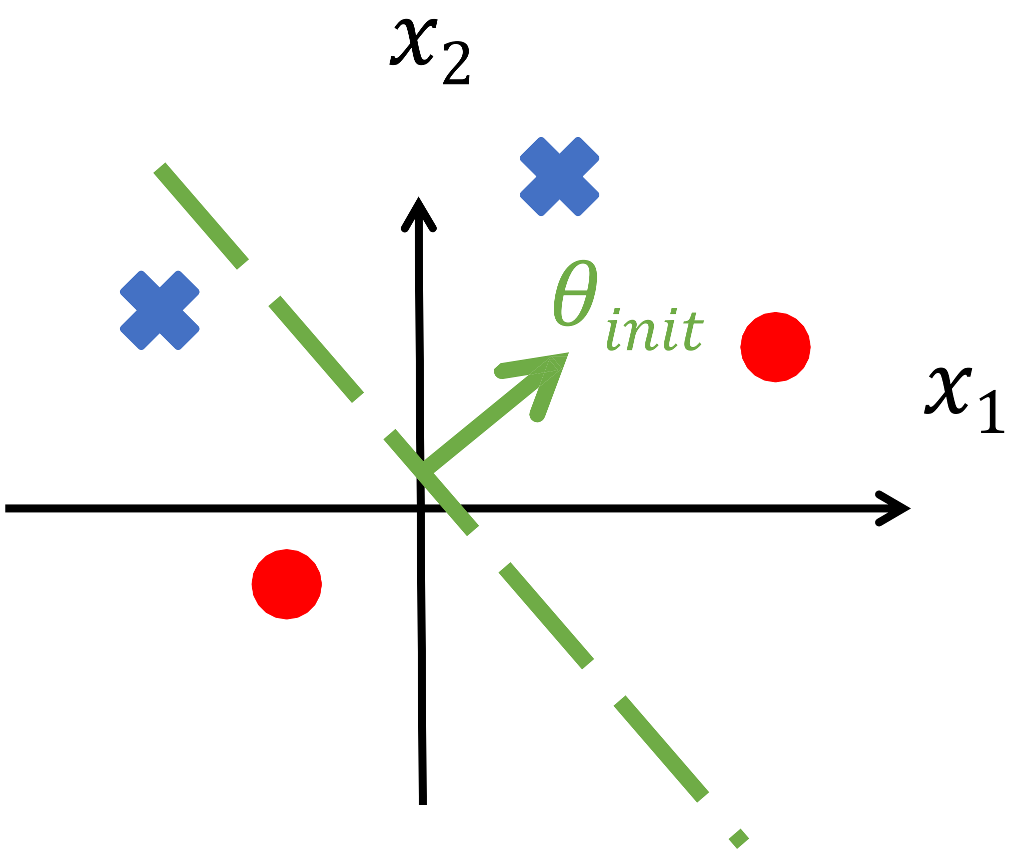
Motivation

Landscape of the training objective

Parameter space



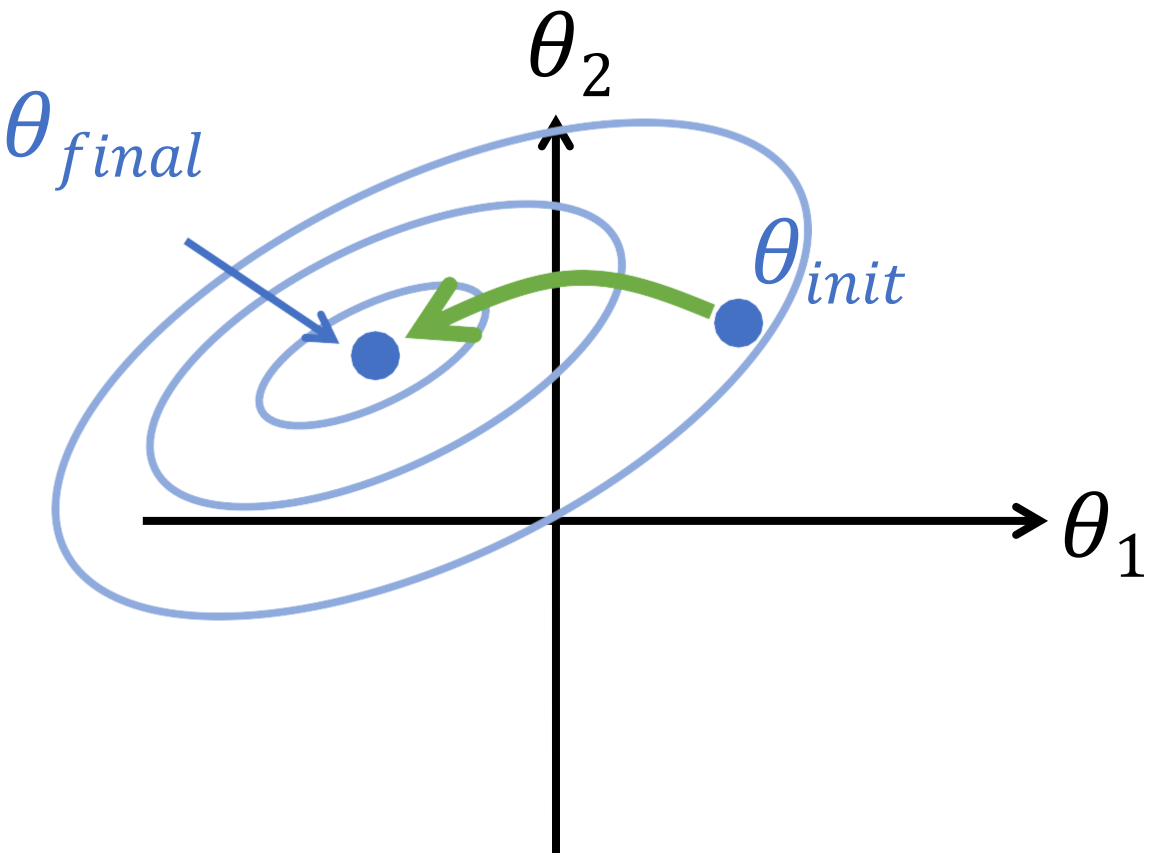
Example space



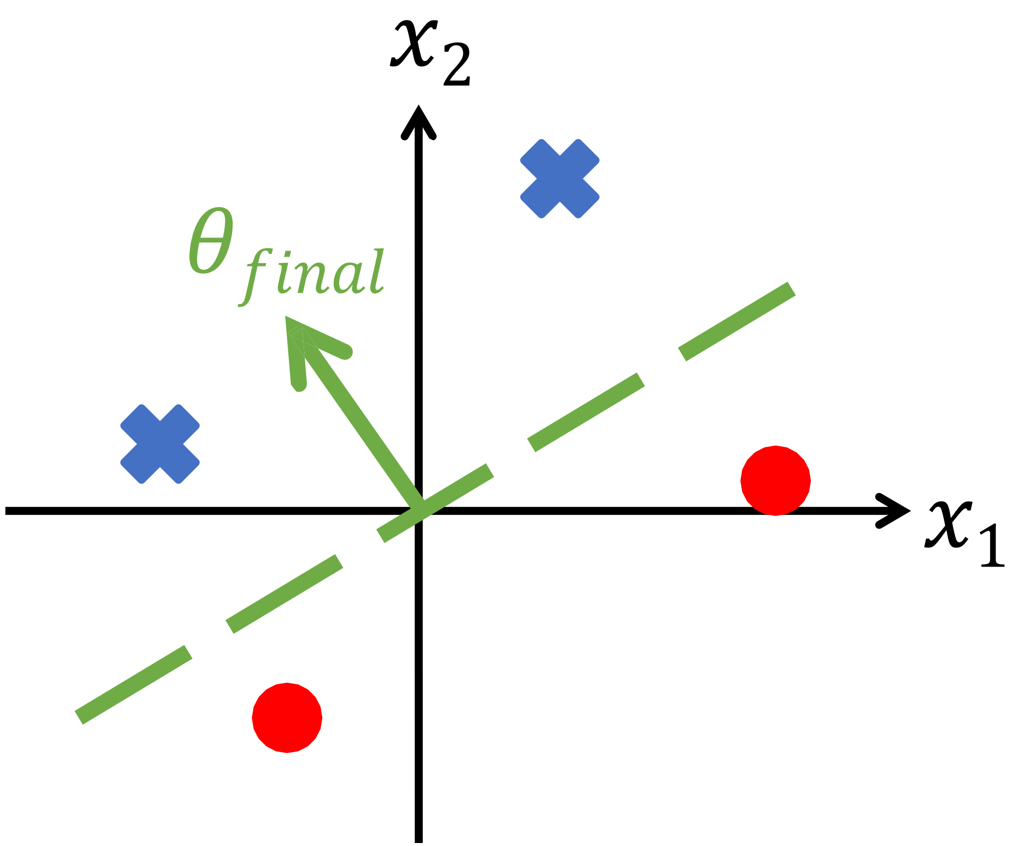
Motivation

Landscape of the training objective

Parameter space



Example space



Gradient Descent

Definition

- Basic gradient descent iteration:

$$\theta_{k+1} = \theta_k - \alpha_k \nabla h(\theta_k)$$

Learning rate: α_k

(aka "step size")

Gradient: $\nabla h(\theta) = \begin{bmatrix} \frac{\partial h(\theta)}{\partial [\theta]_1} \\ \frac{\partial h(\theta)}{\partial [\theta]_2} \\ \vdots \\ \frac{\partial h(\theta)}{\partial [\theta]_n} \end{bmatrix}$

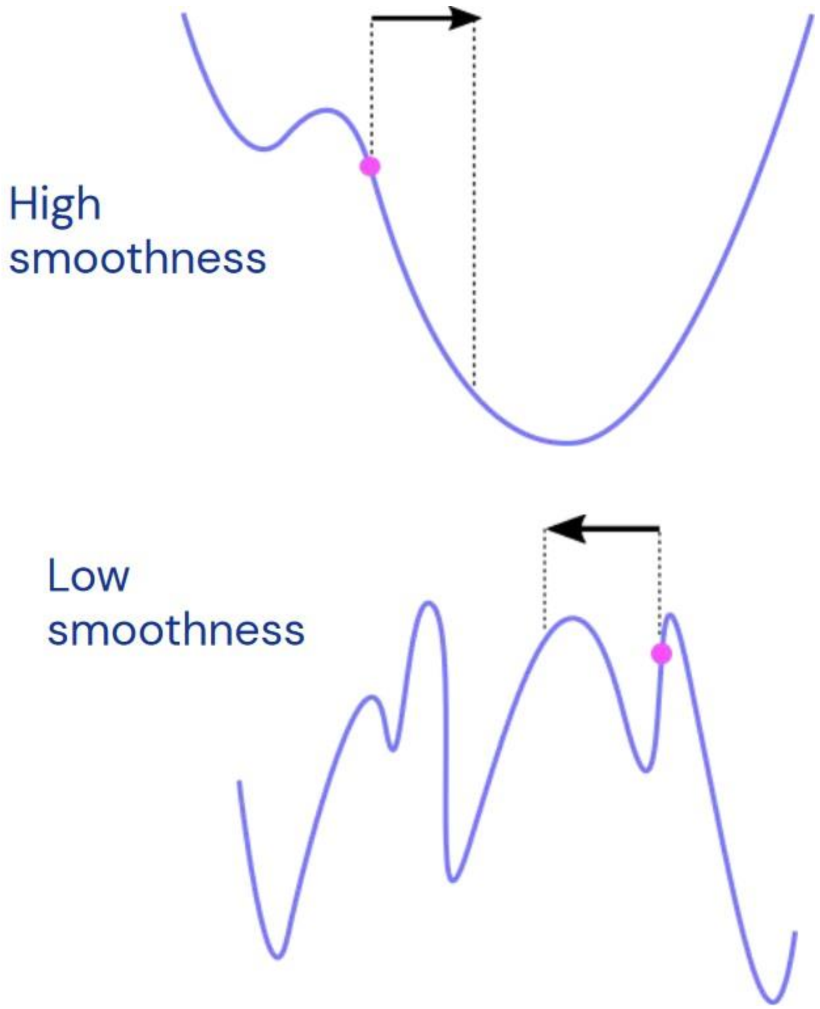
Gradient Descent

Intuition: Steepest Descent

$$\theta_{k+1} = \theta_k - \alpha_k \nabla h(\theta_k)$$

- Gradient direction $\nabla h(\theta)$ gives *greatest* reduction in $h(\theta)$ per unit of change* in θ.

- If $h(\theta)$ is "sufficiently smooth", and learning rate small, gradient will keep pointing down-hill over the region in which we take our step



Gradient Descent

Minimizing a local approximation

- 1st-order Taylor series for $h(\theta)$ around current θ is:

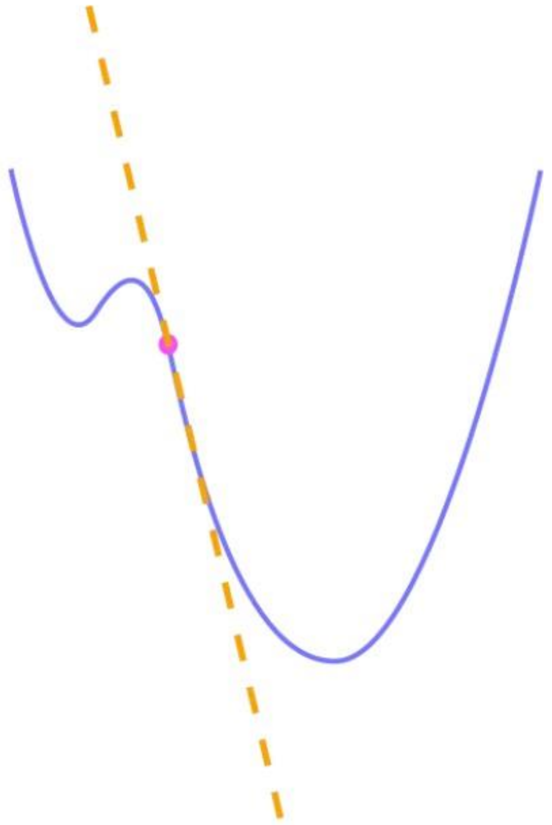
$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d$$

- For small enough d this will be a reasonable approximation
- Gradient update computed by minimizing this within a sphere of radius r:

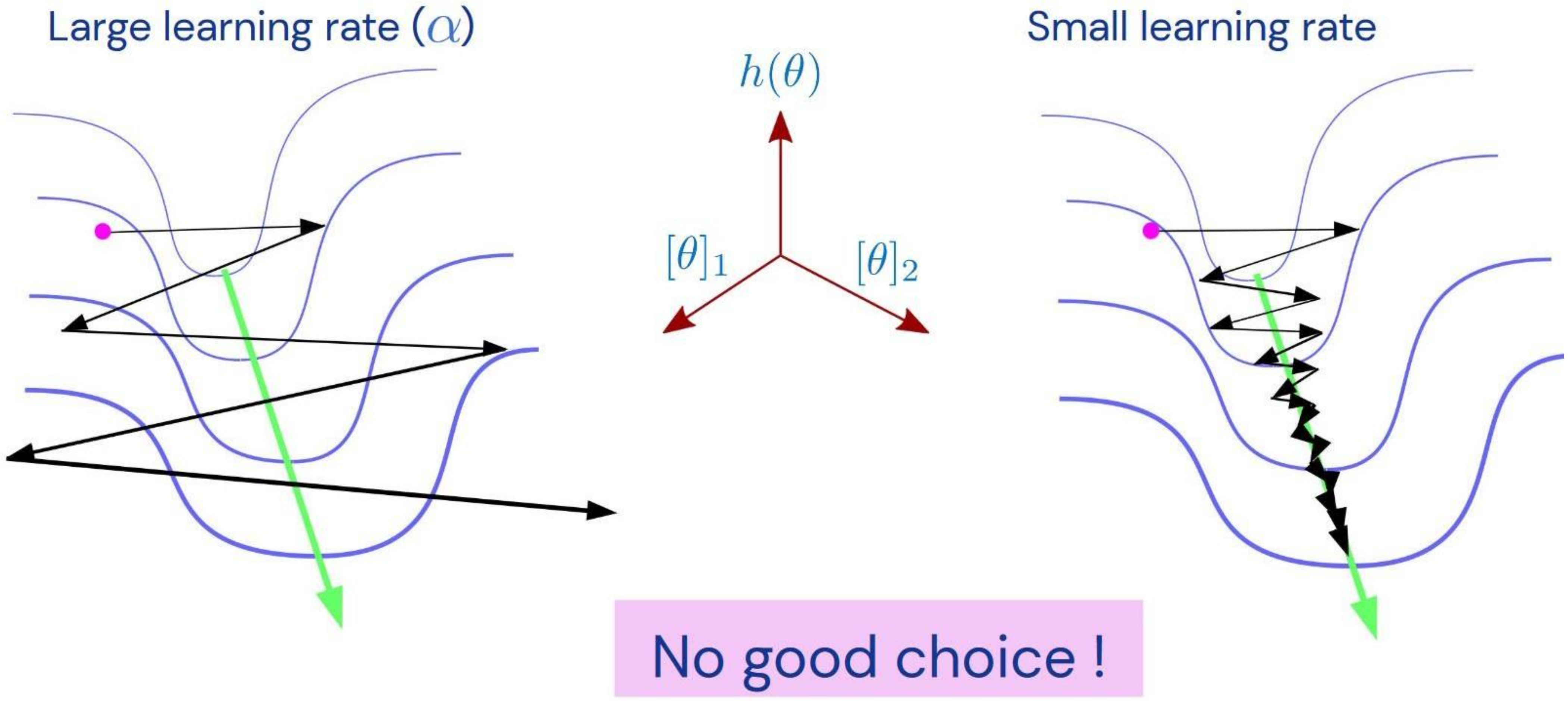
$$-\alpha \nabla h(\theta) = \arg\min_{d:\|d\| \le r} \left(h(\theta) + \nabla h(\theta)^\top d \right)$$

where

$$r = \alpha \|\nabla h(\theta)\|$$



Gradient Descent
Problems and Limitations



Gradient Descent
Technical Assumptions

- $h(\theta)$ has Lipschitz continuous derivatives (i.e. is "Lipschitz smooth"):

$$\|\nabla h(\theta) - \nabla h(\theta')\| \leq L\|\theta - \theta'\|$$ (an **upper bound** on the curvature)

- $h(\theta)$ is strongly convex (perhaps only near minimum):

$$h(\theta + d) \geq h(\theta) + \nabla h(\theta)^\top d + \frac{\mu}{2}\|d\|^2$$ (a **lower bound** on the curvature)

- And *for now*: Gradients are computed exactly (i.e. **not** stochastic)

Gradient Descent

Convergence Theory:Upper Bounds

If previous conditions hold and we take $\alpha_k = \dfrac{2}{L+\mu}$:

$$h(\theta_k) - h(\theta^*) \leq \frac{L}{2}\left(\frac{\kappa - 1}{\kappa + 1}\right)^{2k} \|\theta_0 - \theta^*\|^2$$

minimizer

where $\kappa = L/\mu$.

Number of iterations to achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$ is

$$k \in \mathcal{O}\left(\kappa \log \frac{1}{\epsilon}\right)$$

Gradient Descent

Convergence Theory: useful in practice?

- Issues with bounds such as this one:
  - too pessimistic (they must cover worst-case examples)
  - some assumptions too strong (e.g. convexity)
  - other assumptions too weak (real problems have additional useful structure)
  - rely on crude measures of objective (e.g. condition numbers)
  - usually focused on asymptotic behavior
- The design/choice of an optimizer should always be informed by **practice** more than anything else. But theory can help guide the way and build intuitions.

Momentum Methods

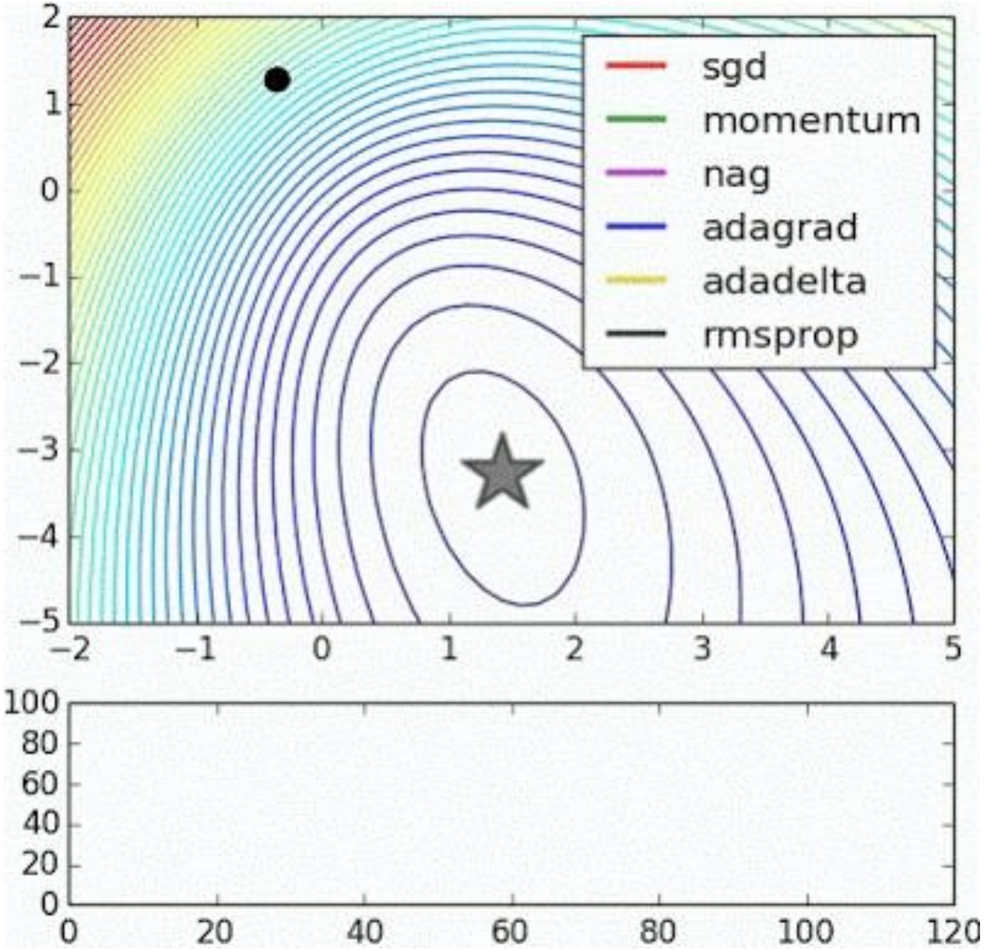
Motivation

- Motivation:
  - the gradient has a tendency to flip back and forth as we take steps when the learning rate is large
  - e.g. the narrow valley example
- The key idea:
  - accelerate movement along directions that point consistently down-hill across many consecutive iterations (i.e. have low curvature)
- How?
  - treat current solution for θ like a “ball” rolling along a “surface” whose height is given by $h(\theta)$, subject the force of gravity

Momentum Methods
Motivation

Momentum Methods
Motivation

- How to update the weights based on the loss function
- *Learning rate (+scheduling)*
- Stochastic gradient descent, momentum, and their variants
  - RMSProp is usually a good first choice
  - More info:
  - http://ruder.io/optimizing-gradient-descent/



Momentum Methods

Mathematical Formulation

- Classical Momentum:

$$v_{k+1} = \eta_k v_k - \nabla h(\theta_k) \qquad v_0 = 0$$

$$\theta_{k+1} = \theta_k + \alpha_k v_{k+1}$$

Learning rate: α_k

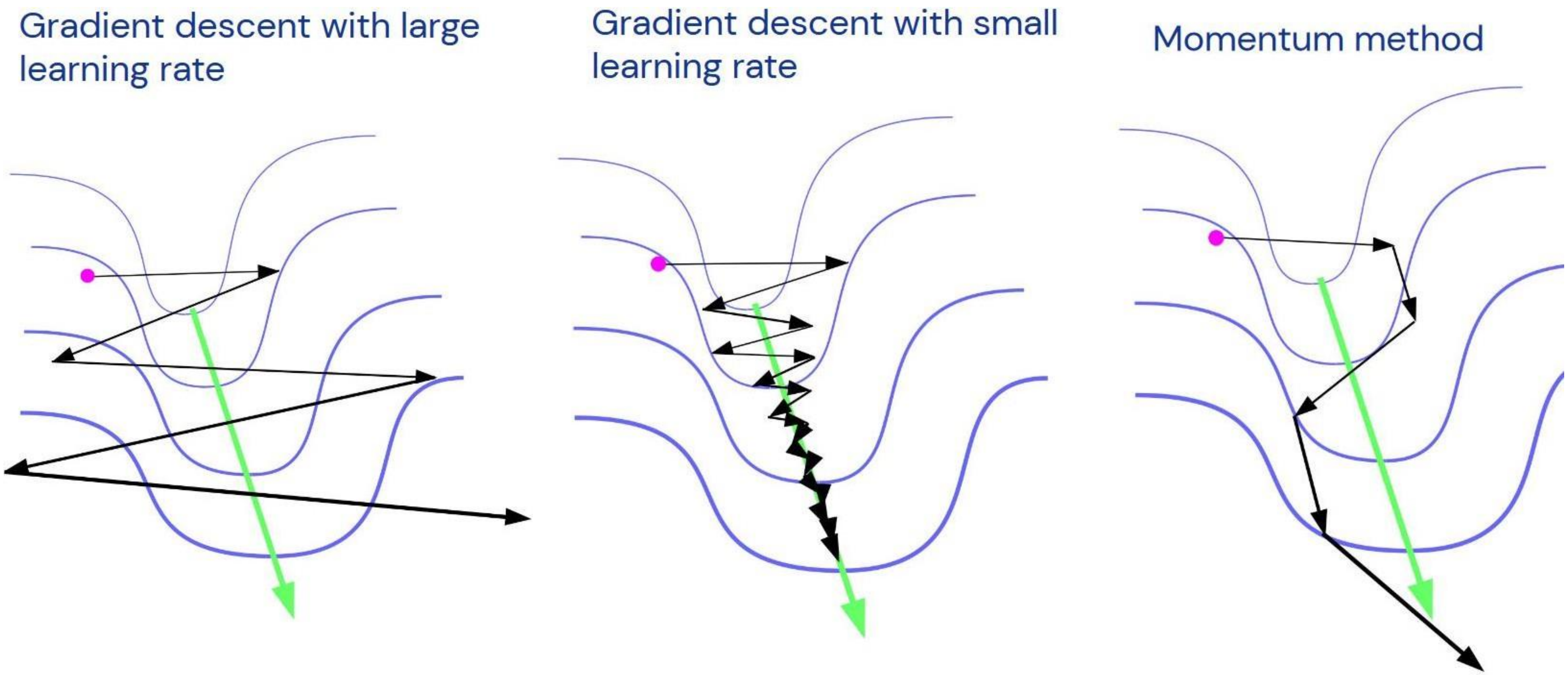
Momentum constant: η_k

- Nesterov's variant:

$$v_{k+1} = \eta_k v_k - \nabla h(\theta_k + \alpha_k \eta_k v_k) \qquad v_0 = 0$$

$$\theta_{k+1} = \theta_k + \alpha_k v_{k+1}$$

Momentum Methods
Narrow 2D Valley Revisited



Momentum Methods

Upper Bounds

Given objective $h(\theta)$ satisfying same technical conditions as before, and careful choice of α_k and η_k, Nesterov's momentum method satisfies:

$$h(\theta_k) - h(\theta^*) \leq L \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa}} \right)^k \|\theta_0 - \theta^*\|^2 \qquad \kappa = \frac{L}{\mu}$$

Number of iterations to achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$:

$$k \in \mathcal{O}\left(\sqrt{\kappa} \log \frac{1}{\epsilon} \right)$$

Momentum Methods

1st order methods and lower bounds

- A **first-order method** is one where updates are linear combinations of observed gradients. i.e.:

$$\theta_{k+1} - \theta_k = d \in \mathrm{Span}\{\nabla h(\theta_0), \nabla h(\theta_1), \ldots, \nabla h(\theta_k)\}$$

- Included:
  - gradient descent
  - momentum methods
  - conjugate gradients (CG)
- Not included:
  - preconditioned gradient descent / 2nd-order methods

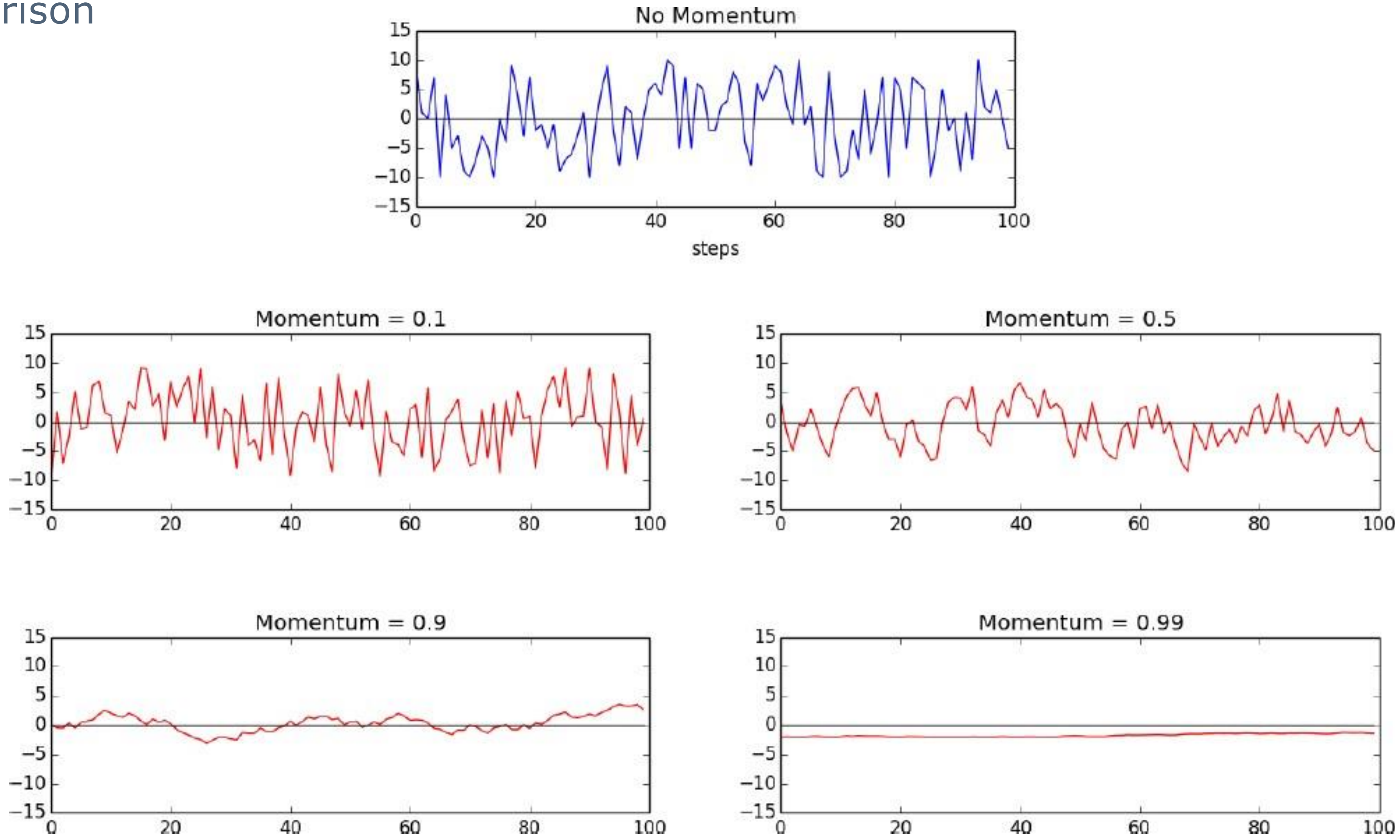
Momentum Methods

Comparison

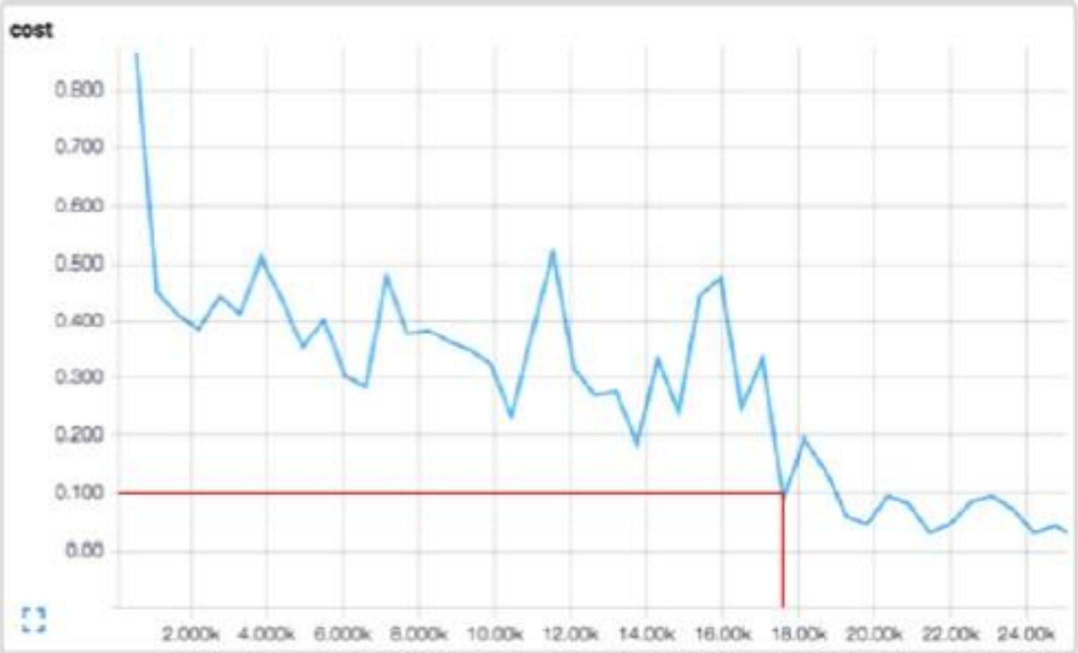
To achieve $h(\theta_k) - h(\theta^*) \leq \epsilon$ the number of iterations k satisfies:

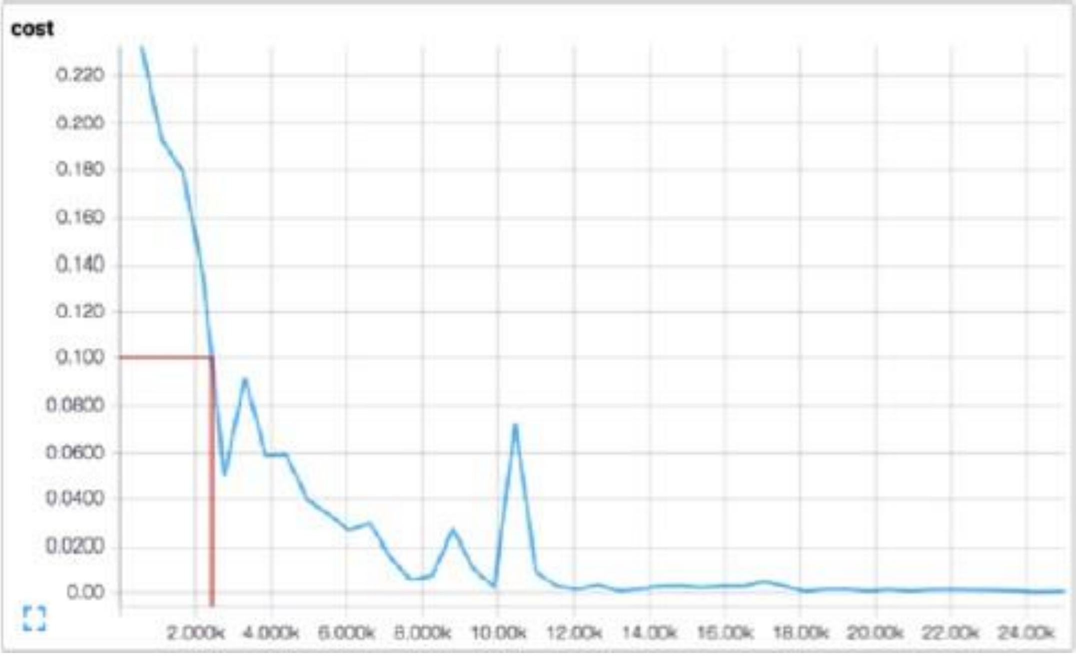
- (Worst-case) lower bound for 1st-order methods: $k \in \Omega\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$
- Upper bound for gradient descent: $k \in \mathcal{O}\left(\kappa\log\frac{1}{\epsilon}\right)$
- Upper bound for GD w/ Nesterov's momentum: $k \in \mathcal{O}\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$

Momentum Methods
Comparison



Momentum Methods
Comparison





2nd Order Methods

Limitations of 1st order methods

- For any 1st-order method, the number of steps needed to converge grows with "condition number":

$$\kappa = \frac{L}{\mu}$$

Max curvature

Min curvature

- This will be very large for some problems (e.g. certain deep architectures)
- 2nd-order methods can improve (or even eliminate) this dependency

2^{nd} Order Methods

Derivation of Newton's Method

- Approximate $h(\theta)$ by its 2nd-order Taylor series around current θ:

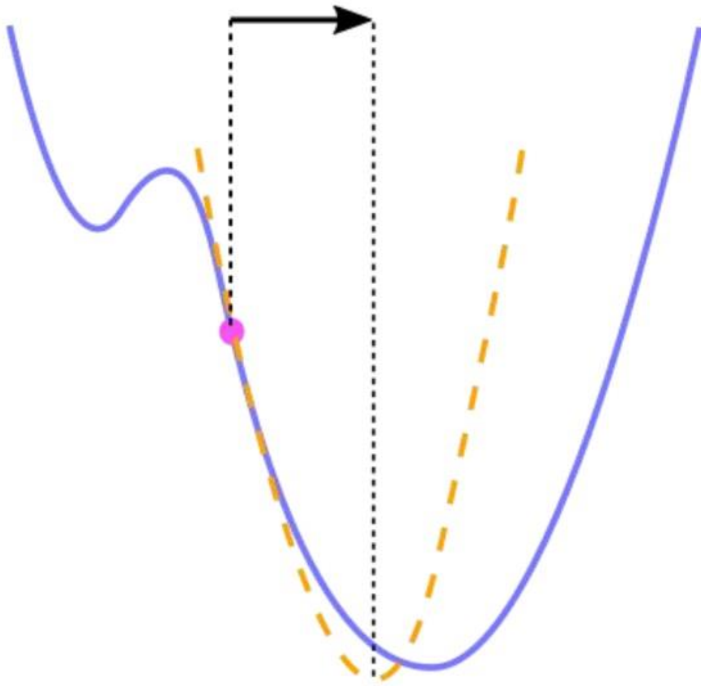
$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d$$

- Minimize this local approximation to obtain:

$$d = -H(\theta)^{-1} \nabla h(\theta)$$

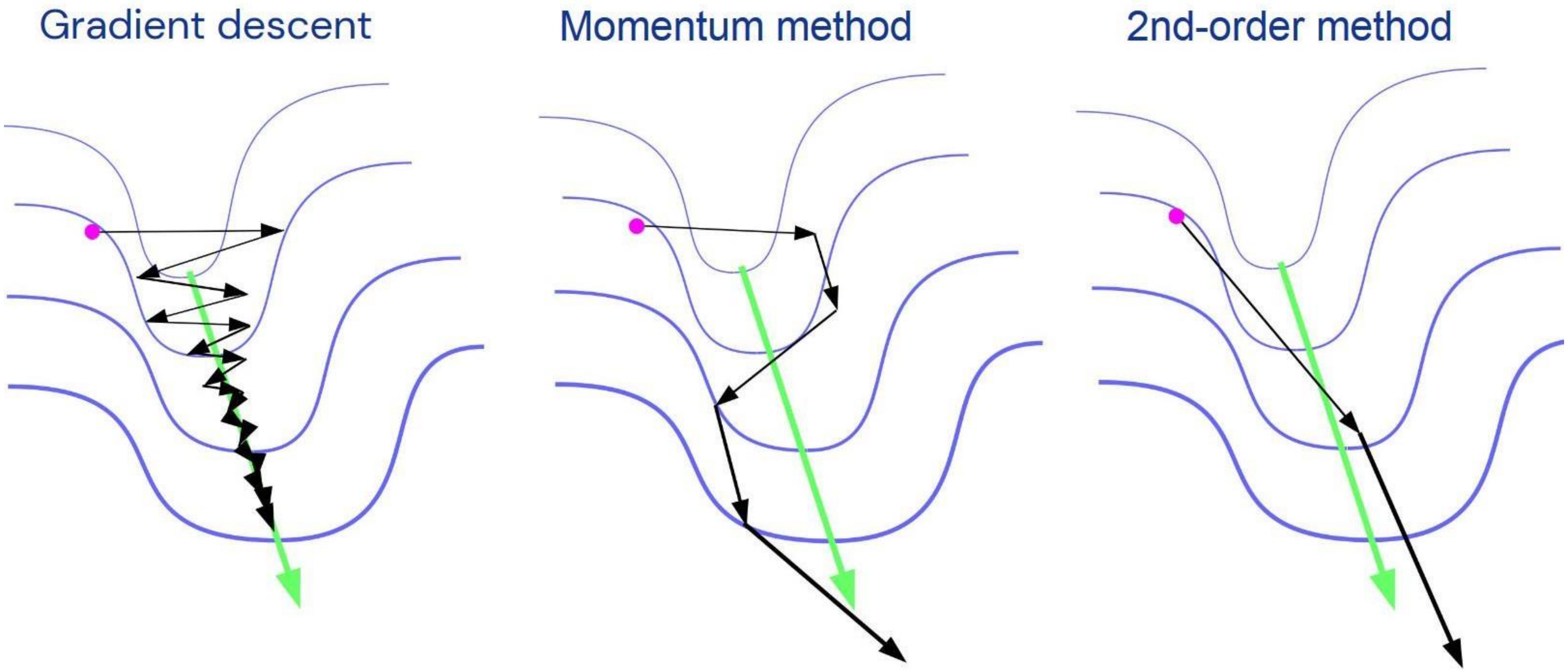
- Update current iterate with this:

$$\theta_{k+1} = \theta_k - H(\theta)^{-1} \nabla h(\theta_k)$$



2^{nd} Order Methods

2D Narrow Valley Revisited (again)



2^{nd} Order Methods

Comparison to Gradient Descent

- Maximum allowable global learning rate for GD to avoid divergence:

$$\alpha = 1/L$$

L is maximum curvature aka "Lipschitz constant"

- Gradient descent implicitly minimizes a bad approximation of 2nd-order Taylor series:

$$h(\theta + d) \approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d$$

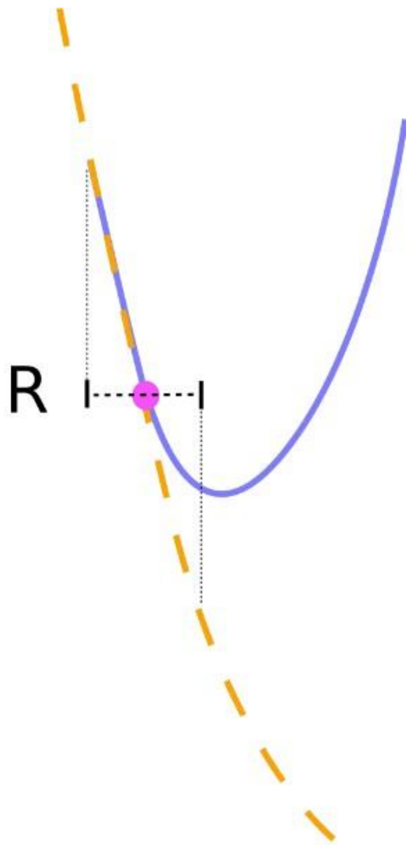
$$\approx h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top (LI) d$$

- LI is too pessimistic / conservative an approximation of $H(\theta)$! Treats all directions as having max curvature.

2nd Order Methods

Local Quadratic Approximation

- Quadratic approximation of objective is only trustworthy in a local region around current θ
- Gradient descent (implicitly) approximates the curvature everywhere by its global max (and so doesn't have this problem)
- Newton's method uses $H(\theta)$, which may become an underestimate in the region we are taking our update step



Solution: Constrain update d to lie in a "trust region" R around, where approximation remains "good enough"

2nd Order Methods

Trust regions and "damping"

- If we take $R = \{d : \|d\|_2 \leq r\}$ then computing
$$\arg\min_{d \in R} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top H(\theta) d \right)$$
is often equivalent to
$$-(H(\theta) + \lambda I)^{-1} \nabla h(\theta) = \arg\min_{d} \left(h(\theta) + \nabla h(\theta)^\top d + \frac{1}{2} d^\top (H(\theta) + \lambda I) d \right)$$
for some λ.

- λ depends on r in a complicated way, but we can just work with λ directly

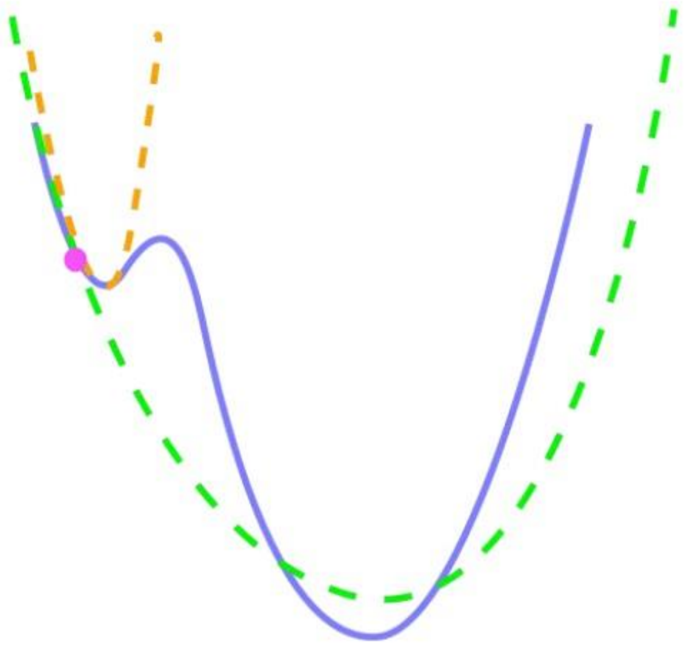
2^{nd} Order Methods

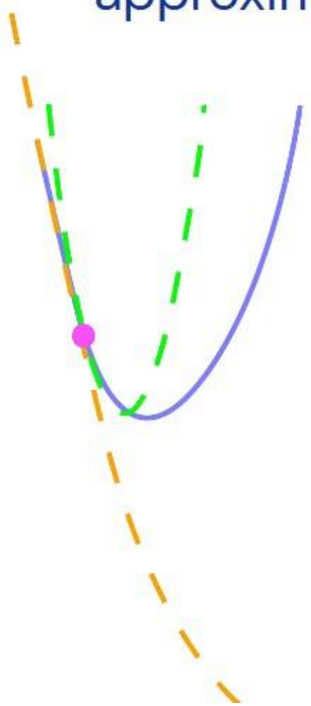
Alternative Curvature Matrices

$H(\theta)$ does not necessarily give the best quadratic approximation for optimization. Different replacements for $H(\theta)$ could produce:

A more global approximation

A more conservative approximation





2nd Order Methods

Alternative Curvature Matrices

- The most important family of related examples includes:
  - Generalized Gauss-Newton matrix (GGN)
  - Fisher information matrix
  - “Empirical Fisher”
- Nice properties:
  - always positive semi-definite (i.e. no negative curvature)
  - give parameterization invariant updates in small learning rate limit (unlike Newton’s method!)
  - work much better in practice for neural net optimization

2^{nd} Order Methods

Limitations

- For neural networks, $\theta \in \mathbb{R}^n$ can have 10s of millions of dimensions
- We simply cannot compute and store an $n \times n$ matrix, let alone invert it!
- To use 2nd-order methods, we must simplify the curvature matrix's
  - computation,
  - storage,
  - and inversion

This is typically done by approximating the matrix with a simpler form.

Stochastic Methods

Motivation

- Typical objectives in machine learning are an average over training cases of case-specific losses:

$$h(\theta) = \frac{1}{m}\sum_{i=1}^{m} h_i(\theta)$$

- m can be **very** big, and so computing the gradient gets expensive:

$$\nabla h(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla h_i(\theta)$$

Stochastic Methods

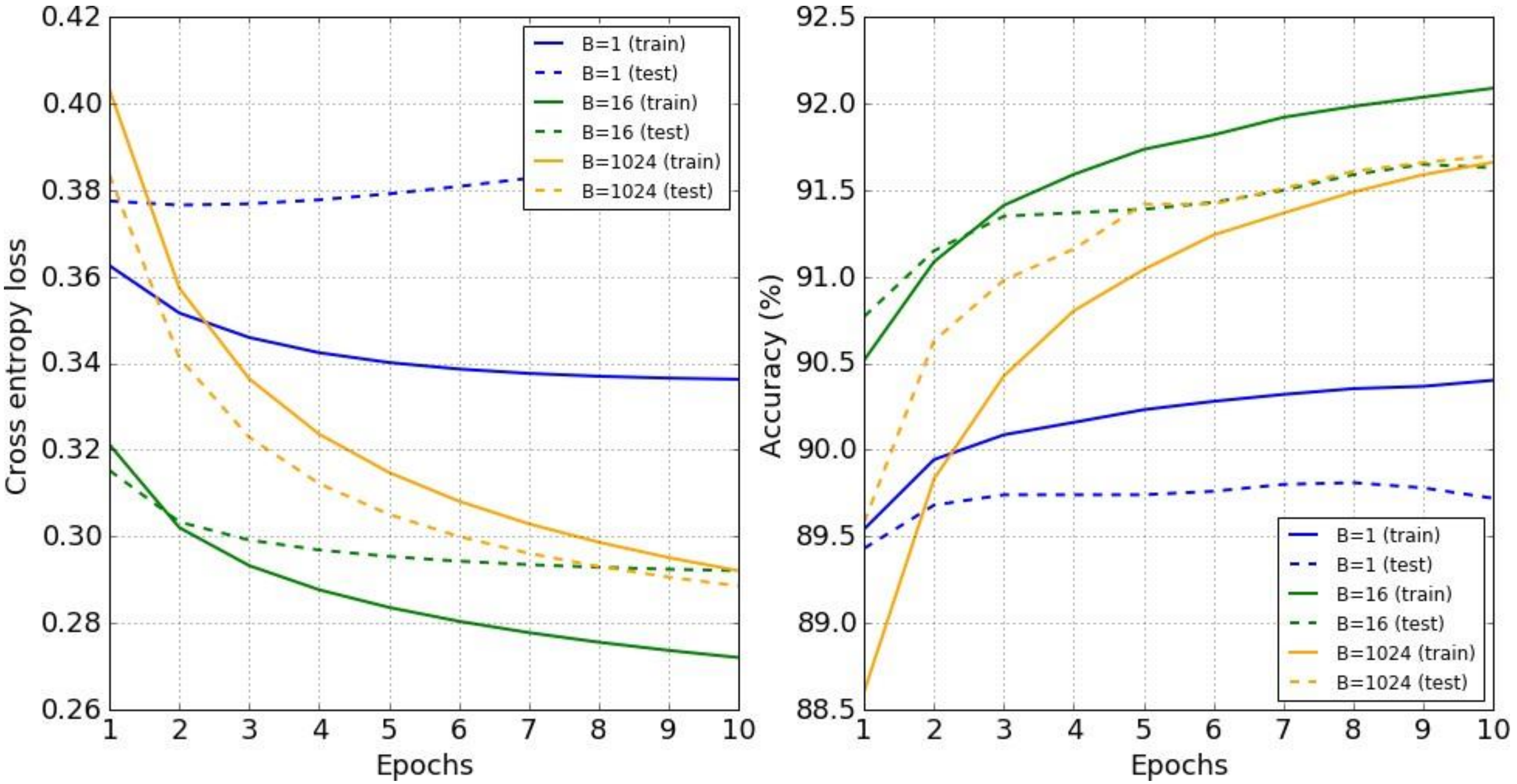
Mini Batching

- Fortunately there is often significant statistical overlap between $h_i(\theta)$'s
- Early in learning, when "coarse" features of the data are still being learned, most $\nabla h_i(\theta)$' s will look similar
- **Idea:** randomly subsample a "mini-batch" of training cases $S \subset \{1, 2, ..., m\}$ of size $b \ll m$, and estimate gradient as:

$$\widetilde{\nabla} h(\theta) = \frac{1}{b} \sum_{i \in S} \nabla h_i(\theta)$$

Stochastic Methods

Mini Batching



Stochastic Methods

Stochastic Gradient Descent

- Stochastic gradient descent (SGD) replaces $\nabla h(\theta)$ with its mini-batch estimate $\widetilde{\nabla} h(\theta)$, giving:

$$\theta_{k+1} = \theta_k - \alpha_k \widetilde{\nabla} h(\theta_k)$$

- To ensure convergence, need to do one of the following:
  - Decay learning rate: $\alpha_k = 1/k$
  - Use "Polyak averaging": $\bar{\theta}_k = \frac{1}{k+1} \sum_{i=0}^{k} \theta_i$ or $\bar{\theta}_k = (1-\beta)\theta_k + \beta\bar{\theta}_{k-1}$
  - Slowly increase the mini-batch size during optimization

Stochastic Methods
Convergence

- Stochastic methods converge slower than corresponding non-stochastic versions
- Asymptotic rate for SGD with Polyak averaging:

Gradient estimate covariance matrix

$$E[h(\theta_k)] - h(\theta^*) \in \frac{1}{2k} \operatorname{tr}\left(H(\theta^*)^{-1}\Sigma\right) + \mathcal{O}\left(\frac{1}{k^2}\right)$$

- Iterations to converge:

$$k \in \mathcal{O}\left(\operatorname{tr}\left(H(\theta^*)^{-1}\Sigma\right)\frac{1}{\epsilon}\right) \quad \text{vs} \quad k \in \mathcal{O}\left(\sqrt{\kappa}\log\frac{1}{\epsilon}\right)$$

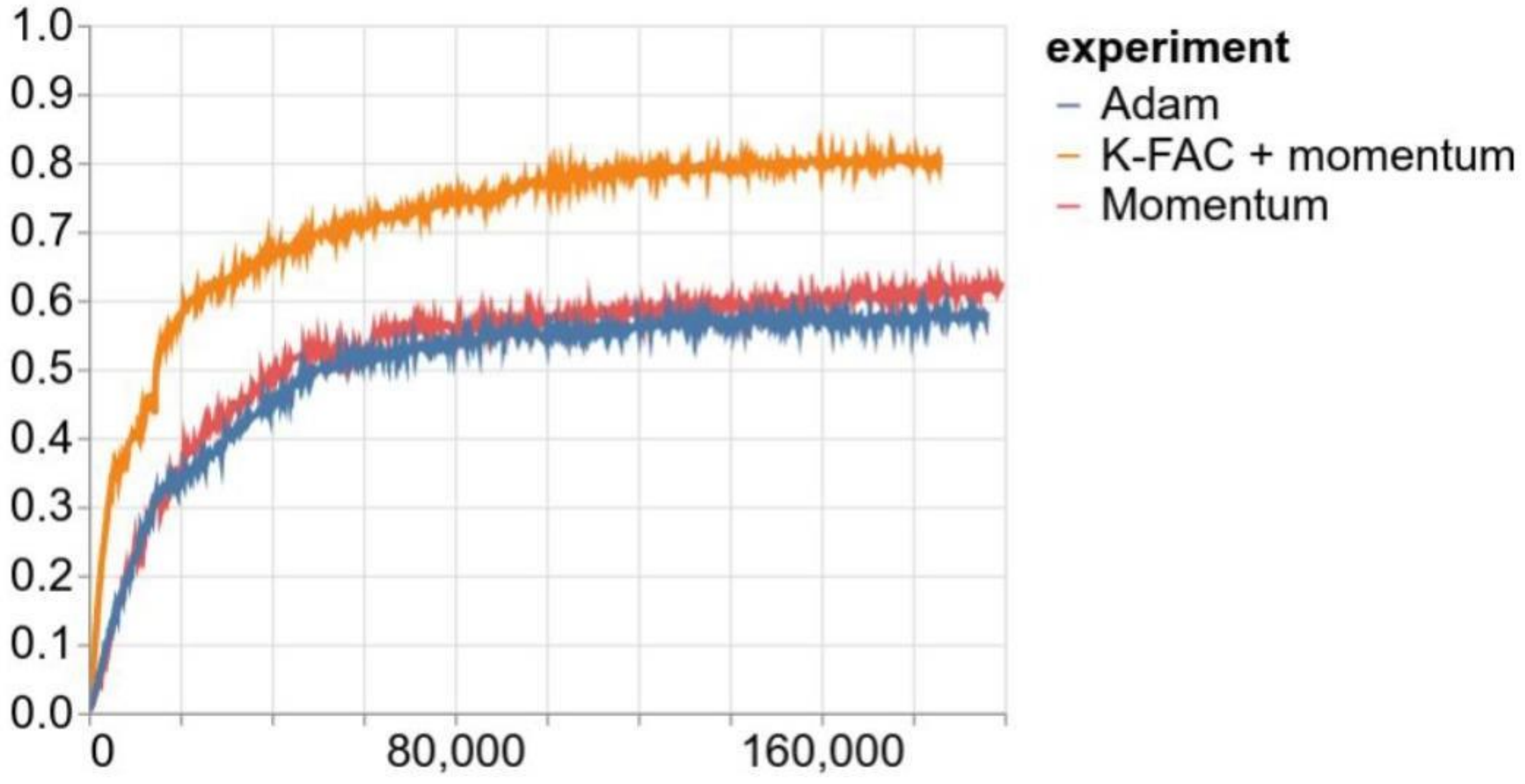
no log!

Stochastic Methods

Stochastic 2nd order and momentun methods

- Mini-batch gradients estimates can be used with 2nd-order and momentums methods too
- Curvature matrices estimated stochastically using decayed averaging over multiple steps
- No stochastic optimization method that sees the same amount of data can have better **asymptotic** convergence speed than SGD with Polyak averaging
- *But…* **pre-asymptotic** performance usually matters more in practice. So stochastic 2nd-order and momentum methods can still be useful if:
    - the loss surface curvature is bad enough and/or
    - the mini-batch size is large enough

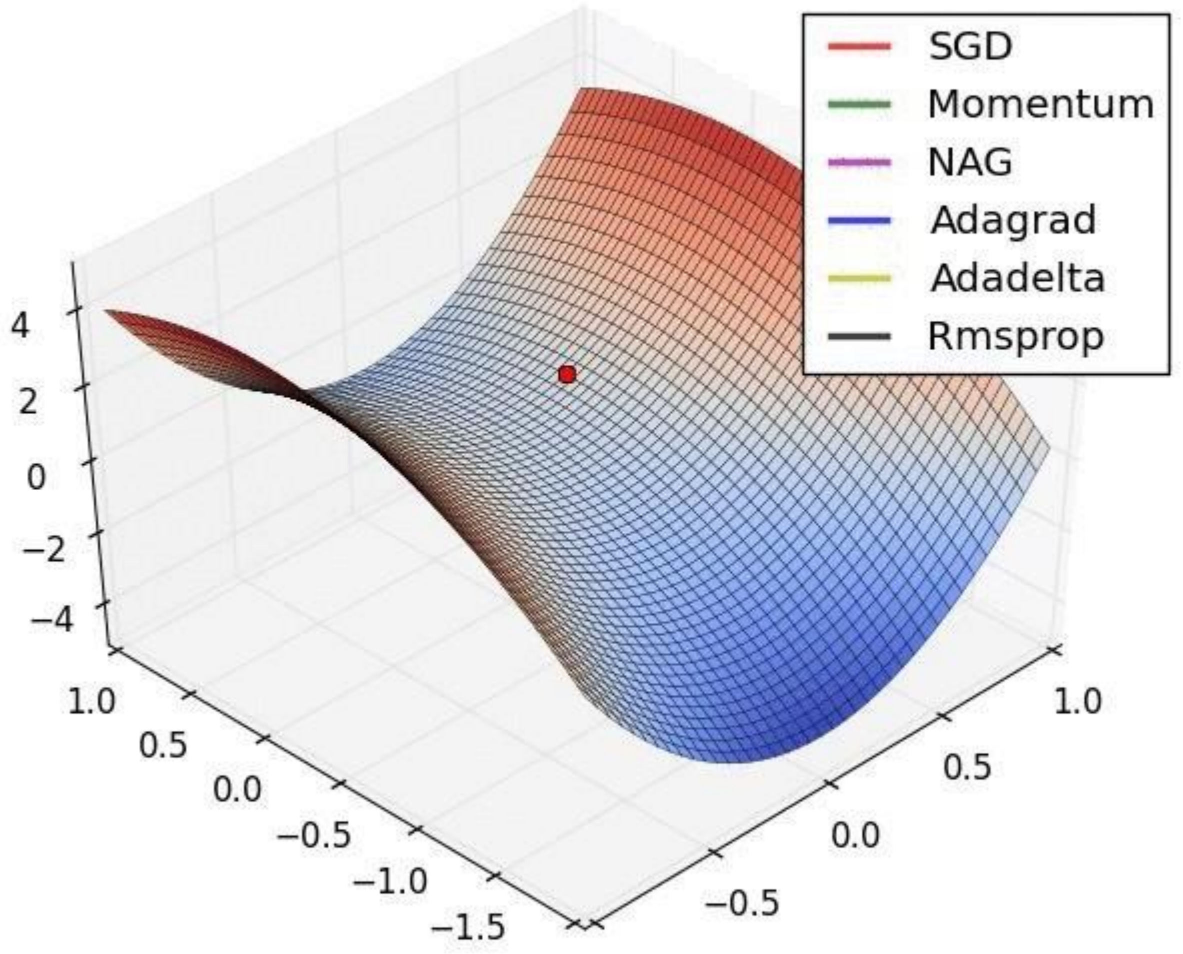
Stochastic Methods
Experiements on Deep Nets



Details

- Mini-batch size of 512
- Imagenet dataset
- 100 layer deep convolutional net without skips or batch norm
- Carefully initialized parameters

Stochastic Methods
Experiments on Deep Nets



Optimization

Summary

- Optimization methods:
  - enable learning in models by adapting parameters to minimize some objective
  - main engine behind neural networks
- 1st-order methods (gradient descent):
  - take steps in direction of “steepest descent”
  - run into issues when curvature varies strongly in different directions
- Momentum methods:
  - use principle of momentum to accelerate along directions of lower curvature
  - obtain “optimal” convergence rates for 1st-order methods

Optimization

Summary

- 2nd-order methods:
  - improve convergence in problems with bad curvature, even more so than momentum methods
  - require use of trust-regions/damping to work well
  - also require the use of curvature matrix approximations to be practical in high dimensions (e.g. for neural networks)
- Stochastic methods:
  - use "mini-batches" of data to estimate gradients
  - asymptotic convergence is slower
  - pre-asymptotic convergence can be sped up using 2nd-order methods and/or momentum