10-05/10-805: Machine Learning with Large Datasets

Fall 2022

Optimization for Deep Learning

Announcements

- HW4b due on Thursday
- Recitation this week will be a discussion on step size/learning rate tuning

Key course topics

Data preparation
- Data cleaning
- Data summarization
- Visualization
- Dimensionality reduction

Training
- Distributed ML
- **Large-scale optimization**
- **Scalable deep learning**
- Efficient data structures
- Hyperparameter tuning

Inference
- Hardware for ML
- Techniques for low-latency inference (compression, pruning, distillation)

Infrastructure / Frameworks
- Apache Spark
- TensorFlow
- AWS / Google Cloud / Azure

Advanced topics
- Federated learning
- Neural architecture search
- Productionizing ML

MOVING FORWARD

What makes deep learning expensive?

Training requires lots of computation

- Specialized hardware (GPUs, TPUs) can help with matrix computations
- Can use advanced iterative optimization methods
- Can parallelize training

Hyperparameter tuning and neural architecture search (NAS) make this worse

- Lots of knobs to tune!

Resulting models can be large!

- Can be expensive to store model, perform inference

MOVING FORWARD

What makes deep learning expensive?

Training requires lots of computation

- Specialized hardware (GPUs, TPUs) can help with matrix computations
- Can use advanced iterative optimization methods
- Can parallelize training

Hyperparameter tuning and neural architecture search (NAS) make this worse

- Lots of knobs to tune!

Resulting models can be large!

- Can be expensive to store model, perform inference

MOVING FORWARD

What makes deep learning expensive?

Training requires lots of computation

- Specialized hardware (GPUs, TPUs) can help with matrix computations
- Can use advanced iterative optimization methods
- Can parallelize training

Hyperparameter tuning and neural architecture search (NAS) make this worse

- Lots of knobs to tune!

Resulting models can be large!

- Can be expensive to store model, perform inference

Outline

1. Optimization review
2. DL Optimization: adaptive learning rates, momentum
3. Other tricks: early stopping, batch norm

RECALL

Empirical risk minimization

$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

- A popular approach in supervised ML
- Given a loss ℓ and data (x_1, y_1), … (x_n, y_n), we estimate a predictor *f* by minimizing the *empirical risk*
- We typically restrict this predictor to lie in some class, *F*
  - Could reflect our prior knowledge about the task
  - Or may be for computational convenience

Question: how should we select our function class, *F*, and our loss function, ℓ ??

RECALL

Previous assumption: linear models

$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

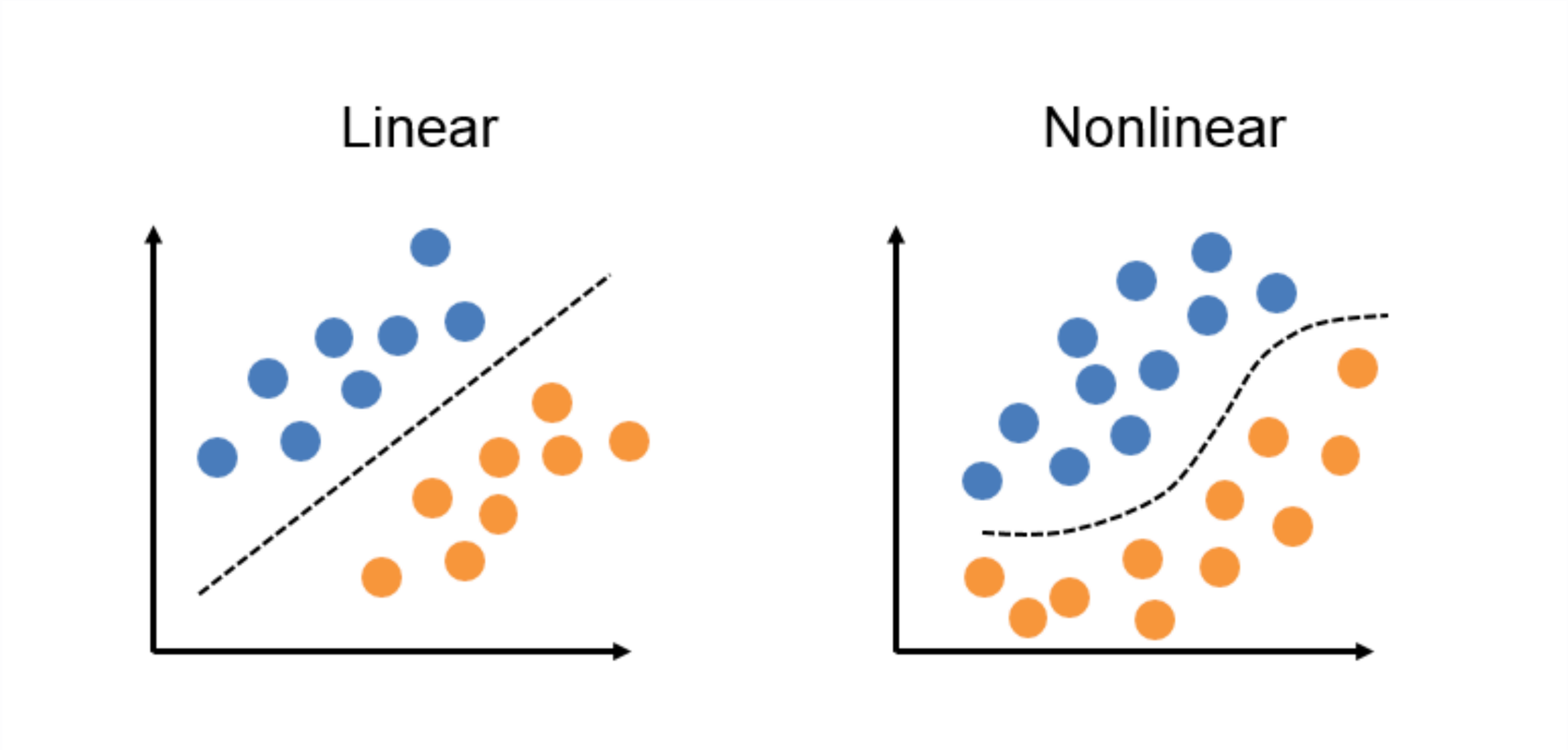
Assume: $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$

Losses:

- Square loss: linear regression $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (\mathbf{w}^\top \mathbf{x}_i - y_i)^2$
- Log loss: logistic regression $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \ell_{log}(y_i \cdot \mathbf{w}^\top \mathbf{x}_i)$
- Hinge loss: SVMs $\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \max\{0, 1 - y_i \cdot \mathbf{w}^\top \mathbf{x}_i\}$

RECALL

Not all data is linear …



How can we learn more complicated hypotheses?

ONE APPROACH

Neural networks

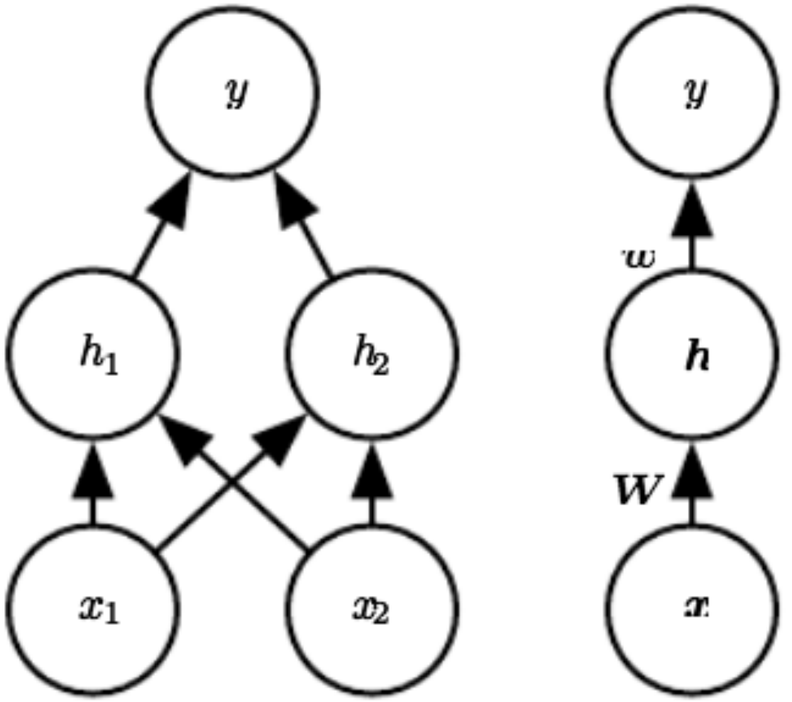
Goal: apply a nonlinear transformation to our input data: $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}; \theta)$

Neural networks provide a way to learn such a transformation

Simple approach: feedforward neural networks

- E.g., suppose we have: $\phi(\mathbf{x}; \mathbf{W}) = \sigma(\mathbf{W}^\top \mathbf{x})$
- Then: $f(\mathbf{x}) = \mathbf{w}^\top \sigma(\mathbf{W}^\top \mathbf{x})$

$\sigma(\cdot)$ is referred to as an *activation function*



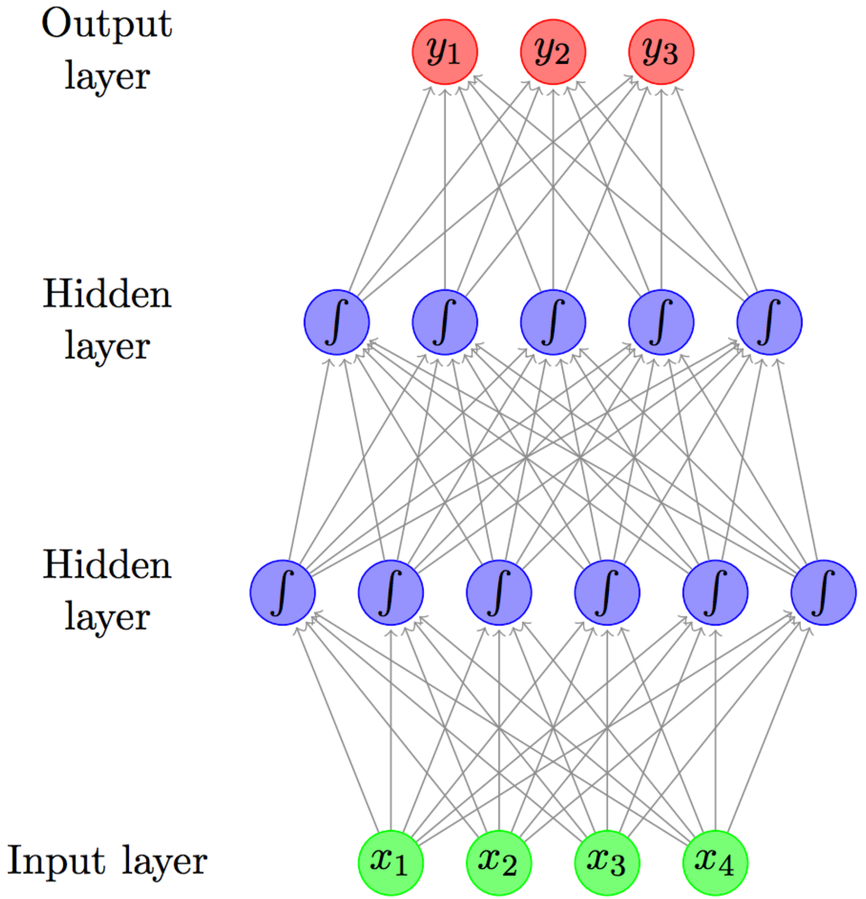
an example of a feedforward network with *one* hidden layer containing *two* units

MORE GENERALLY

Feedforward neural networks

We can consider adding L hidden layers, each with their own activation functions σ_l and corresponding weights $\mathbf{W}_l$

Using this layering technique, the resulting model becomes:



$$f(\mathbf{x}) = \mathbf{w}^\top \sigma_L(\mathbf{W}_L^\top(\sigma_{L-1}(\mathbf{W}_{L-1}^\top \ldots \sigma_2(\mathbf{W}_2^\top \sigma_1(\mathbf{W}_1^\top \mathbf{x}))\ldots)$$

QUESTION

How do we train neural networks?

$$\hat{f}_n = \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} \ell(f(x_i), y_i)$$

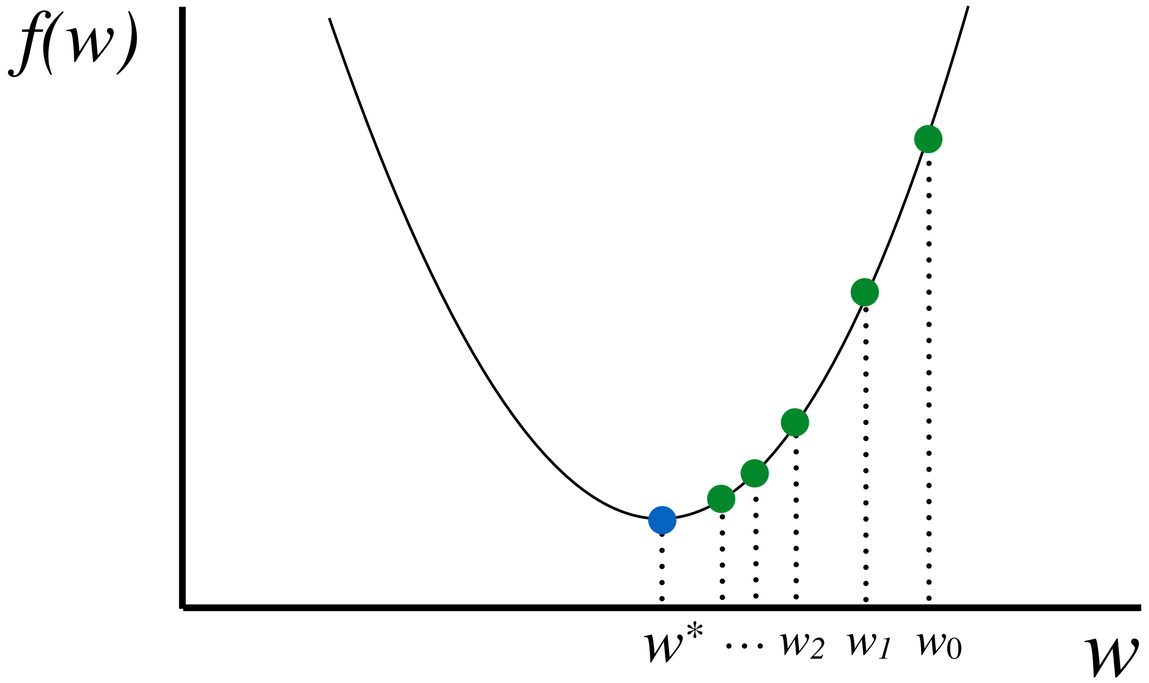
- A popular approach in supervised ML
- Given a loss ℓ and data (x_1, y_1), … (x_n, y_n), we estimate a predictor *f* by minimizing the *empirical risk*
- We typically restrict this predictor to lie in some class, *F*
  - Could reflect our prior knowledge about the task
  - Or may be for computational convenience

To train: solve the ERM objective via an iterative optimization method

RECALL

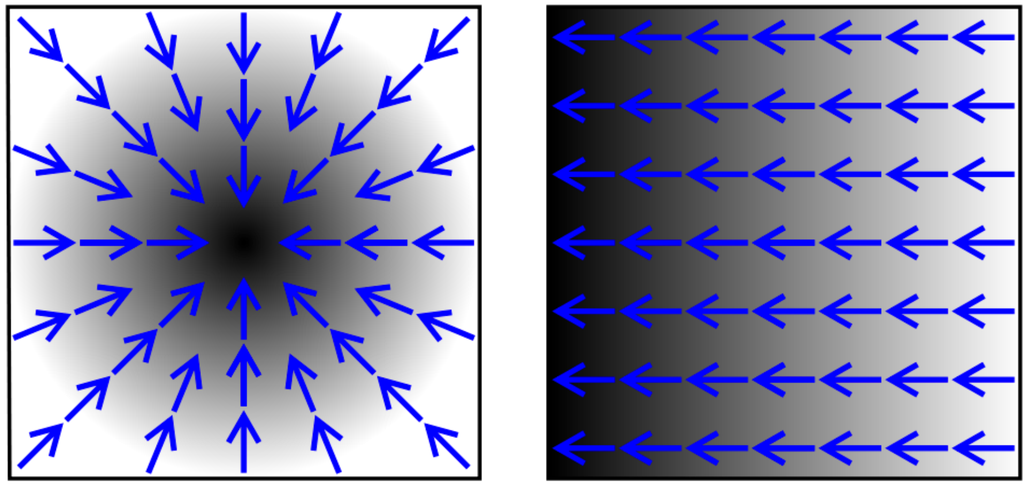
Gradient descent

Start at a random point
Repeat
    Determine a descent direction
    Choose a step size
    Update
Until stopping criterion is satisfied



RECALL

Choosing a descent direction



"Gradient2" by Sarang. Licensed under CC BY-SA 2.5 via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Gradient2.svg#/media/File:Gradient2.svg

2D Example:

- Function values are in black/white and black represents higher values
- Arrows are gradients

We can move anywhere in $\mathbb{R}^d$

Negative gradient is direction of *steepest* descent!

Step Size

Update Rule: $\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \nabla f(\mathbf{w}_i)$

Negative Slope

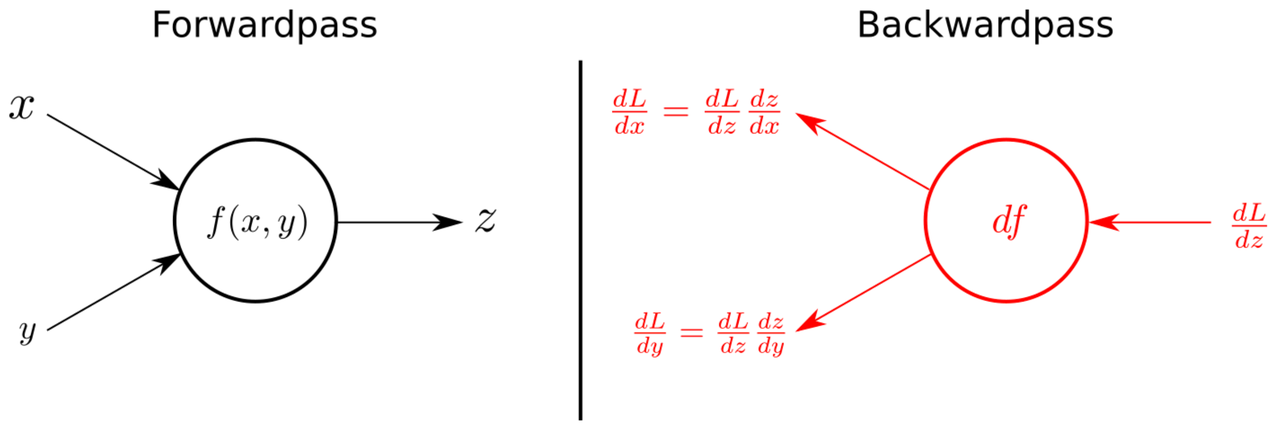
RECALL

How are gradients computed for NNs?

Backpropagation: efficiently computes the gradient of a feedforward neural network

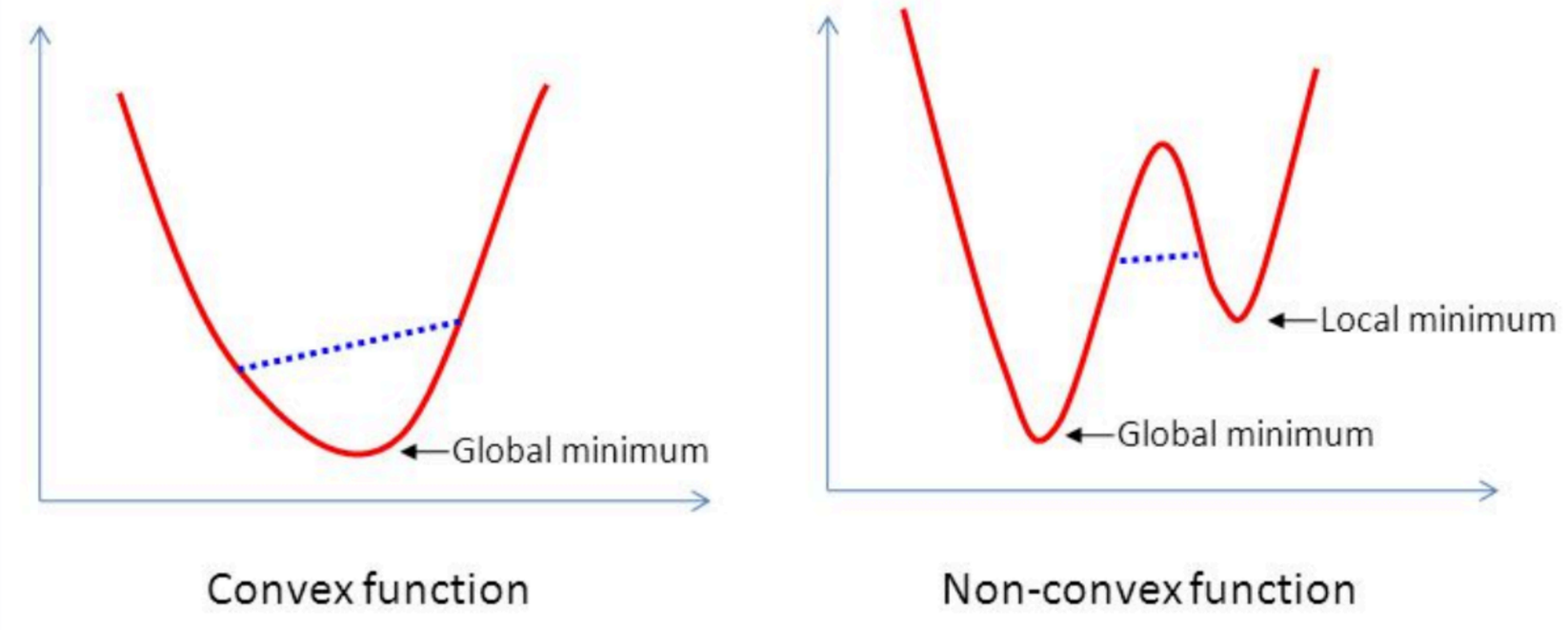
Two main steps:

- *forward pass*: traverse through the network normally, as if we were computing the function value, and save all intermediate values
- *backward pass*: proceed backwards through the network, computing gradients of the function value with respect to intermediates



CHALLENGE

Non-convexity



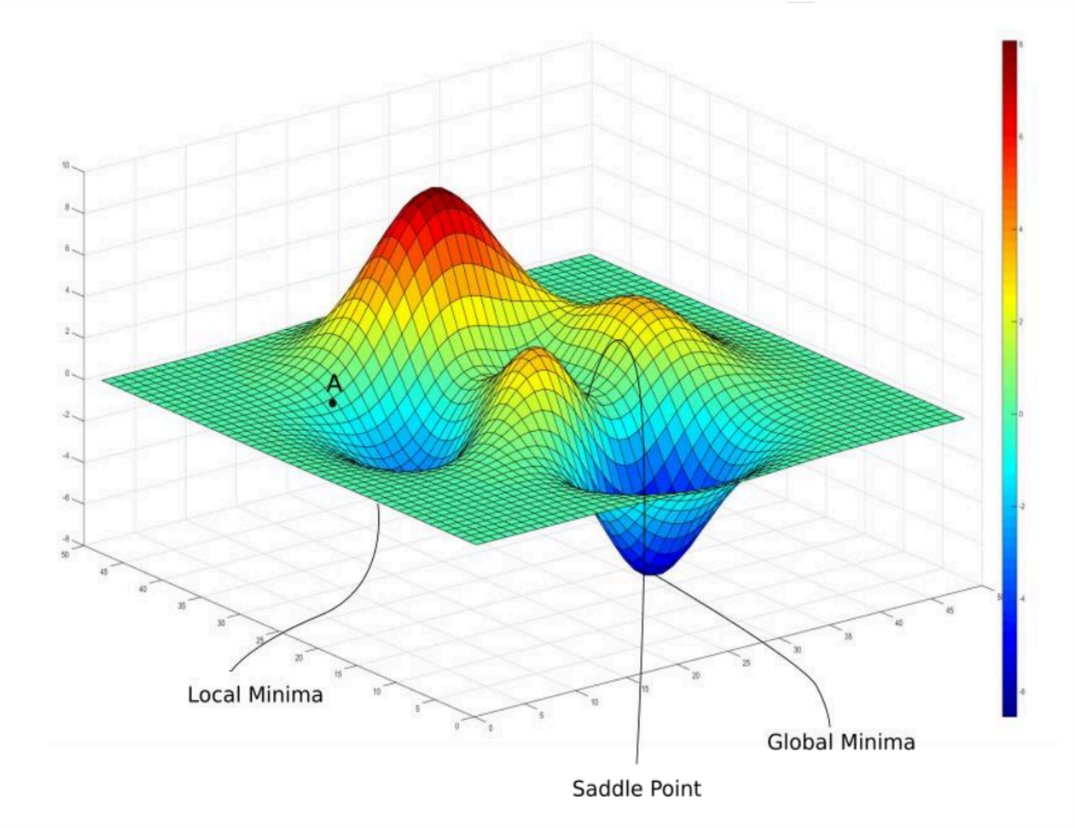
Deep neural networks are non-convex

How would a method like gradient descent work on non-convex objectives?

Can find *stationary points,* i.e., points where $\nabla f(x) \approx 0$

CHALLENGE

Non-convexity

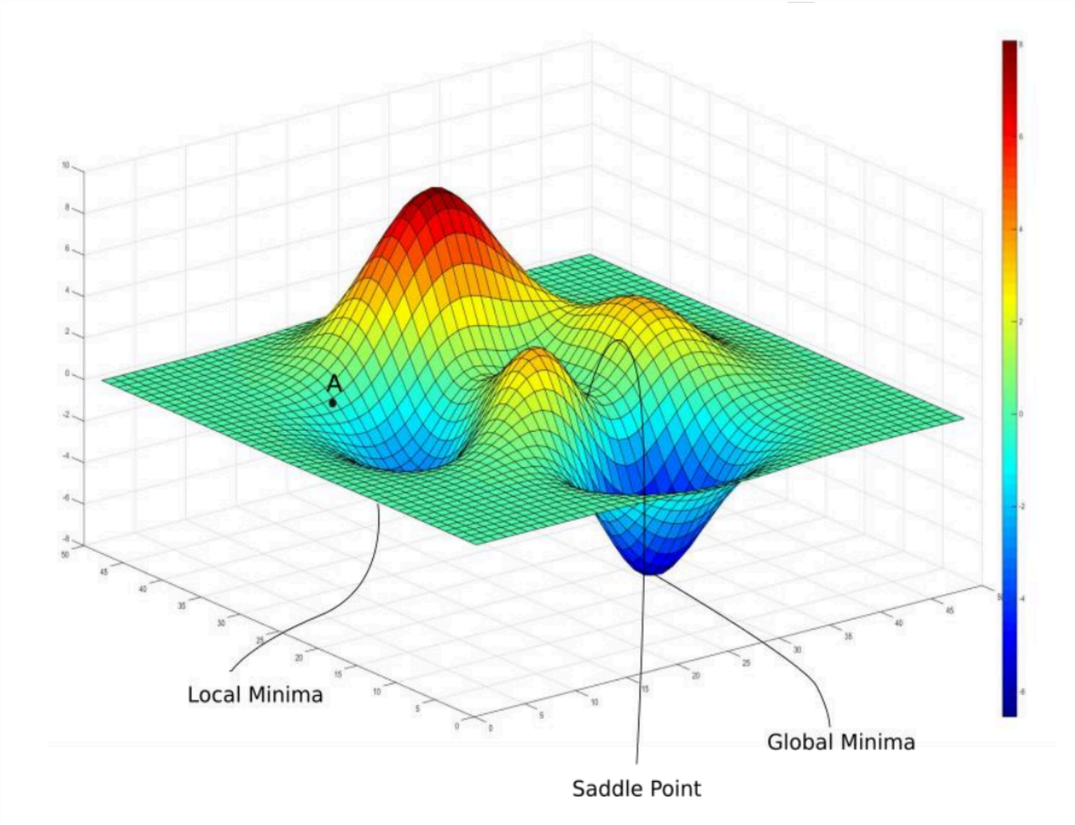


Types of stationary points:

- Global minimum: actual minimizer, i.e., $f(\hat{\mathbf{x}}) \leq f(\mathbf{x}) \;\; \forall\, \mathbf{x} \in \mathbb{R}^k$
- Local minimum: $f(\hat{\mathbf{x}}) \leq f(\mathbf{x}) \;\; \forall\, \mathbf{x} \in \mathbb{R}^k : \;\; \|\mathbf{x} - \hat{\mathbf{x}}\| \leq \epsilon$
- Local maximum: $f(\hat{\mathbf{x}}) \geq f(\mathbf{x}) \;\; \forall\, \mathbf{x} \in \mathbb{R}^k : \;\; \|\mathbf{x} - \hat{\mathbf{x}}\| \leq \epsilon$
- Saddle point: a stationary point that is not a local minimum or maximum

CHALLENGE

Non-convexity

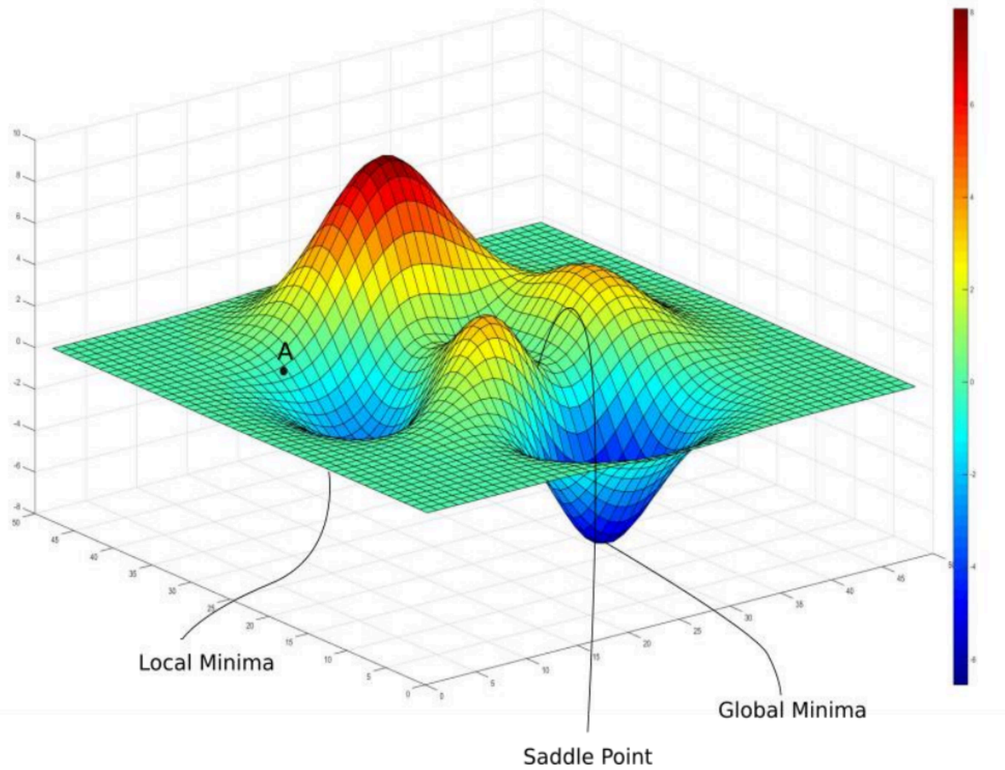


Types of stationary points:

- Global minimum: finding these is hard (both in theory [NP-hard] and in practice)
- Local minimum: can often still provide reasonable solutions
- Saddle point: want to avoid these

CHALLENGE

Non-convexity



What makes non-convex optimization difficult?

- Potentially many local minima
- Saddle points
- Very flat regions
- Widely varying curvature

RECALL

Techniques for large-scale optimization

Reduce computation:

- Parallelize/distribute computation
- Use stochastic methods
- Use first-order methods (i.e., no 2nd-order or higher gradients)

***Reduce communication*:**

- Keep large objects local
- Reduce iterations

RECALL

Techniques for large-scale optimization

Reduce computation:

- Parallelize/distribute computation
- Use stochastic methods
- Use first-order methods (i.e., no 2nd-order or higher gradients)

will talk about parallel/ distributed DL next lecture

***Reduce communication*:**

- Keep large objects local
- Reduce iterations

this is particularly important for DL, since the models are very large

Outline

1. Optimization review
2. DL Optimization: adaptive learning rates, momentum
3. Other tricks: early stopping, batch norm

Techniques for large-scale optimization

Reduce computation:

- Parallelize/distribute computation
- Use stochastic methods
- Use first-order methods (i.e., no 2nd-order or higher gradients)

will talk about parallel/ distributed DL next week

***Reduce communication*:**

- Keep large objects local
- Reduce iterations

this is particularly important for DL, since the models are very large

Reducing iterations

We are interested in optimization techniques that reduce the total number of iterations until convergence

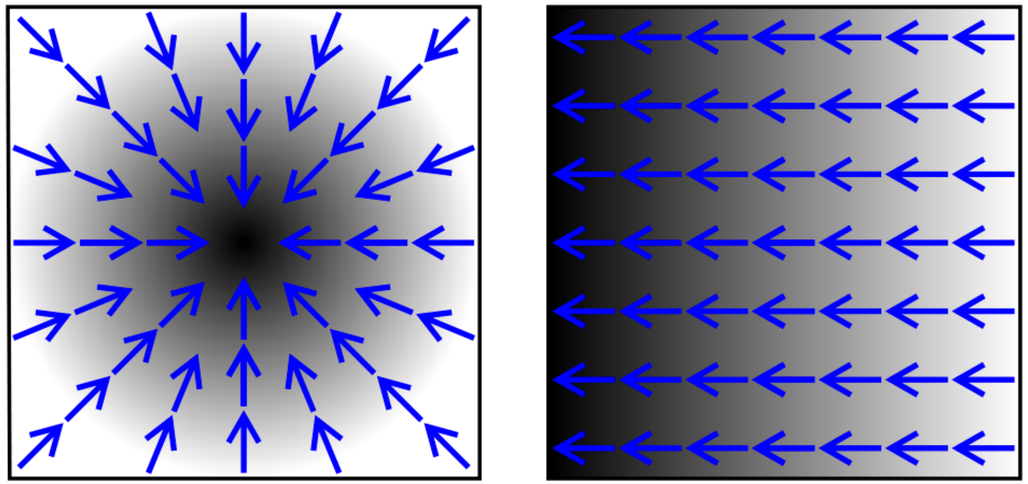
However, these techniques often come at a price: *more computation*

Will discuss two popular techniques for DL that aim to reduce iterations with little additional computation:

- Adaptive learning rate methods
- Momentum-based methods

RECALL

Gradient descent



"Gradient2" by Sarang. Licensed under CC BY-SA 2.5 via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Gradient2.svg#/media/File:Gradient2.svg

2D Example:

- Function values are in black/white and black represents higher values
- Arrows are gradients

We can move anywhere in $\mathbb{R}^d$

Negative gradient is direction of *steepest* descent!

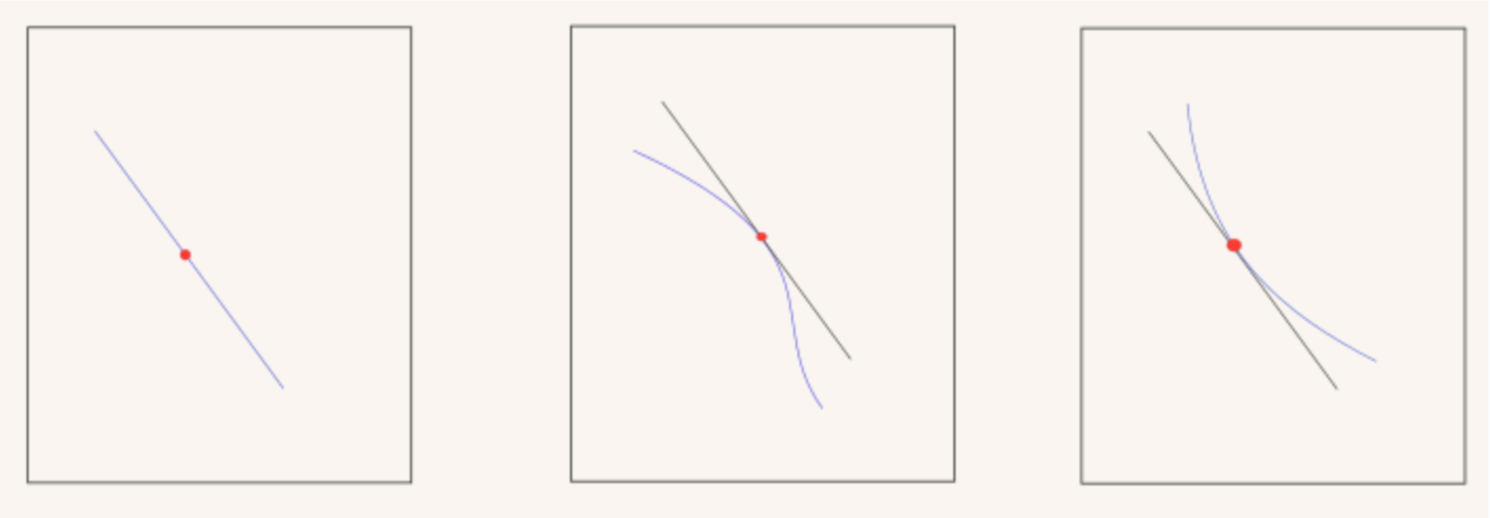
Step Size

Update Rule: $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla f(\mathbf{w}_t)$

Negative Slope

Motivation: Newton's method

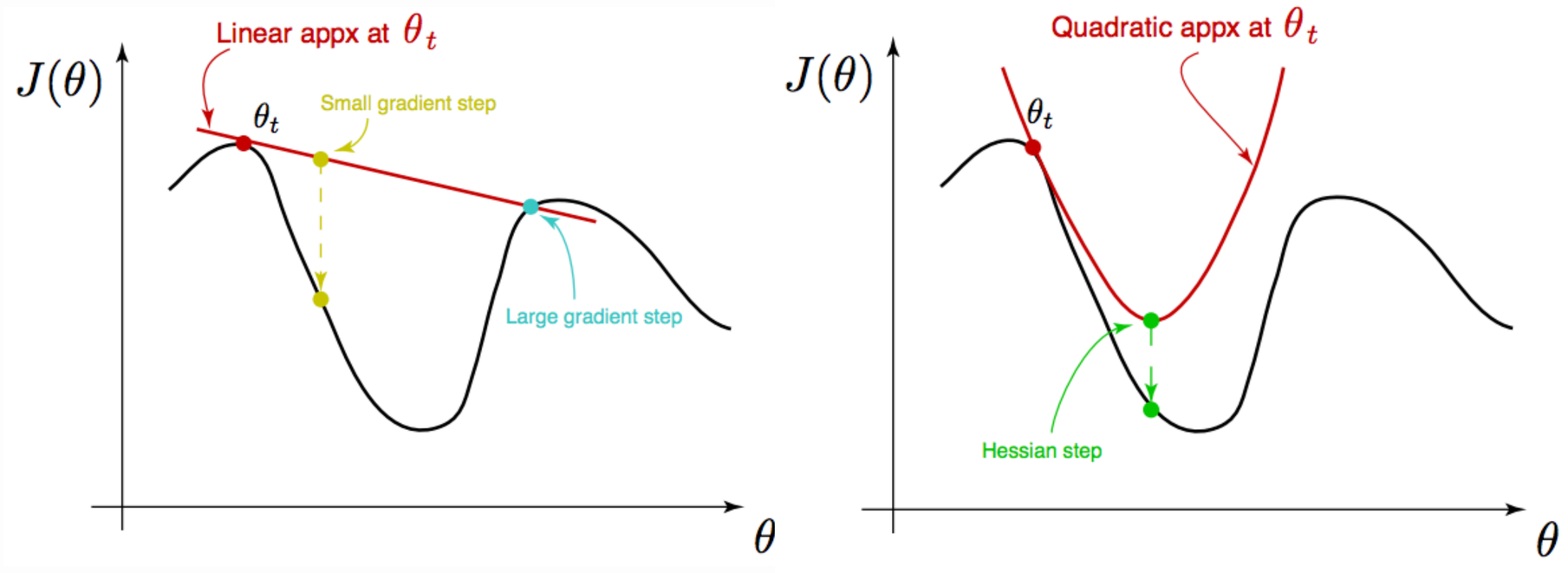
issue: gradient may be the same despite varying curvature



Newton's method uses second order information to estimate the curvature of the function being minimized (i.e., the ERM objective)

Motivation: Newton's method

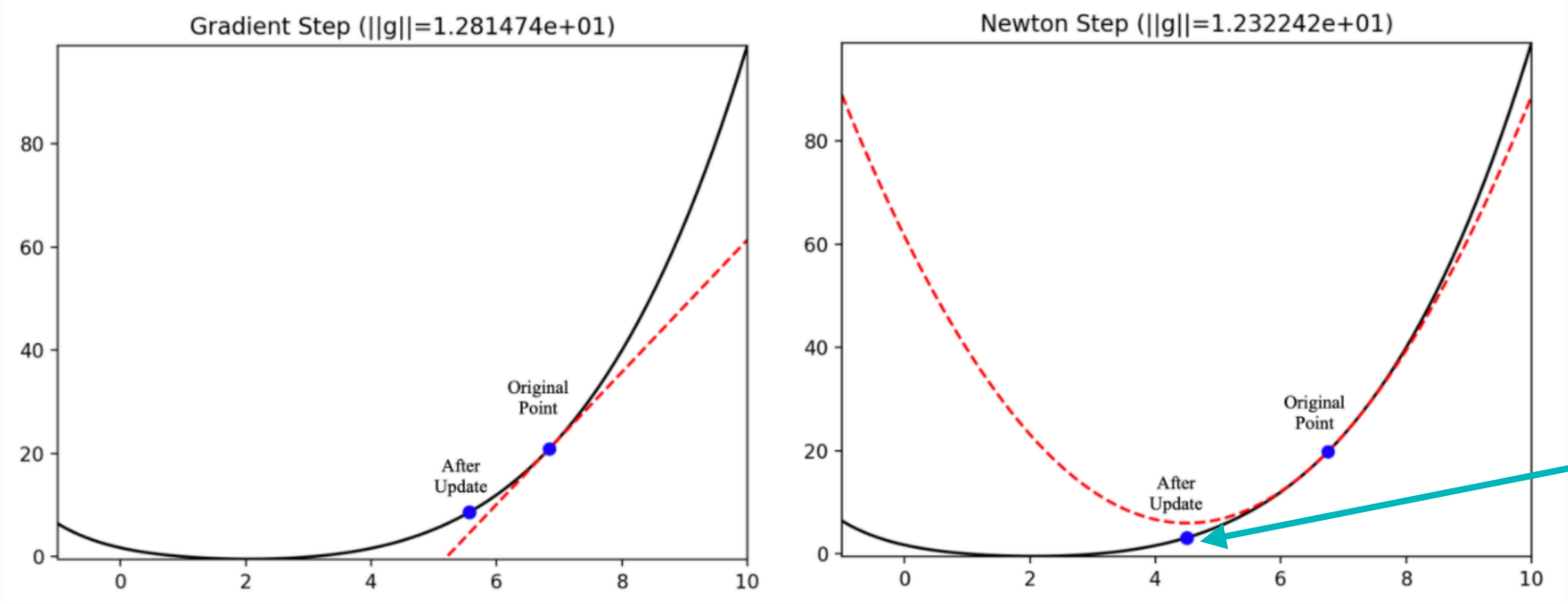
issue: gradient may be the same despite varying curvature



Newton's method uses second order information to estimate the curvature of the function being minimized (i.e., the ERM objective)

Motivation: Newton's method

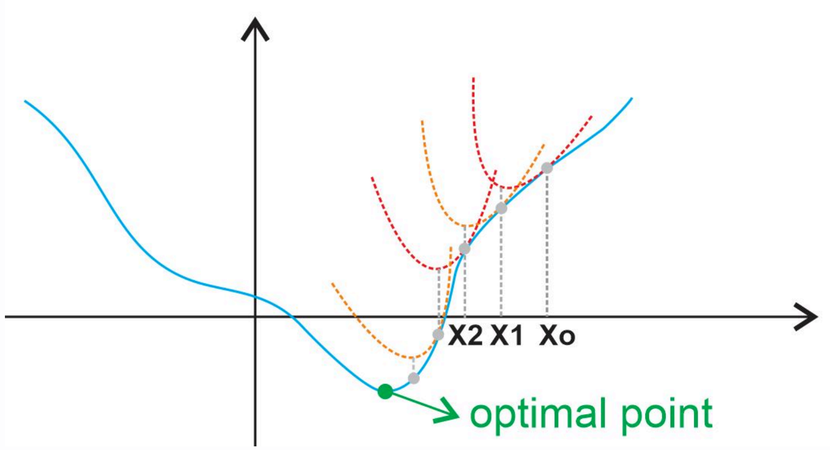
issue: gradient may be the same despite varying curvature



Newton step makes more progress

Newton's method uses second order information to estimate the curvature of the function being minimized (i.e., the ERM objective)

Motivation: Newton's method

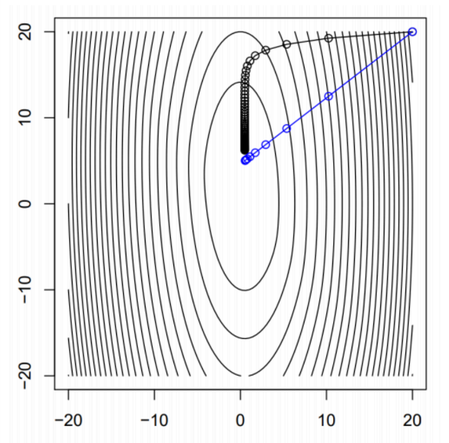


RECALL

Gradient descent $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla f(\mathbf{w}_t)$

can we do better? idea: use second order information (i.e., the hessian)

Newton's method $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \left(\nabla^2 f(\mathbf{w}_t) \right)^{-1} \nabla f(\mathbf{w}_t)$



can significantly reduce number of iterations needed for convergence

however, comes at a cost: need to invert $k \times k$ matrix → $O(k^3)$

often too expensive for neural networks (k can be in the millions)

A CHEAPER APPROACH

AdaGrad

AdaGrad: $$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha G_t^{-1} \nabla f(\mathbf{w}_t)$$

where $$\left(G_t\right)_{ii} = \sqrt{\sum_{j=1}^{t} \nabla f(\mathbf{w}_j)_i^2} \quad , \quad \textit{else} \quad 0$$

- A cheap approximation of the Hessian: only estimate the diagonal elements
- Enforces a separate step size / learning rate for every coordinate of the weight vector
  - Decreases the step size for coordinates with high gradient magnitudes
- Why does this make sense when the model is large?
  - There may be high variability across the coordinates
- Potential issue: step size can become very small in some directions

Duchi et al, 2011

AN ALTERNATIVE TO ADAGRAD

RMSProp

RMSProp: $$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha G_t^{-1} \nabla f(\mathbf{w}_t)$$

where $$\left(G_t\right)_{ii} = \sqrt{(g_t)_i} \;, \quad \textit{else} \quad 0$$

$$\left(g_t\right)_i = \beta(g_{t-1})_i + (1-\beta)(\nabla f(\mathbf{w}_t)_i)^2$$

- Take a moving average of the coordinates of the gradient, instead of a sum
- Puts more of an emphasis on the current gradients than the gradient history
- Can be more effective for DNNs
- Also see: AdaDelta

Hinton [Coursera], 2012

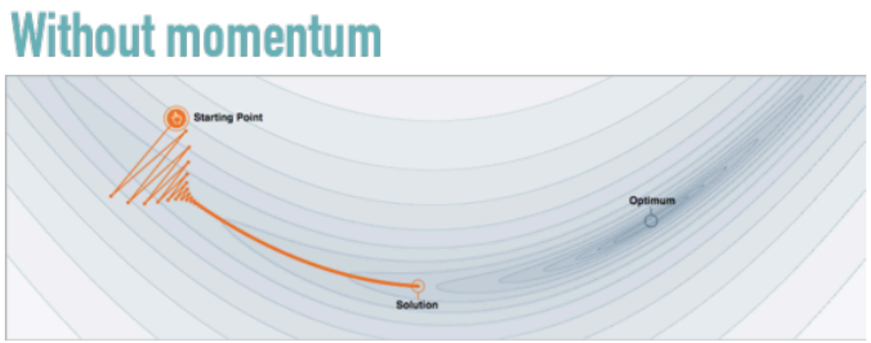
ANOTHER APPROACH

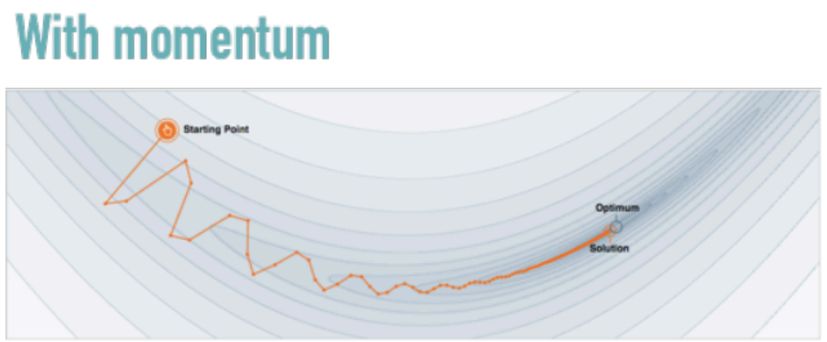
Momentum

Polyak momentum

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla f(\mathbf{w}_t) + \beta(\mathbf{w}_t - \mathbf{w}_{t-1})$$

- Another cheap way to potentially improve gradient descent: use momentum
- Intuition: if current gradient step is in same direction as previous step, then move a little further in that direction; if not, move less far
- Also known as the heavy ball method





Polyak ‘64

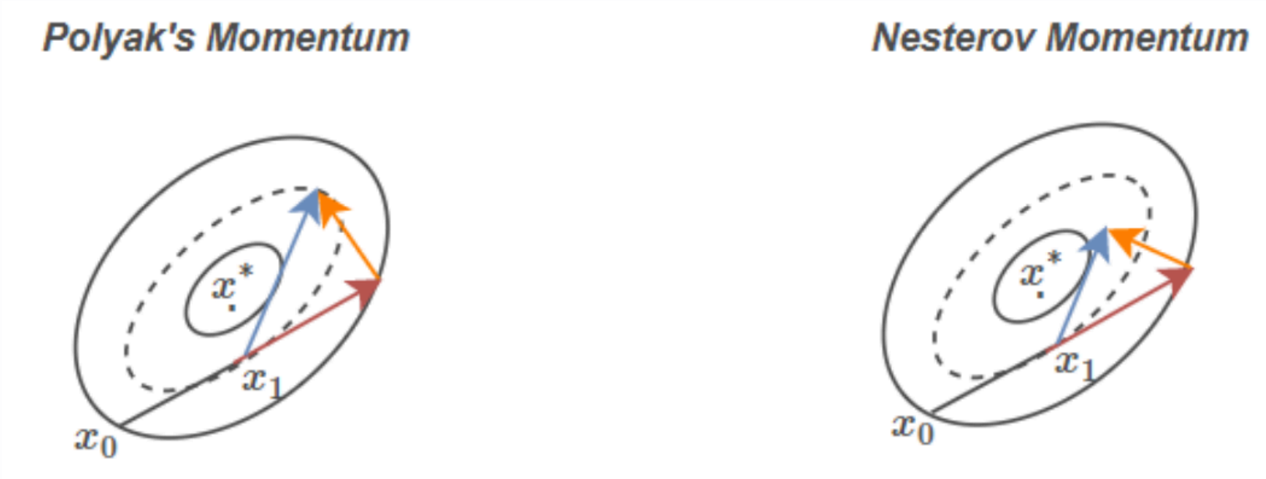
Nesterov accelerated gradient

evaluate gradient at 'look-ahead' point

Nesterov momentum

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla f(\mathbf{w}_t + \beta(\mathbf{w}_t - \mathbf{w}_{t-1})) + \beta(\mathbf{w}_t - \mathbf{w}_{t-1})$$

- A "look-ahead" variant of momentum
- Provably faster convergence than GD for any convex function



Nesterov '83

ONE LAST METHOD: COMBINE EVERYTHING!

Adam

Adam:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha G_t^{-1} \mathbf{v}_t$$

where

$$\left(G_t\right)_{ii} = \sqrt{(g_t)_i} \quad , \quad \textit{else} \quad 0$$

$$\left(g_t\right)_i = \beta_1 (g_{t-1})_i + (1-\beta_1)(\nabla f(\mathbf{w}_t)_i)^2$$

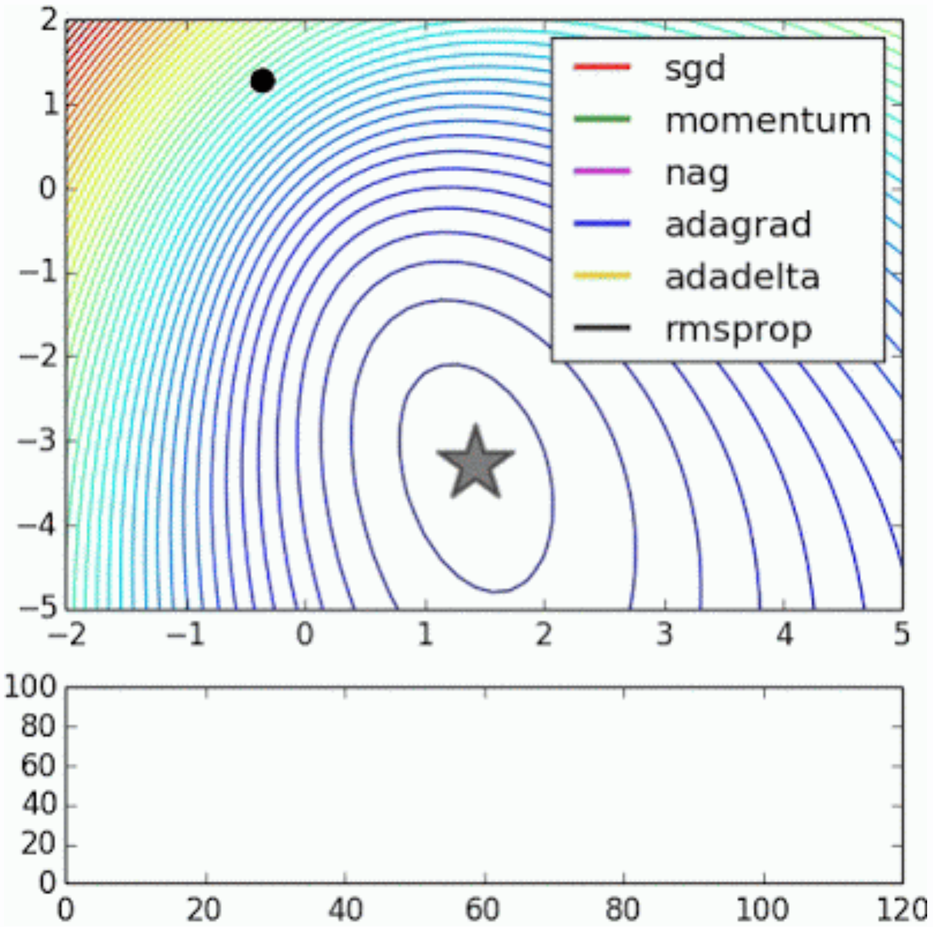
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\nabla f(\mathbf{w}_t)$$

RMSProp update

Momentum

- Combines momentum and RMSprop
- Default choice in most DL frameworks
- However, does not converge (even for convex problems) without modifications
  - [*Reddi, Kale, Kumar, ICLR '19 best paper*]

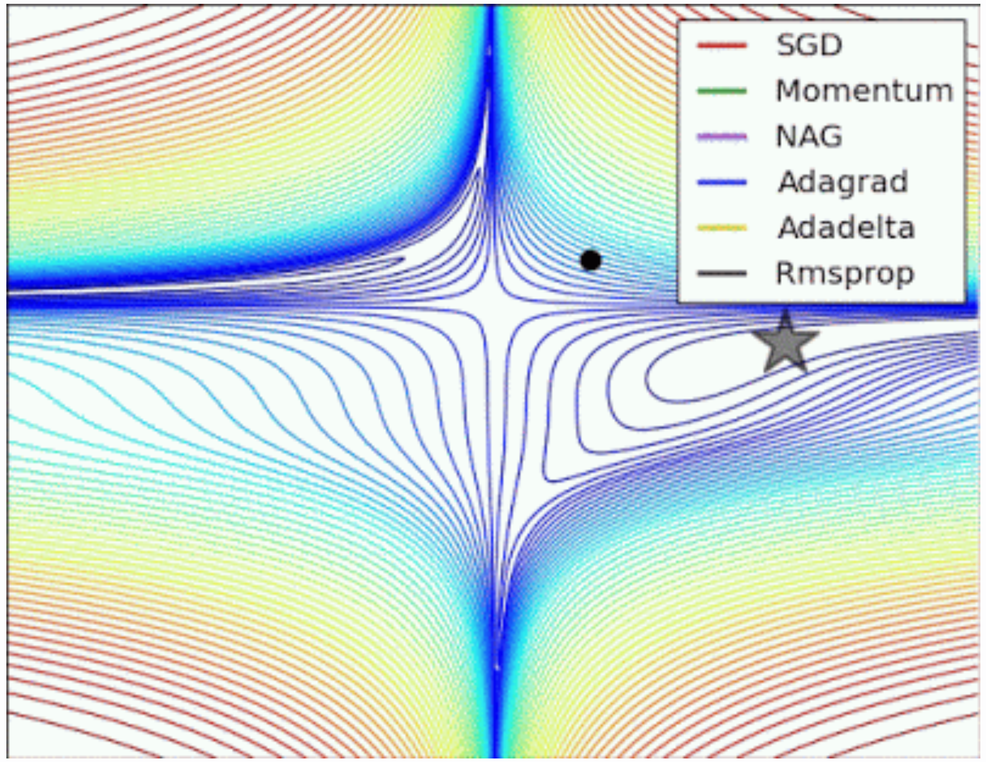
Performance on toy examples



- Logistic regression model (convex)
- All methods improve upon SGD
- AdaGrad converges the fastest
- Momentum methods have smoothing effect

https://imgur.com/a/Hqolp

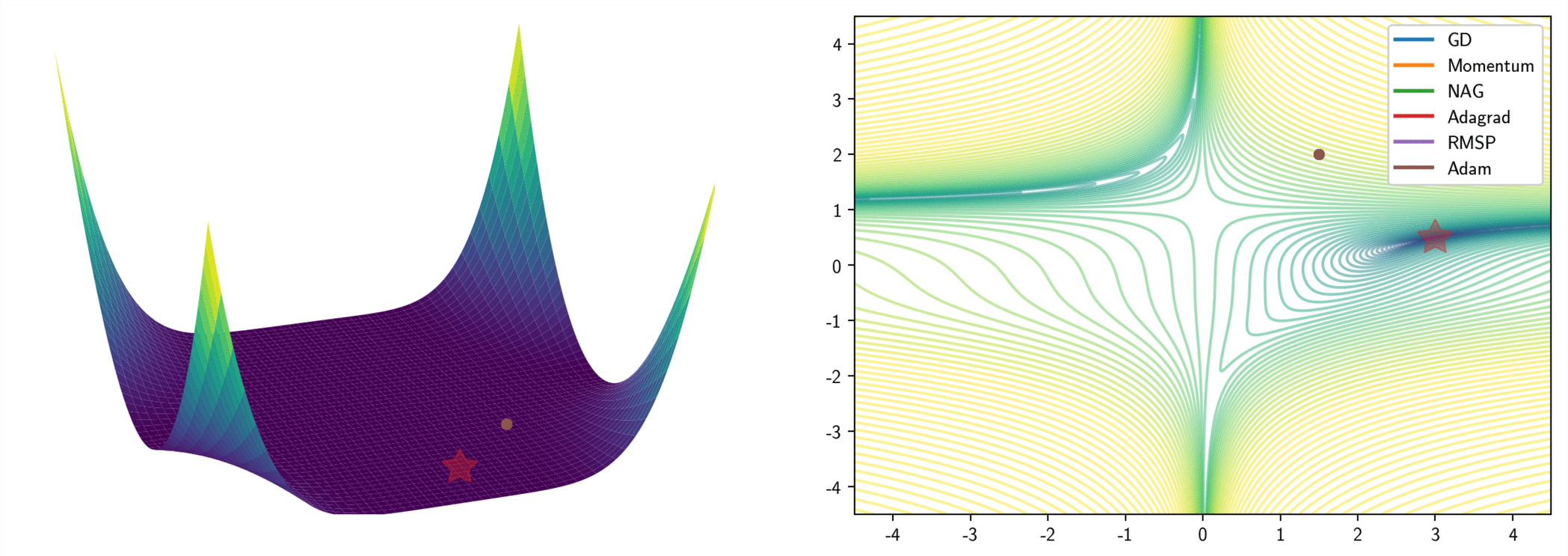
Performance on toy examples



- Beale's function (nonconvex): high variability in terms of gradients (initial gradient is very large)
- AdaGrad & momentum methods are unstable
- RMSProp/AdaDelta perform the best

https://imgur.com/a/Hqolp

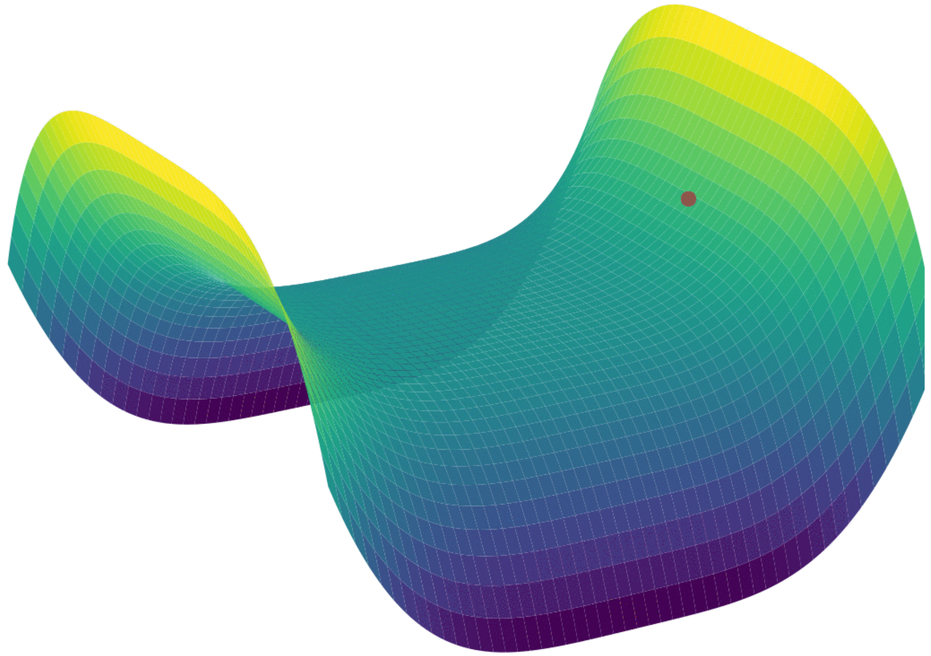
Performance on toy examples

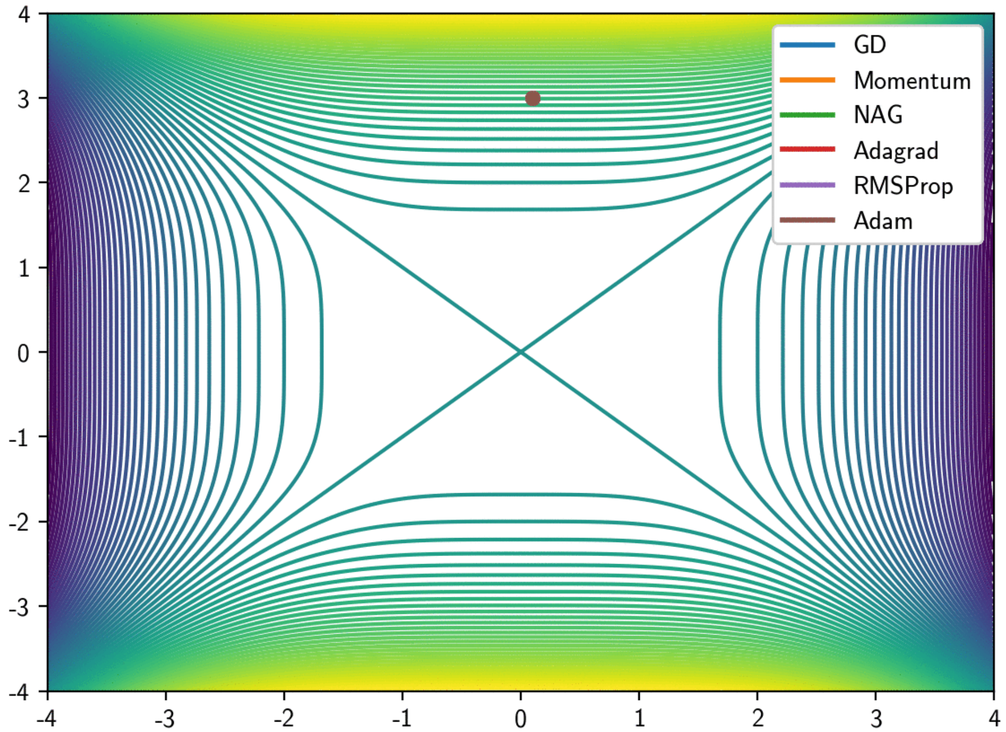


- Beale's function (nonconvex): this time including Adam
- Minimized at (3,0)

https://tangbinh.github.io/01/04/Optimizers.html

Performance on toy examples

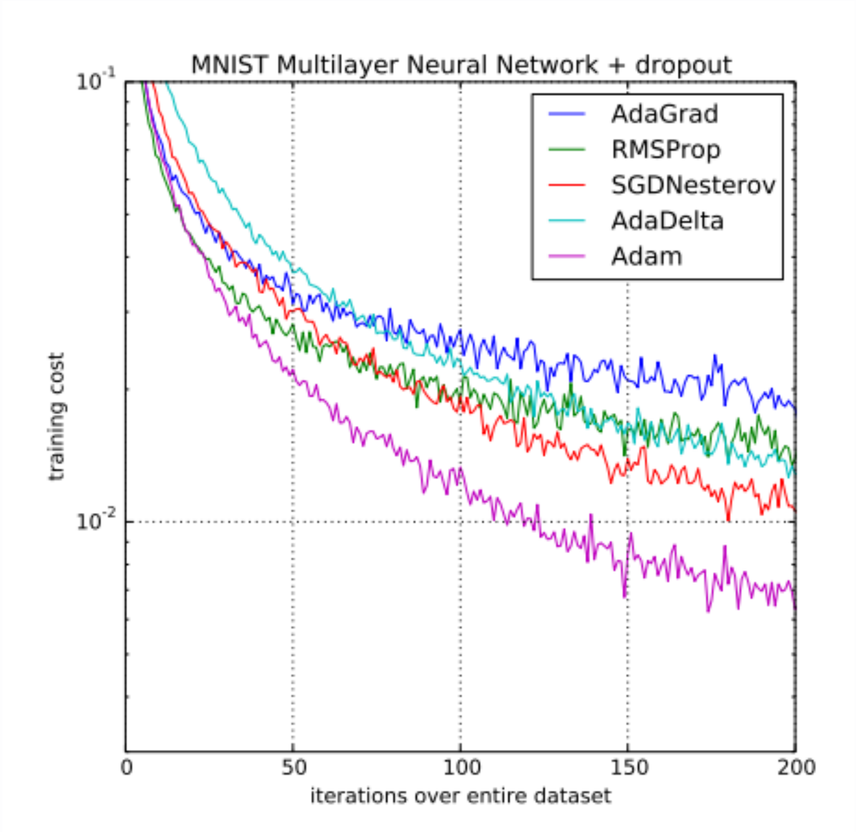


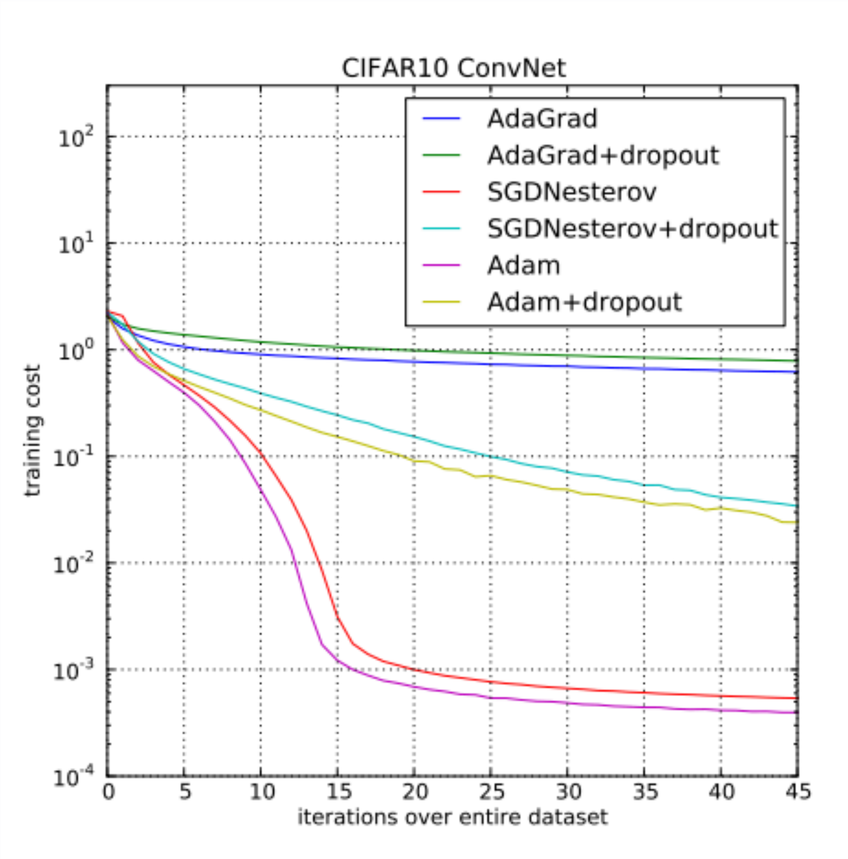


- Loss function: $f(x) = y^4 - x^4$
- Saddle point at (0,0), minimized at $(0, \infty)$

https://tangbinh.github.io/01/04/Optimizers.html

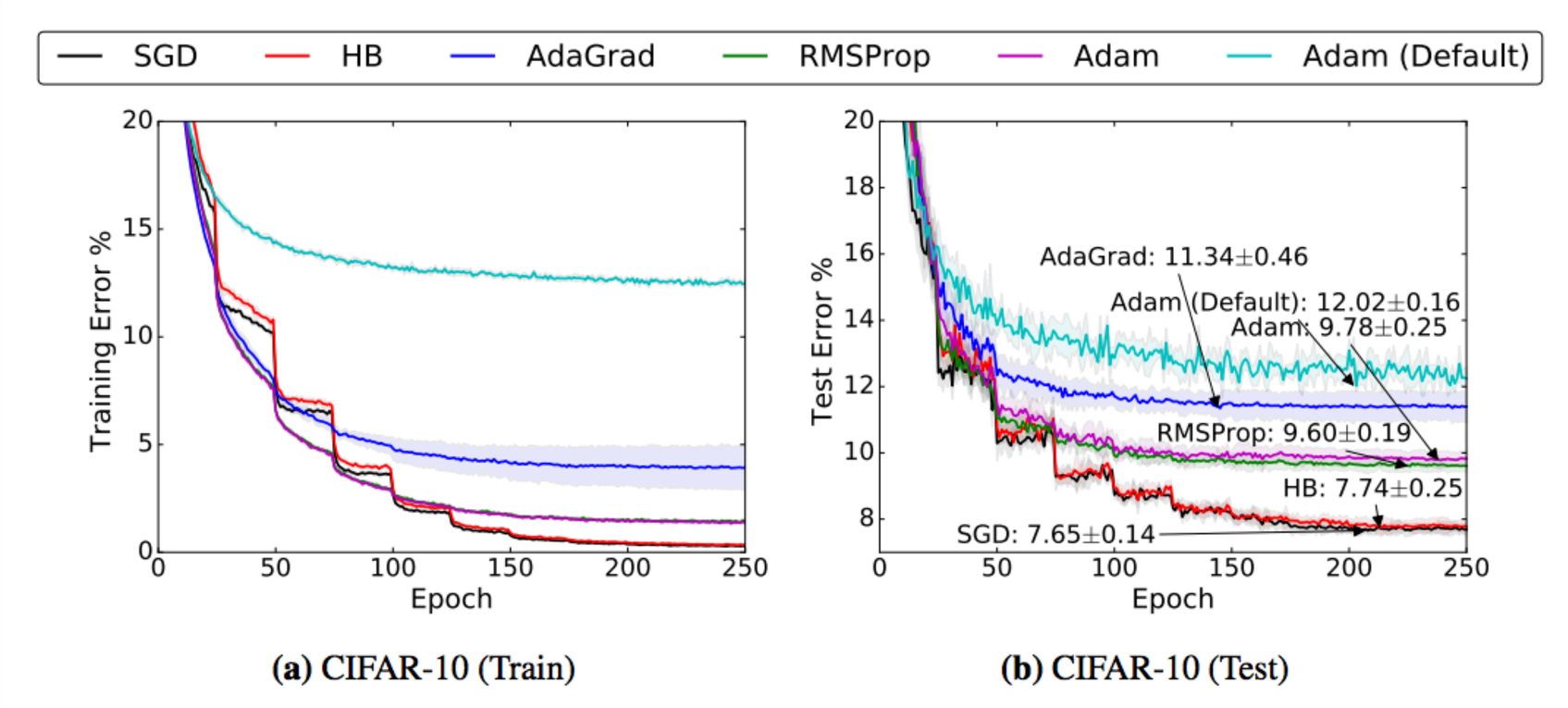
Performance on DL benchmarks





- Initial results: “Adam: A Method for Stochastic Optimization”, Kingma & Ba, ICLR 2015

Performance on DL benchmarks



(a) CIFAR-10 (Train)

(b) CIFAR-10 (Test)

- More recently: “The Marginal Value of Adaptive Gradient Methods in Machine Learning”, Wilson et al., NeurIPS 2017
- Performance can particularly drop for test error (i.e., what we actually care about!)

Outline

1. Optimization review
2. DL Optimization: adaptive learning rates, momentum
3. Other tricks: early stopping, batch norm

Batch normalization

- Motivation: Normalizing/whitening (mean = 0, variance = 1) the inputs speeds up training [LeCun et al, 1998]
  - *Could normalization at the hidden layers also be useful?*
- Batch normalization is an attempt to do this [Ioffe and Szegedy, 2014]
  - each unit's pre-activation is normalized (subtract mean, divide by standard deviation)
  - during training, mean and standard deviation computed for each minibatch
  - backpropagation takes into account the normalization
  - at test time, the global mean and standard deviation are used

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

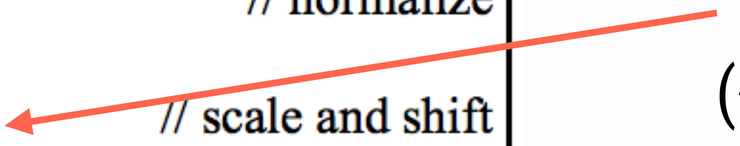
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

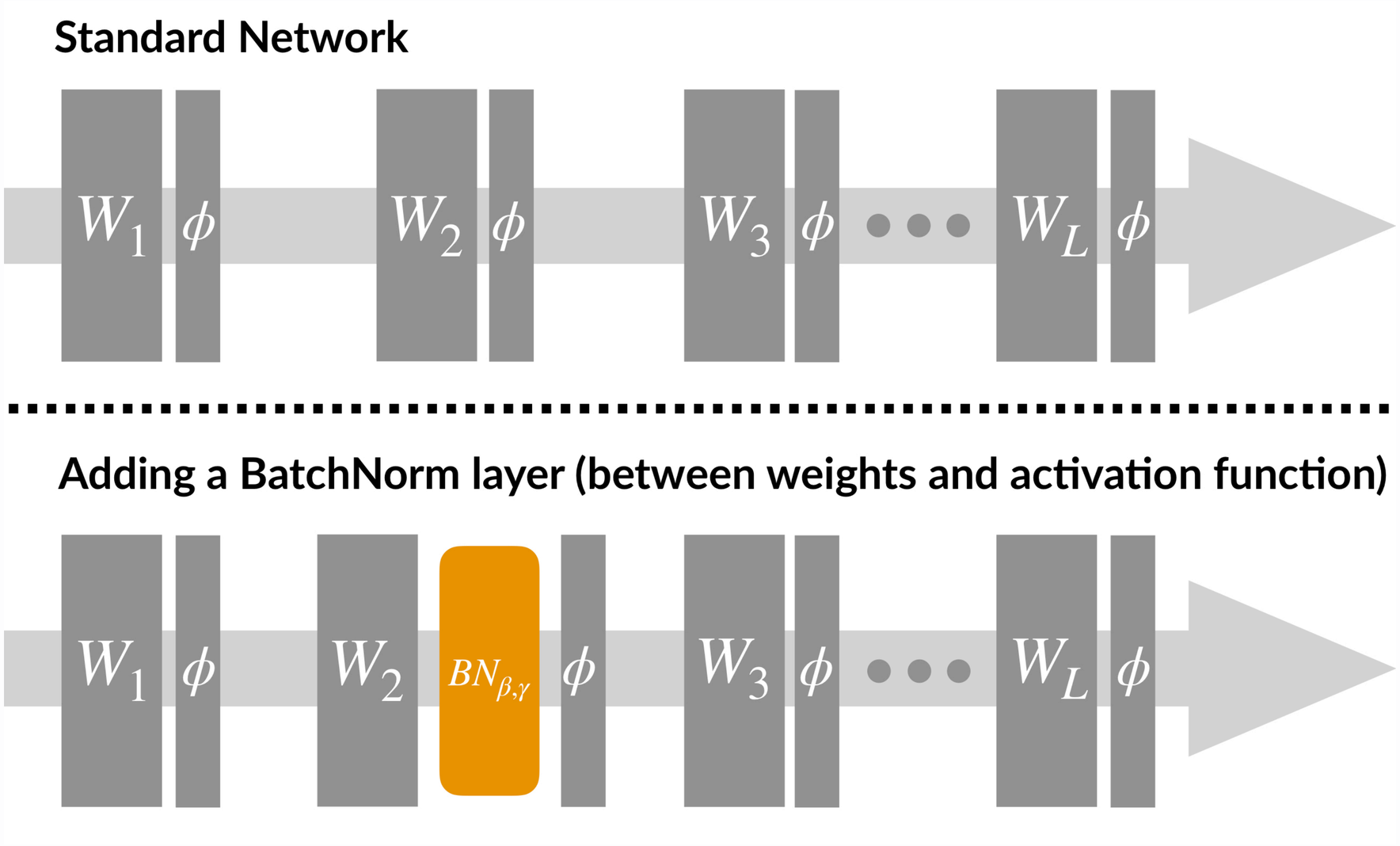
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



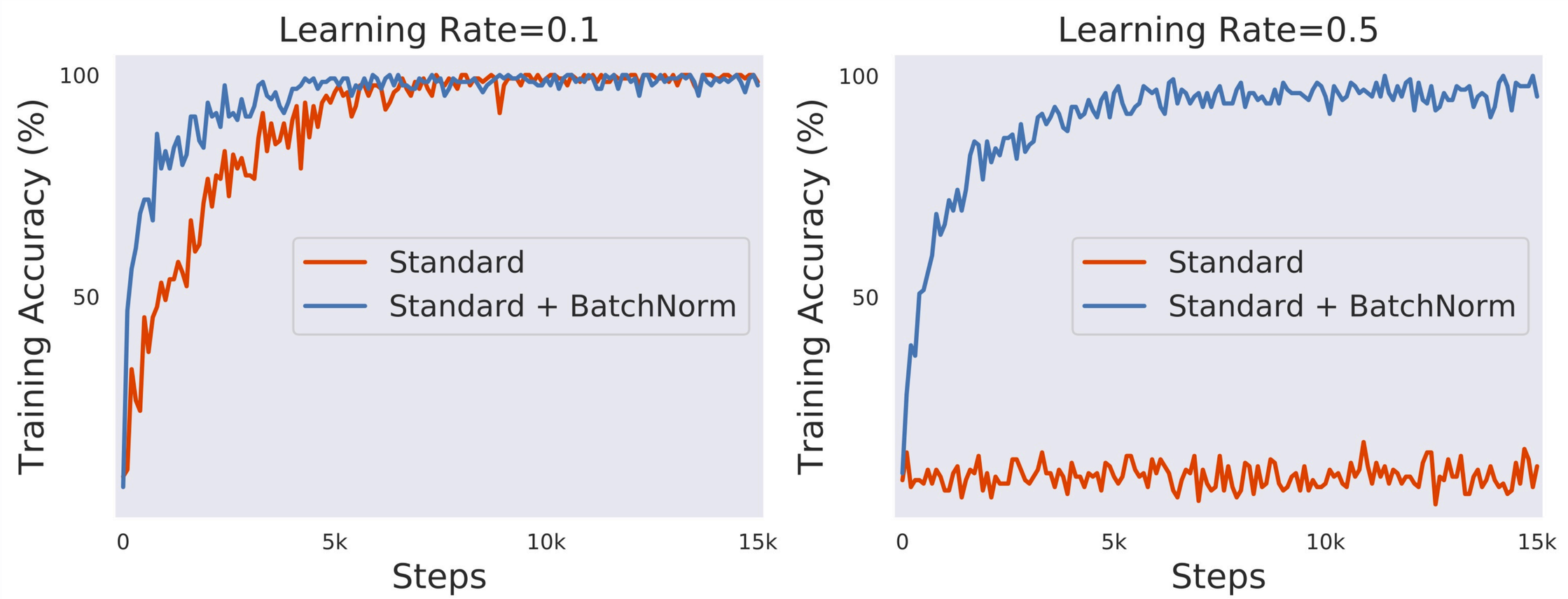
Learned linear transformation (γ and β are trained)

Batch normalization



http://gradientscience.org/batchnorm/

Batch normalization



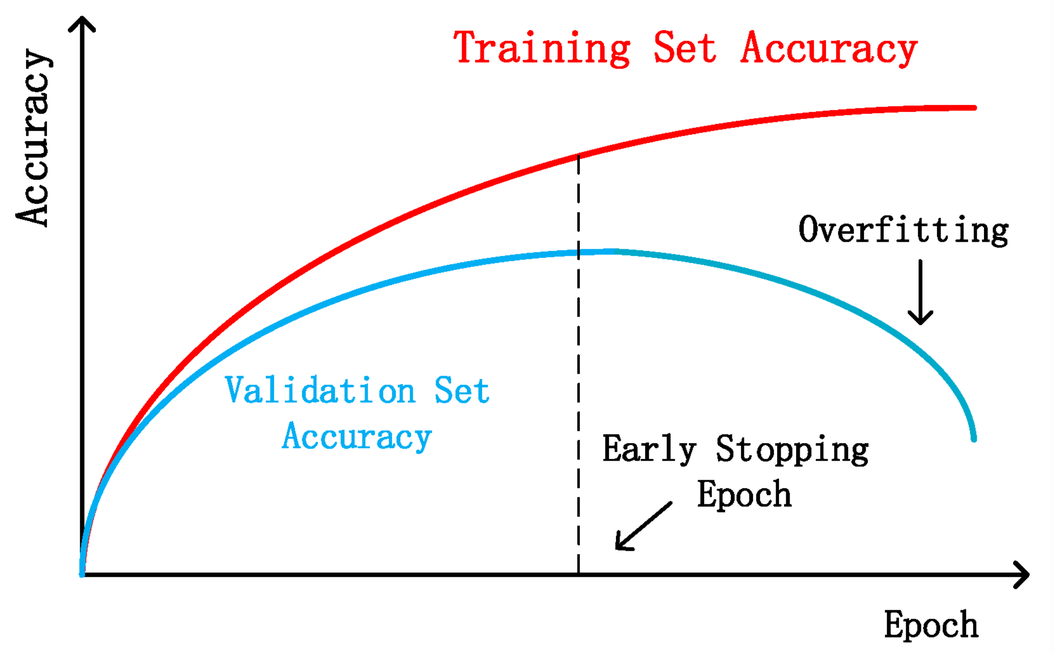
http://gradientscience.org/batchnorm/

ONE LAST TRICK …

Early stopping

- Recall: what we actually care about is generalization performance (i.e., how will the trained model perform on new, unseen data?)
- Many techniques to regularize DNNs and prevent overfitting
- Early stopping is a simple (and cheap!) way to improve generalization and prevent overfitting

- Idea: stop training when validation accuracy starts to reduce



Additional Resources

Optimization for DL

- Deep Learning Book, Chapter 8: https://www.deeplearningbook.org/
- AdaGrad:
    - https://arxiv.org/abs/1002.4908
    - http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf
- Adam: https://arxiv.org/pdf/1412.6980.pdf

Batch normalization

- http://gradientscience.org/batchnorm/