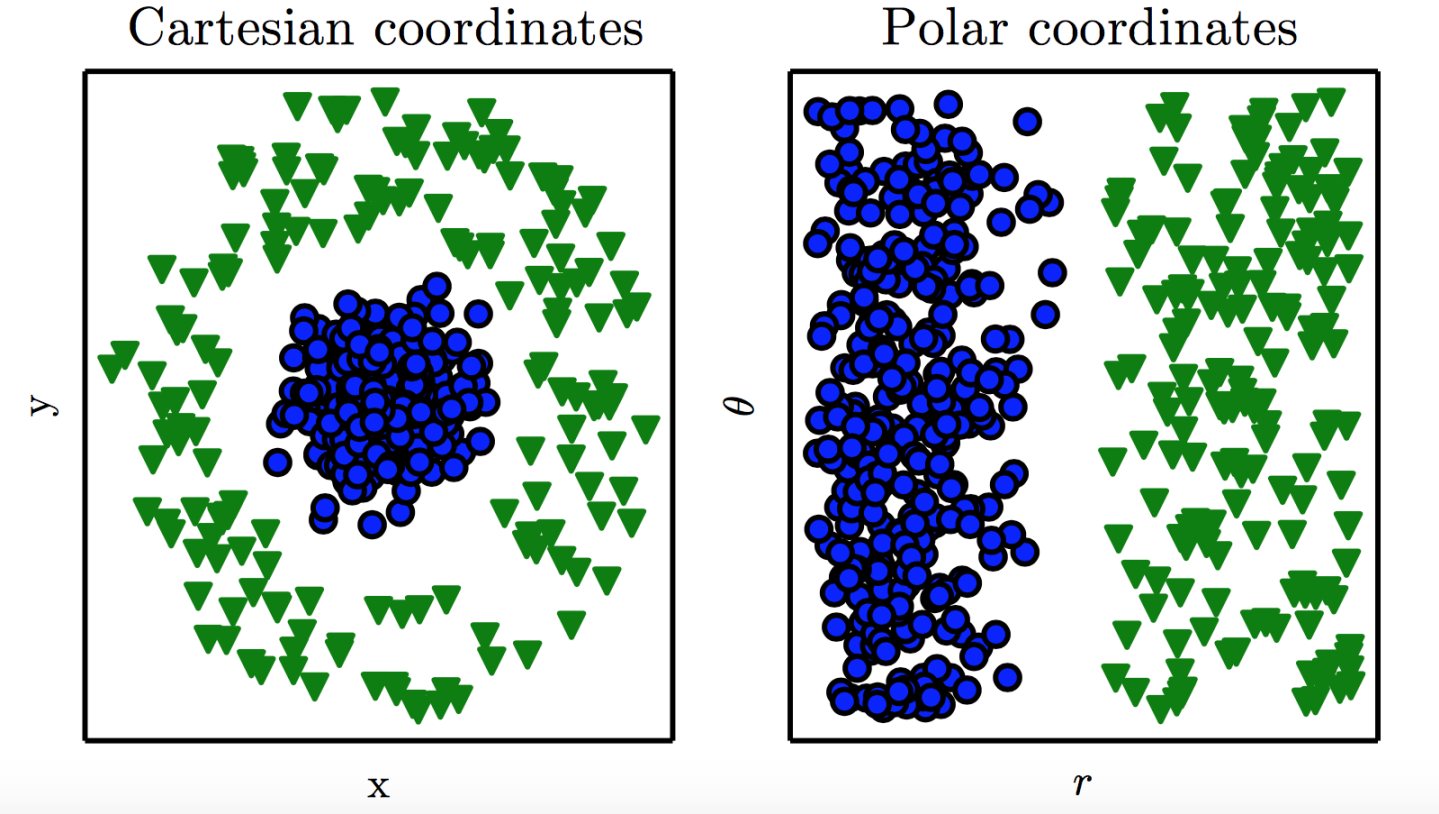
Lecture 11: Neural Networks; Design and Regularization

S-109A Introduction to Data Science

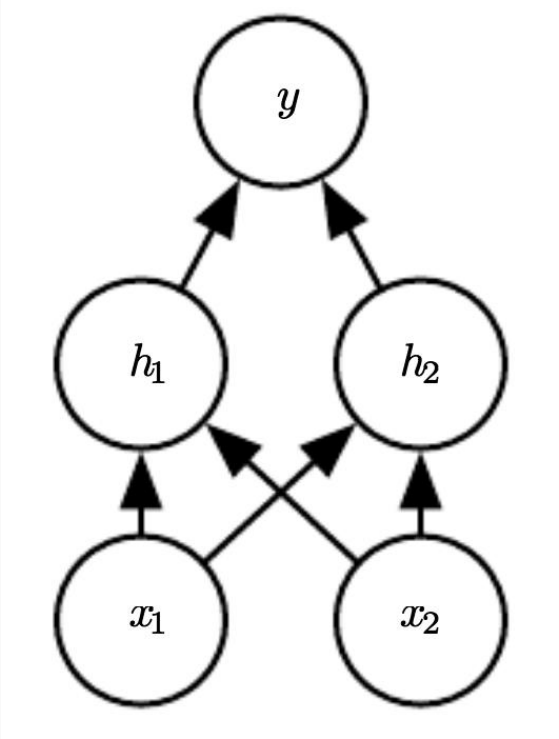
Pavlos Protopapas and Kevin Rader

Representation

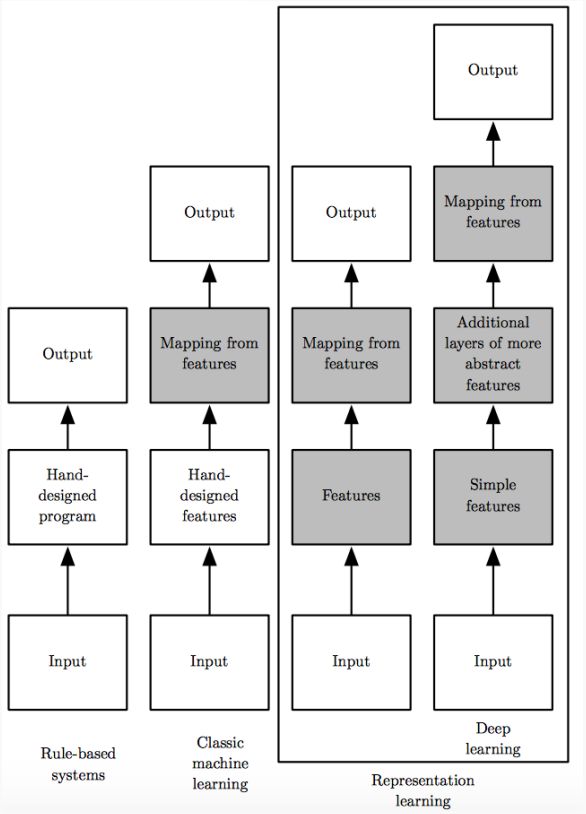
Representation Matters



Neural Network

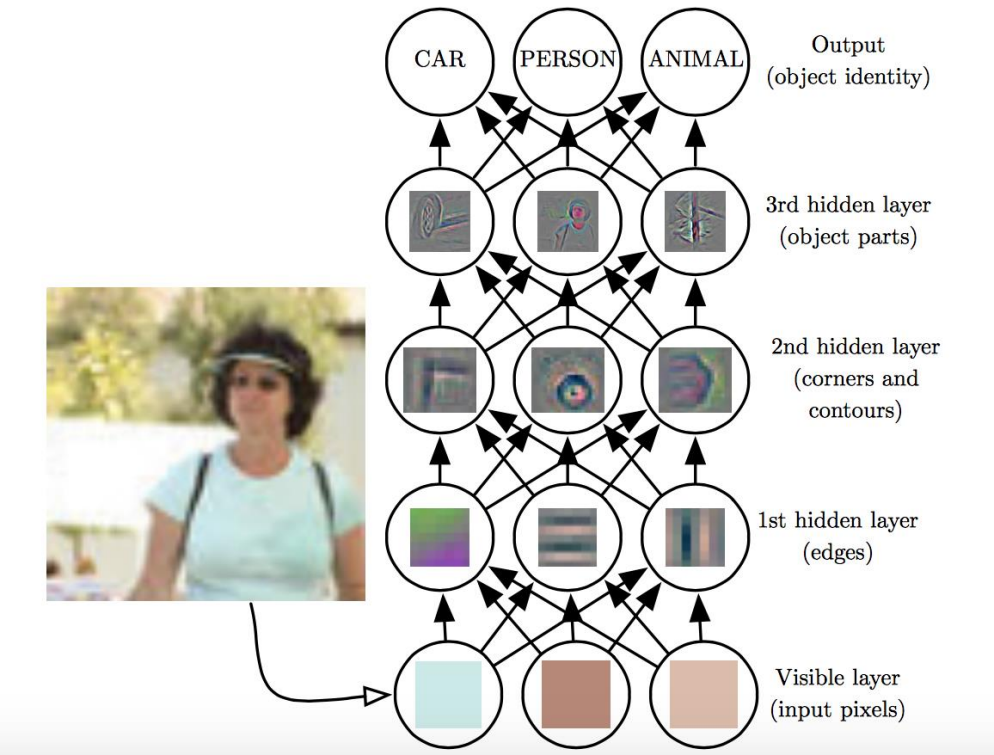


Learning Multiple Components



(Goodfellow 2016)

Depth = Repeated Compositions



Beyond Linear Models

Linear models

- Can be fit efficiently (via convex optimization)
- Limited model capacity

Alternative:

$$f(x) = w^T \boldsymbol{f}(x)$$

Where ϕ is a *non-linear transform*

Traditional ML

Manually engineer ϕ

- Domain specific, enormous human effort

Generic transform

- Maps to a higher-dimensional space
- Kernel methods: e.g. RBF kernels
- Over fitting: does not generalize well to test set
- Cannot encode enough prior information

Deep Learning

- Directly learn ϕ

$$f(x;q) = w^T f(x;q)$$

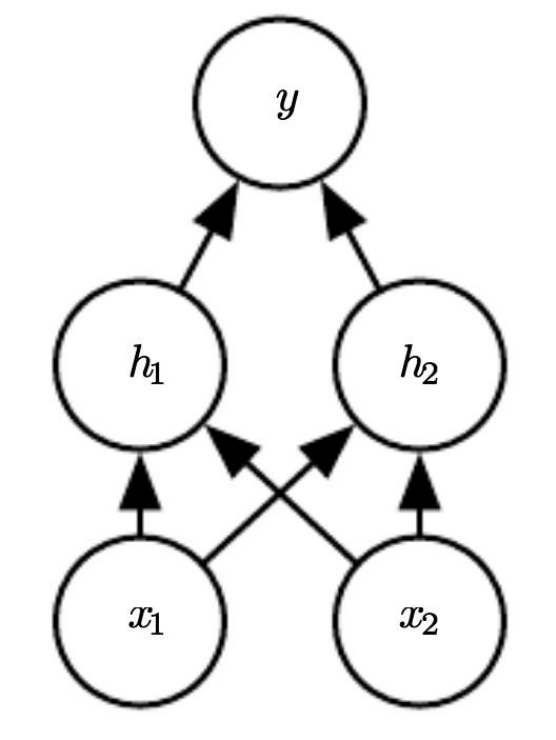
where θ are parameters of the transform

- ϕ defines hidden layers
- Non-convex optimization
- Can encode prior beliefs, generalizes well

Neural Networks

Hand-written digit recognition: MNIST data

Example: Learning XOR



$$h_1 = s(w_1^T x + c_1)$$

$$h_2 = s(w_2^T x + c_2)$$

$$y = s(w^T h + b)$$

where,

$$s(z) = \max\{0, z\}$$

Design Choices

Cost function

Output units

Hidden units

Architecture

Optimizer

Cost Function

Cross-entropy between training data and model distribution (i.e. negative log-likelihood)

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

Do not need to design separate cost functions

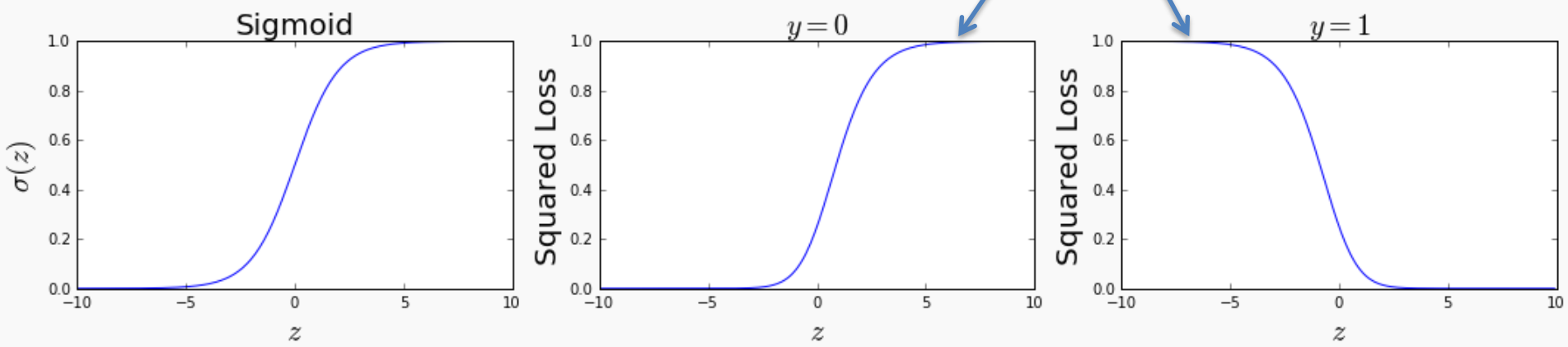
Gradient of cost function must be large enough

Cost Function

Example: sigmoid output + squared loss

$$s(z) = \frac{1}{1+e^{-z}} \qquad L_{sq}(y,z) = (y - s(z))^2$$

Flat surfaces

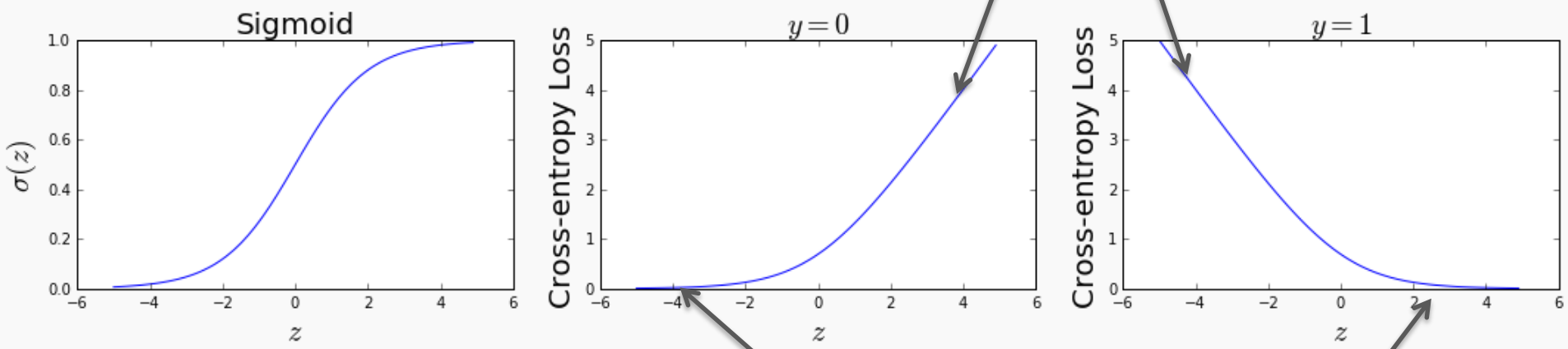


Cost Function

Example: sigmoid output + cross-entropy loss

$$L_{ce}(y,z) = -(y\log(z) + (1-y)\log(1-z))$$

log undoes exp



Saturates only when the model makes correct predictions

Design Choices

Cost function

Output units

Hidden units

Architecture

Optimizer

Output Units

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

(Goodfellow 2017)

Softmax Output

Discrete / Multinoulli output distribution

For output scores $z_1, \ldots, z_n$

Log-likelihood undoes exp

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$

$$\approx z_i - \max_j z_j$$

Design Choices

Cost function

Output units

Hidden units

Architecture

Optimizer

Hidden Units

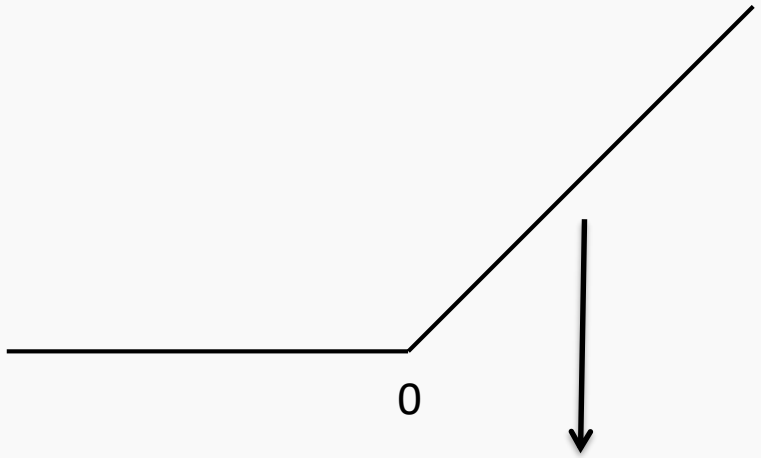
$$\mathbf{h} = g(\mathbf{W}^T x + \mathbf{b})$$

with activation function g

- Ensure gradients remain large through hidden unit
- Preferred: piece-wise linear activation
- Avoid sigmoid/tanh activation
  - Do not provide useful gradient info when they saturate

ReLU

Rectified Linear Units



$$g(z) = \max\{0, z\}$$

Gradient is 1 whenever unit is active

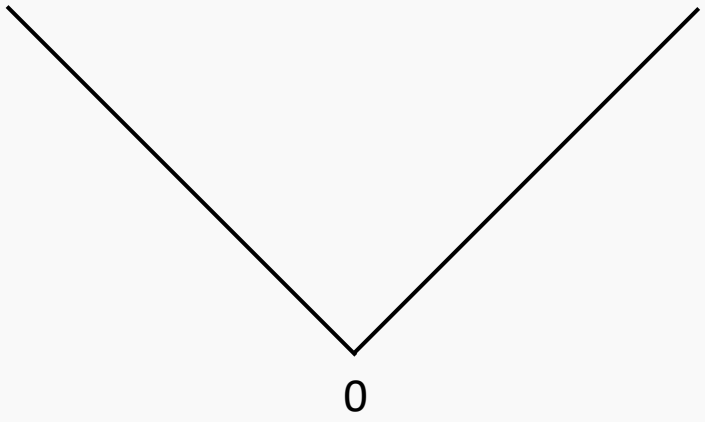
- More useful for learning compared to sigmoid
- No useful gradient information when z<0

Generalized ReLU

Generalization: For $\alpha_i > 0$

$$g(z;a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$$

E.g. Absolute value ReLU: $a_i = -1 \Rightarrow g(z) = |z|$

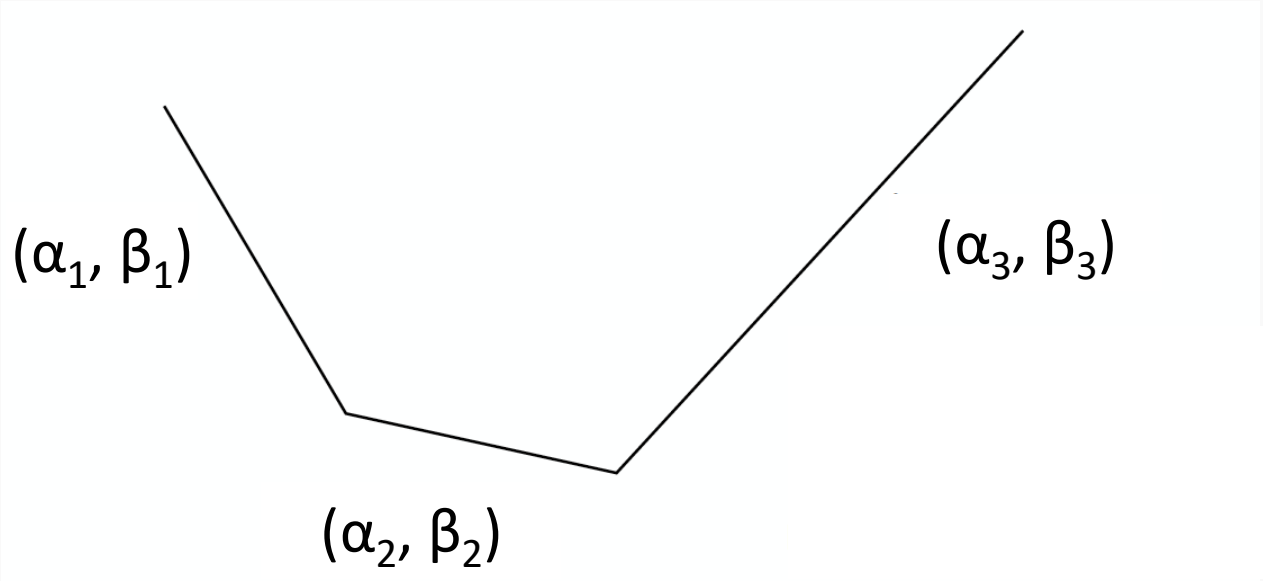


Maxout

Directly learn the activation function

– Max of k linear functions

$$g(z) = \max_{i \in \{1, \ldots, k\}} a_i z_i + b_i$$



Design Choices

Cost function

Output units

Hidden units

Architecture

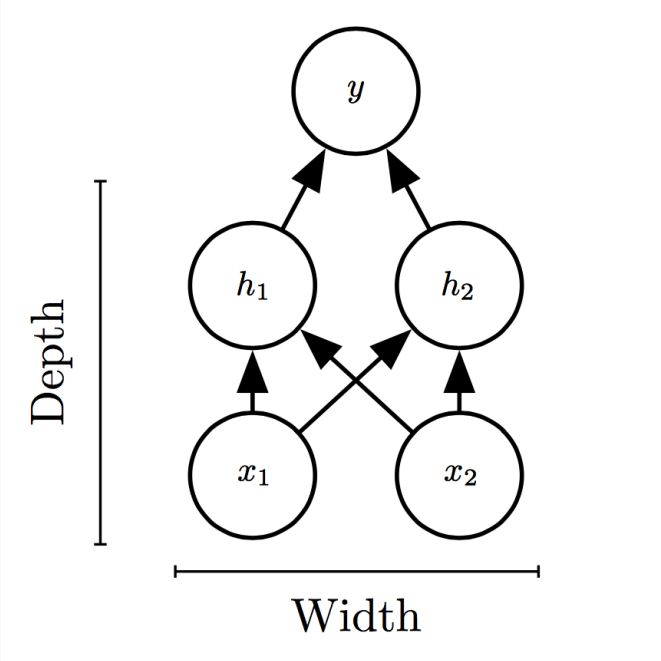
Optimizer

Universal Approximation Theorem

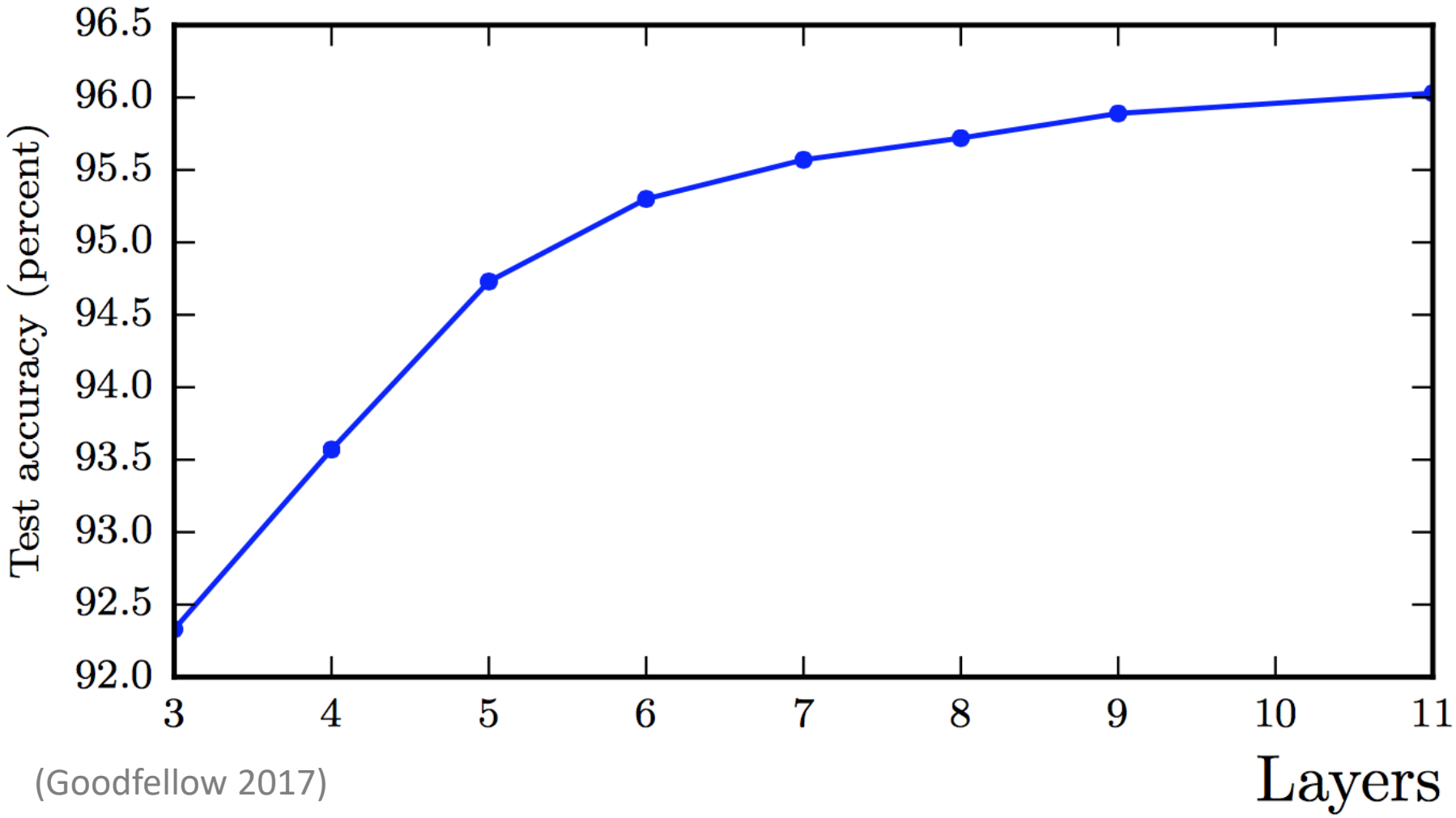
One hidden layer is enough to *represent* an approximation of any function to an arbitrary degree of accuracy

So why deeper?

- Shallow net may need (exponentially) more width
- Shallow net may overfit more

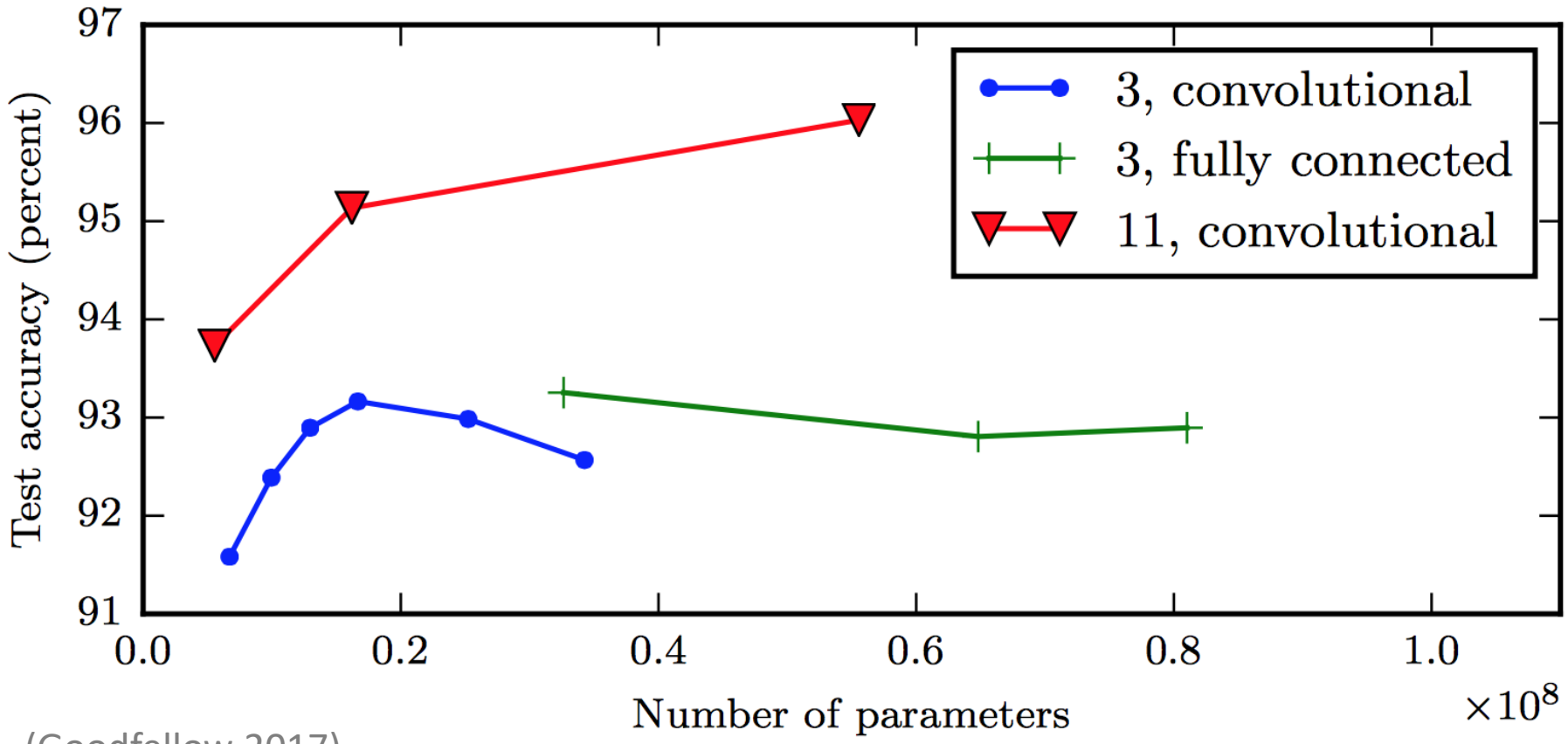


Better Generalization with Depth



(Goodfellow 2017)

Large, Shallow Nets Overfit More



(Goodfellow 2017)

Design Choices

Cost function

Output units

Hidden units

Architecture

Optimizer

Backpropagation

Avoids repeated sub-expressions

Uses dynamic programming (table filling)

Trades-off memory for speed

Backprop: Arithmetic

Jacobian-gradient products

$$\mathbf{z} = g(\mathbf{x})$$
$$y = f(\mathbf{z})$$

$$\begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial x_n} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix} \times \begin{bmatrix} \frac{\partial y}{\partial z_1} \\ \vdots \\ \frac{\partial y}{\partial z_m} \end{bmatrix}$$

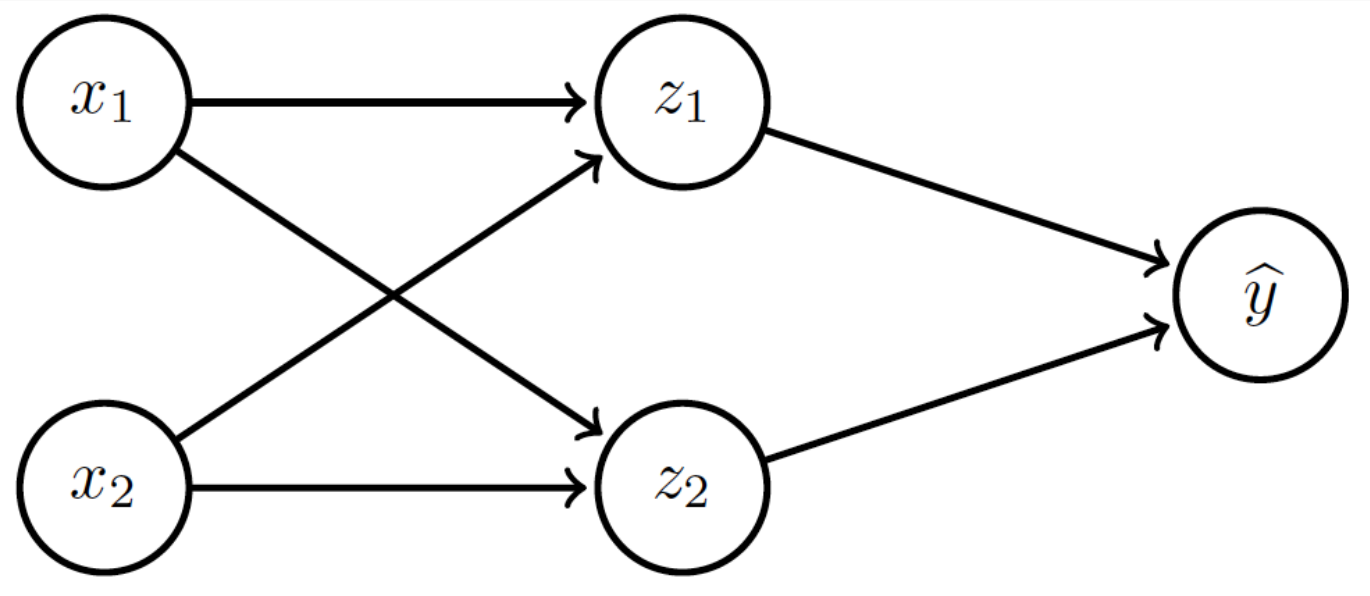
grad w.r.t. **x** Jacobian of 'g' grad w.r.t. **z**

$$\nabla_{\mathbf{x}} y = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{z}} y$$

Apply recursively!

Backprop: Overview

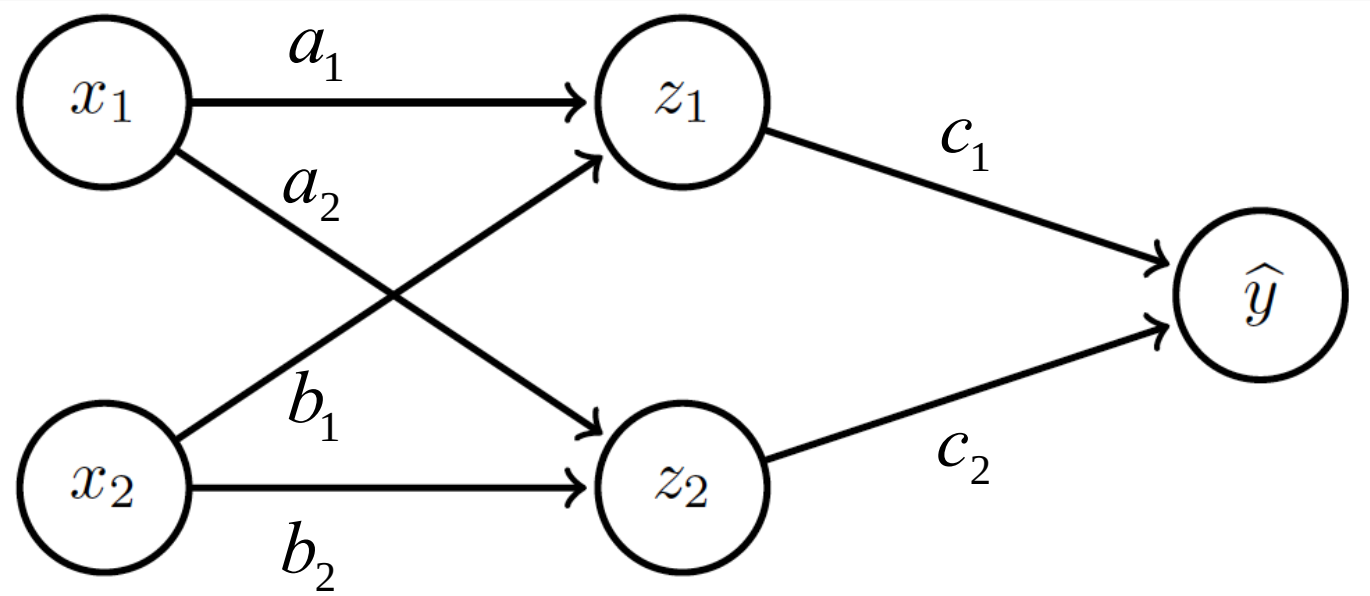
Forward prop: compute activations



Compute loss

Back-prop: compute derivatives

Backprop: Example



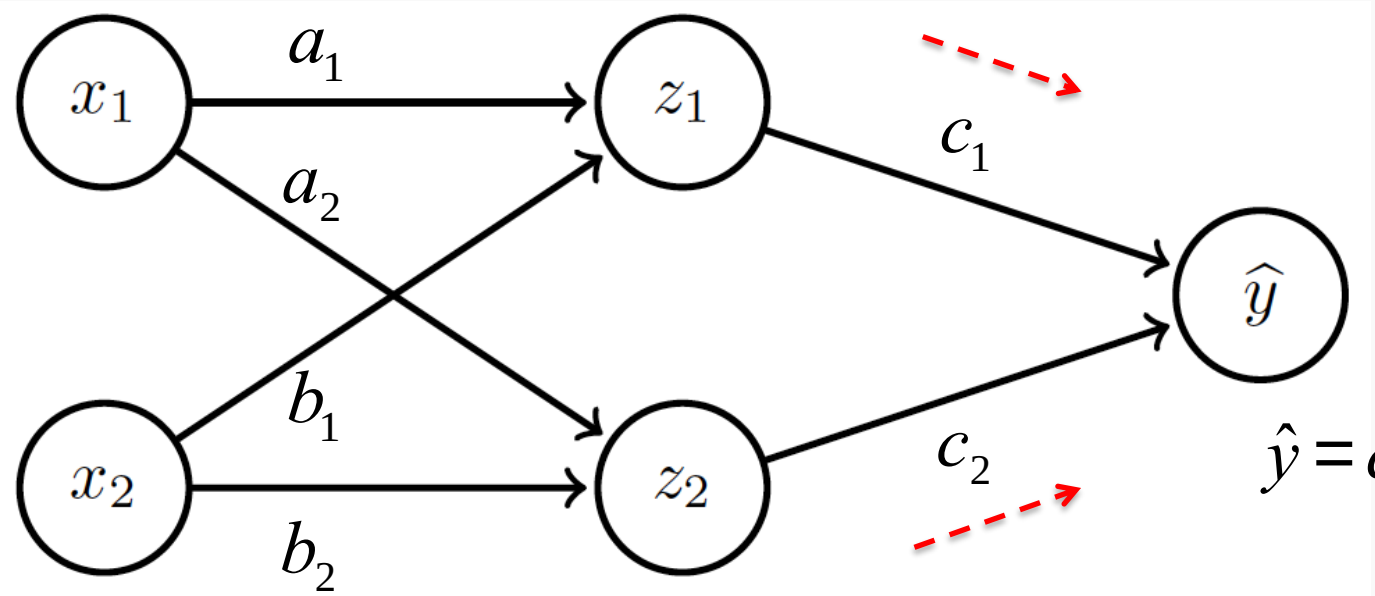
Linear activation functions

No bias

Squared loss

Backprop: Example

$$z_1 = a_1 x_1 + a_2 x_2$$

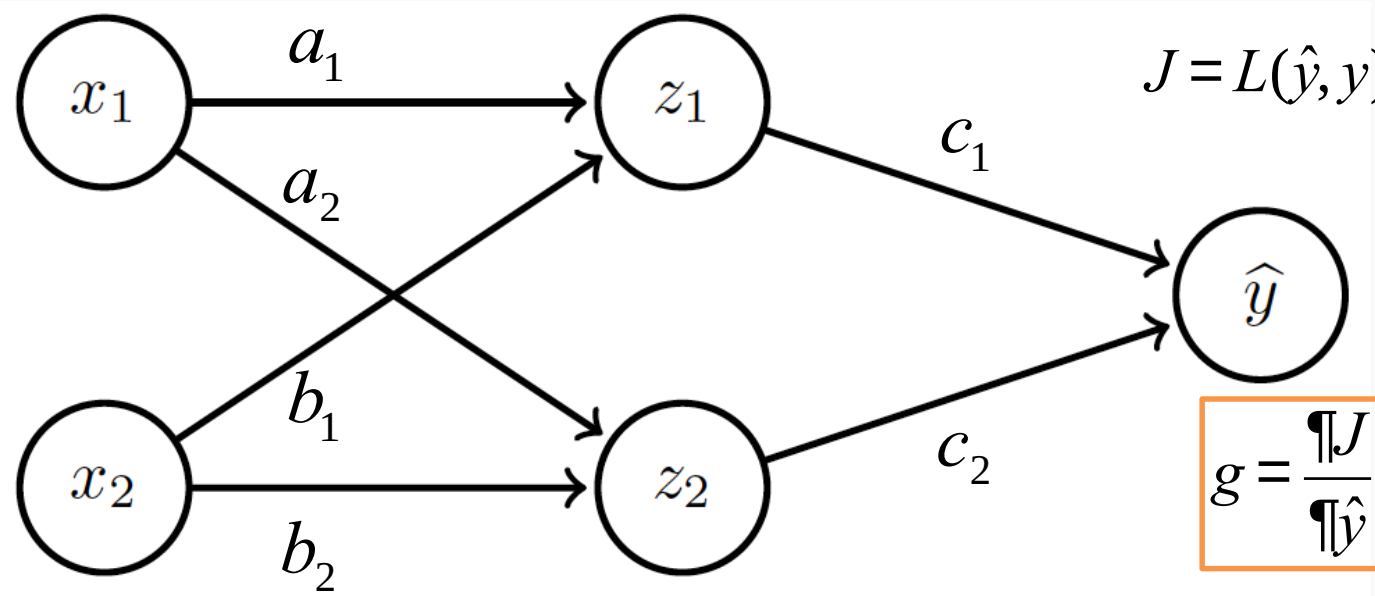


$$\hat{y} = c_1 z_1 + c_2 z_2$$

$$z_2 = b_1 x_1 + b_2 x_2$$

Forward prop: Propagate activations to output layer

Backprop: Example



$$J = L(\hat{y}, y) = \tfrac{1}{2}(\hat{y} - y)^2$$

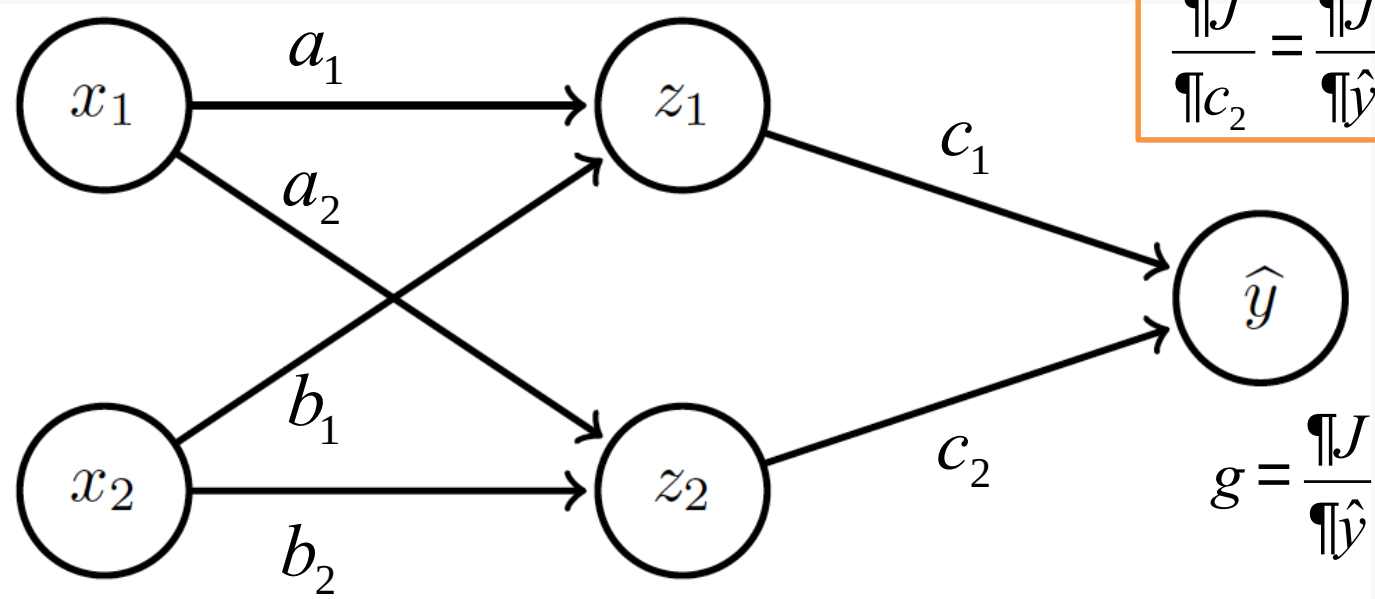
$$\boxed{g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y}$$

Backward prop: Compute loss and its derivative

Backprop: Example

$$\frac{\partial J}{\partial c_1} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial c_1}$$

$$\frac{\partial J}{\partial c_2} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial c_2}$$



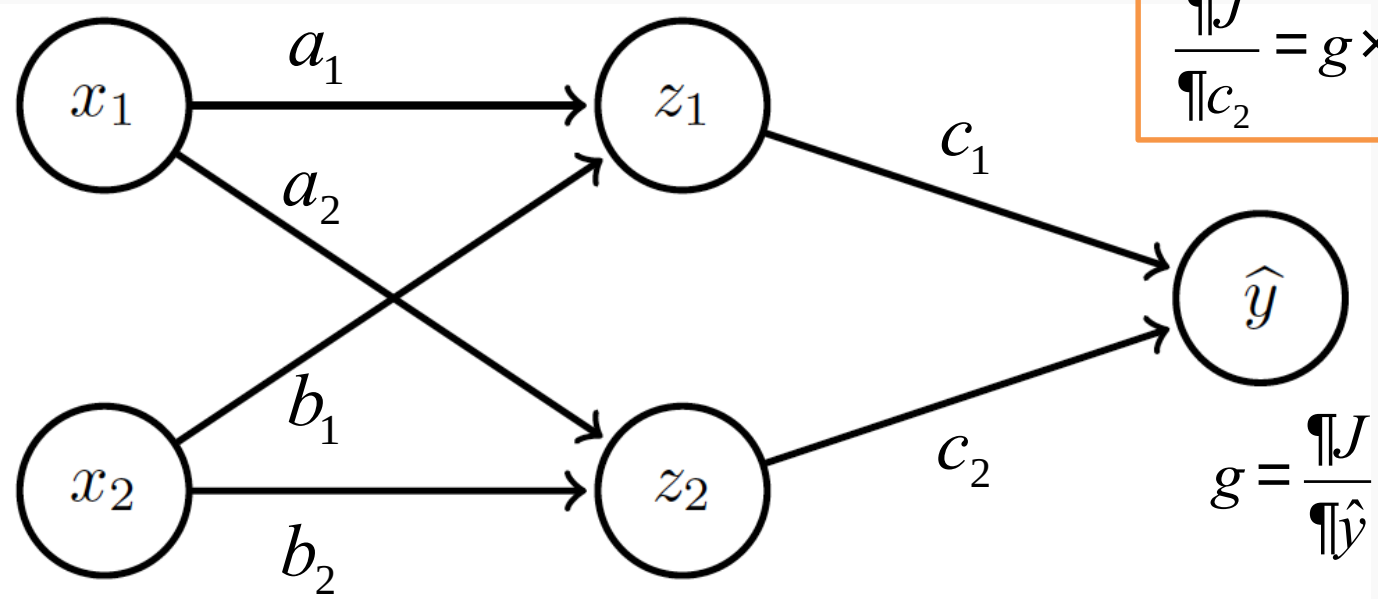
$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Backward prop: Compute derivatives w.r.t. weights c_1 and c_2

Backprop: Example

$$\frac{\partial J}{\partial c_1} = g \times z_1$$

$$\frac{\partial J}{\partial c_2} = g \times z_2$$

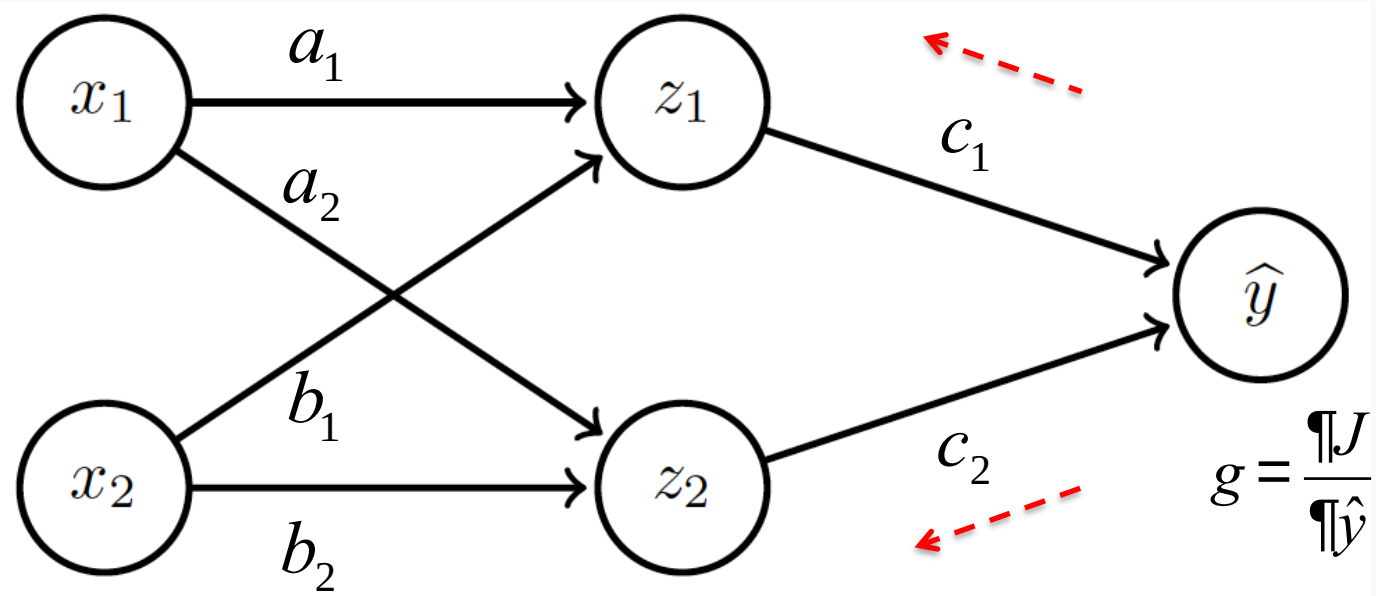


$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Backward prop: Compute derivatives w.r.t. weights c_1 and c_2

Backprop: Example

$$g_1 = \frac{\partial J}{\partial z_1} = g c_1$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2 = \frac{\partial J}{\partial z_2} = g c_2$$

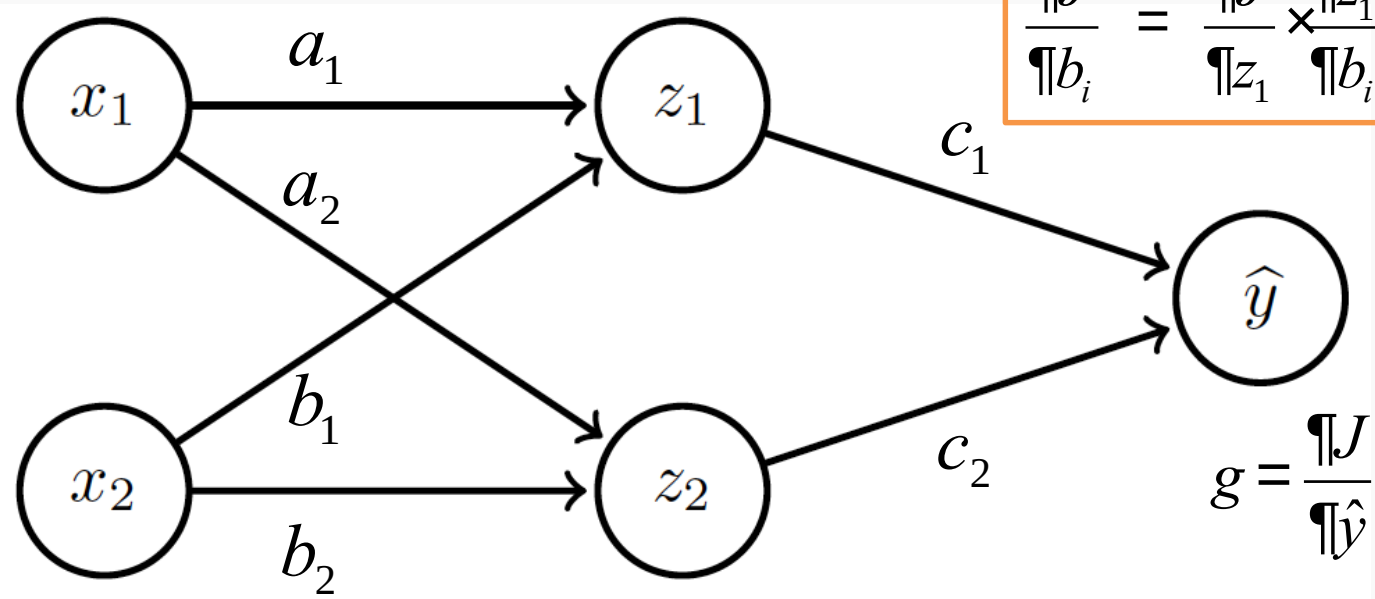
Backward prop: Propagate derivative to hidden layer

Backprop: Example

$$\frac{\partial J}{\partial a_i} = \frac{\partial J}{\partial z_1} \times \frac{\partial z_1}{\partial a_i} + \frac{\partial J}{\partial z_2} \times \frac{\partial z_2}{\partial a_i}$$

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial z_1} \times \frac{\partial z_1}{\partial b_i} + \frac{\partial J}{\partial z_2} \times \frac{\partial z_2}{\partial b_i}$$

$$g_1 = \frac{\partial J}{\partial z_1} = g c_1$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2 = \frac{\partial J}{\partial z_2} = g c_2$$

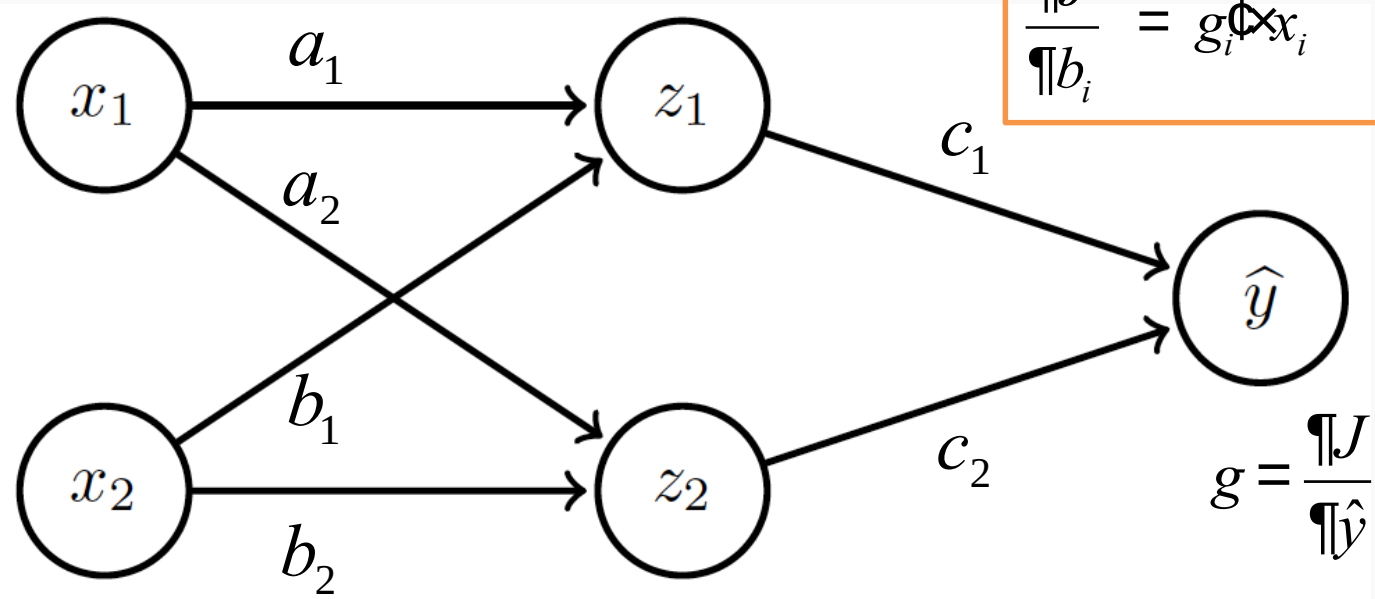
Backward prop: Compute derivatives w.r.t. weights a_1, a_2, b_1 and b_2

Backprop: Example

$$\frac{\partial J}{\partial a_i} = g_i \times x_i$$

$$\frac{\partial J}{\partial b_i} = g_i \times x_i$$

$$g_1 = \frac{\partial J}{\partial z_1} = g c_1$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2 = \frac{\partial J}{\partial z_2} = g c_2$$

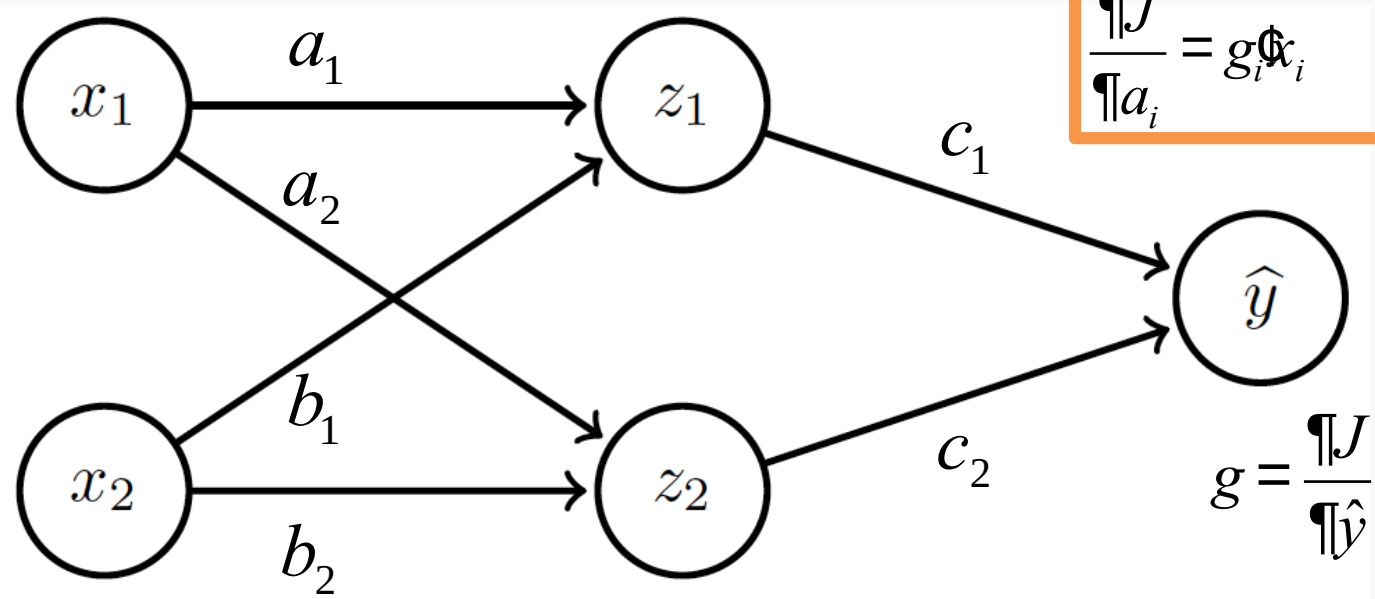
Backward prop: Compute derivatives w.r.t. weights a_1, a_2, b_1 and b_2

Backprop: Example

$$g_1 = \frac{\partial J}{\partial z_1} = g c_1$$

$$\frac{\partial J}{\partial c_1} = g z_1 \qquad \frac{\partial J}{\partial c_2} = g z_2$$

$$\frac{\partial J}{\partial a_i} = g_i x_i \qquad \frac{\partial J}{\partial b_i} = g_i x_i$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2 = \frac{\partial J}{\partial z_2} = g c_2$$

Regularization

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error

Outline

Norm Penalties

Early Stopping

Data Augmentation

Bagging

Dropout

Outline

Norm Penalties

Early Stopping

Data Augmentation

Bagging

Dropout

Norm Penalties

Optimize:

$$J(q; X, y) + a\,\mathrm{W}(q)$$

Biases not penalized

L_2 regularization:

- decays weights
- MAP estimation with Gaussian prior

$$\mathrm{W}(q) = \frac{1}{2}\|\mathbf{w}\|_2^2$$

L_1 regularization:

- encourages sparsity
- MAP estimation with Laplacian prior

$$\mathrm{W}(q) = \|\mathbf{w}\|_1$$

Norm Penalties as Constraints

$$\min_{\mathrm{W}(q) \le K} J(q; X, y)$$

Useful if *K* is known in advance

Optimization:

- Construct Lagrangian and apply gradient descent
- Projected gradient descent

Outline

Norm Penalties

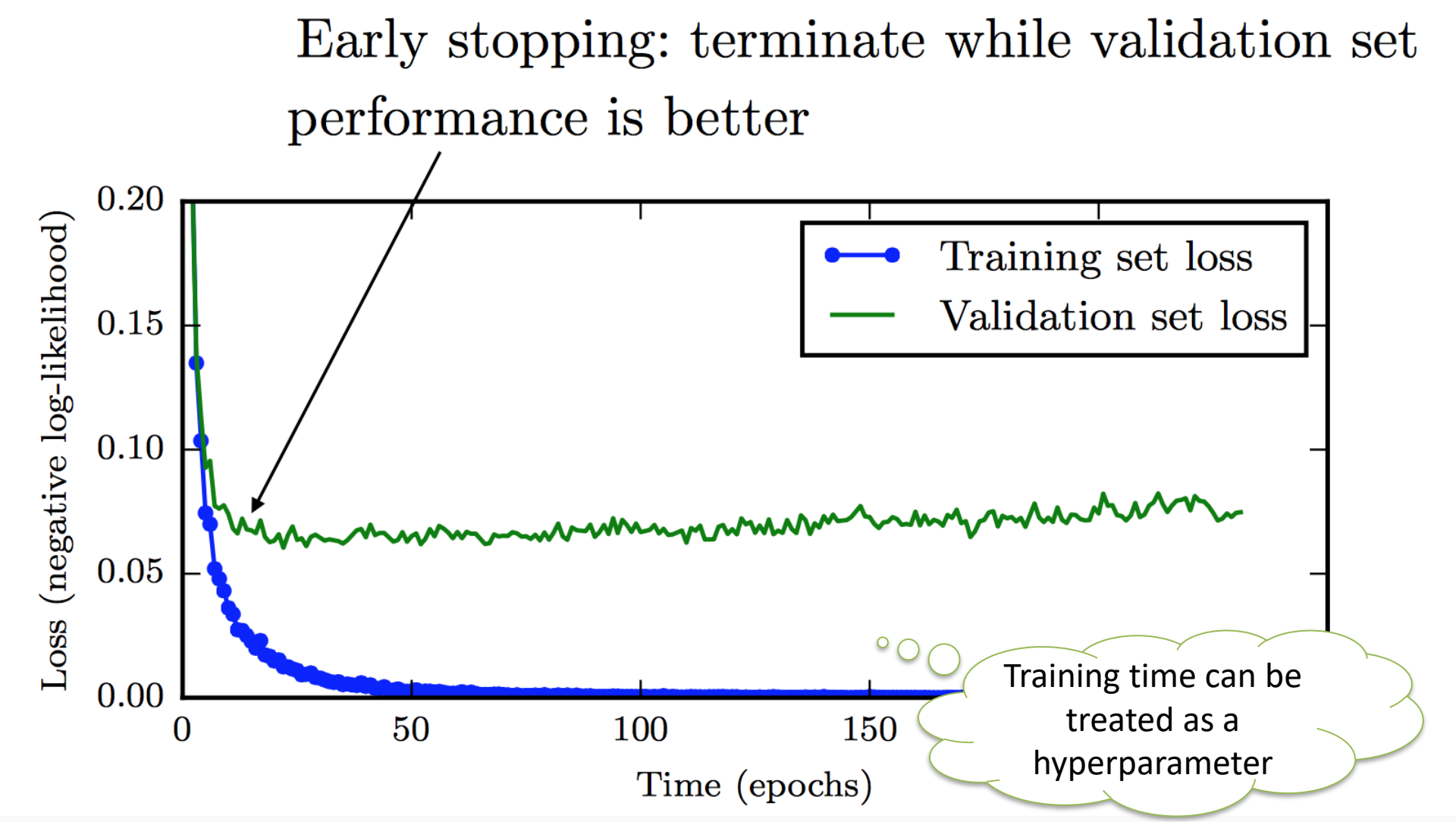
Early Stopping

Data Augmentation

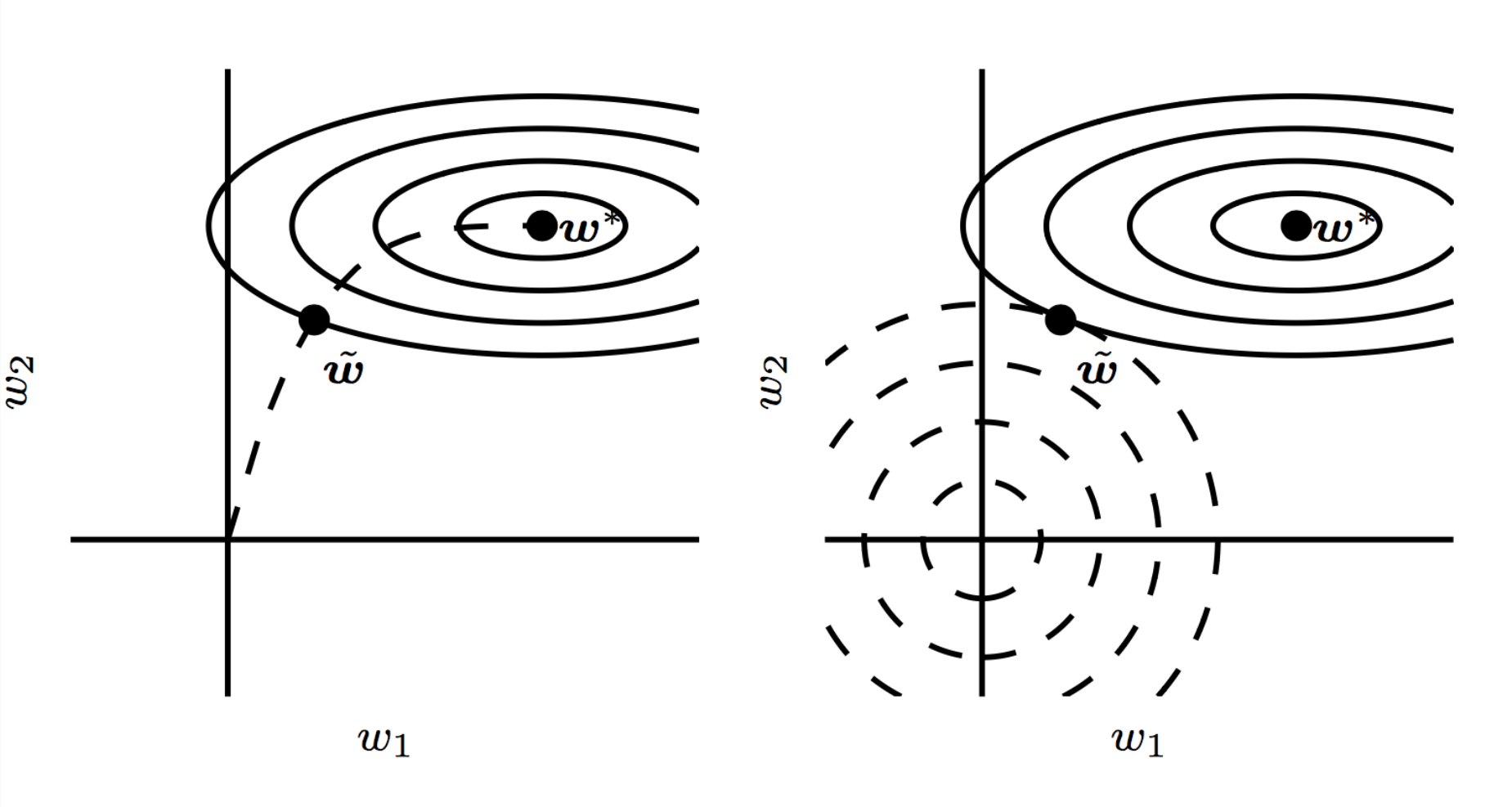
Bagging

Dropout

Early Stopping



Early Stopping



Outline

Norm Penalties

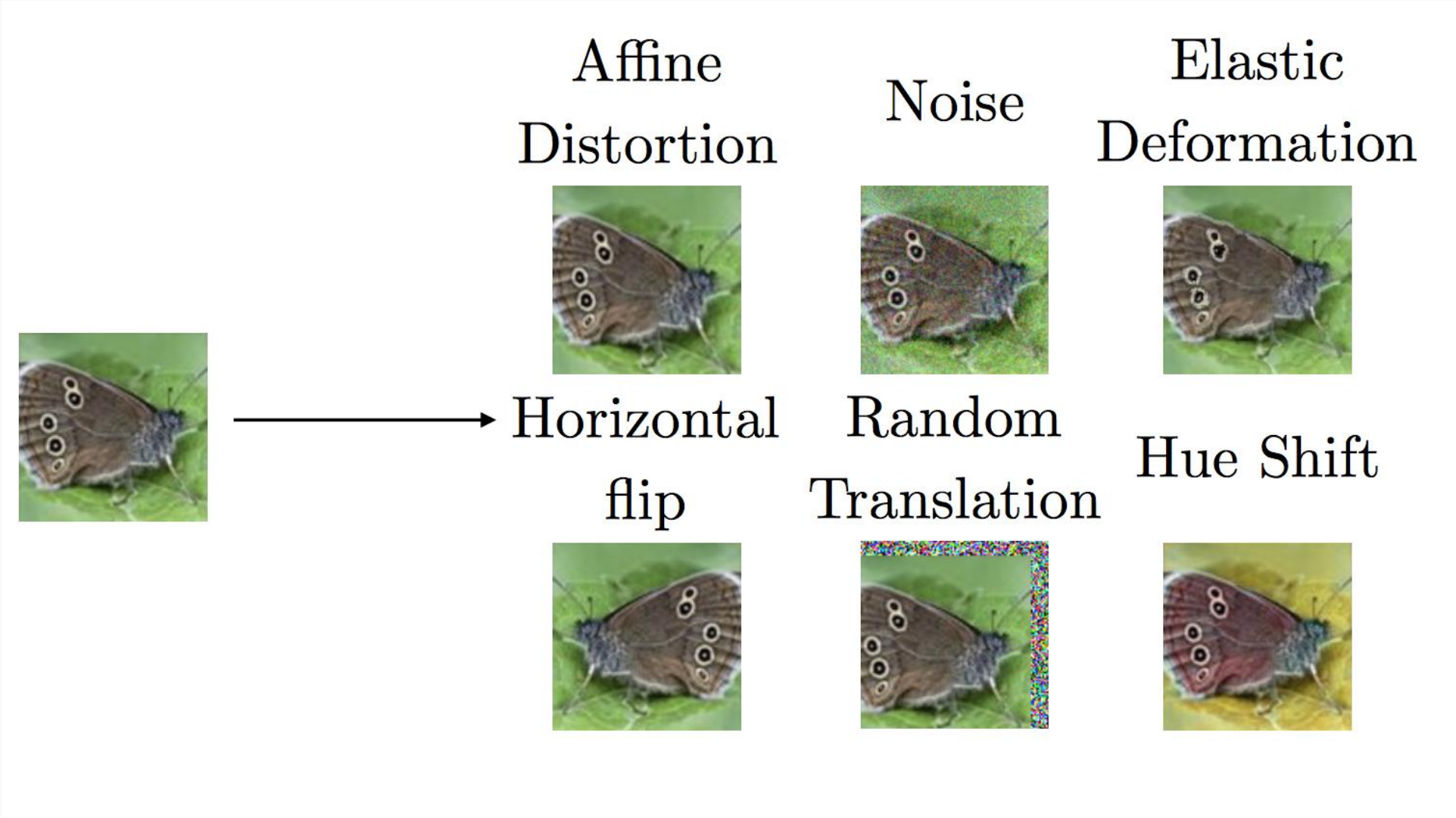
Early Stopping

Data Augmentation

Bagging

Dropout

Data Augmentation



Outline

Norm Penalties

Early Stopping

Data Augmentation

Bagging

Dropout

Noise Robustness

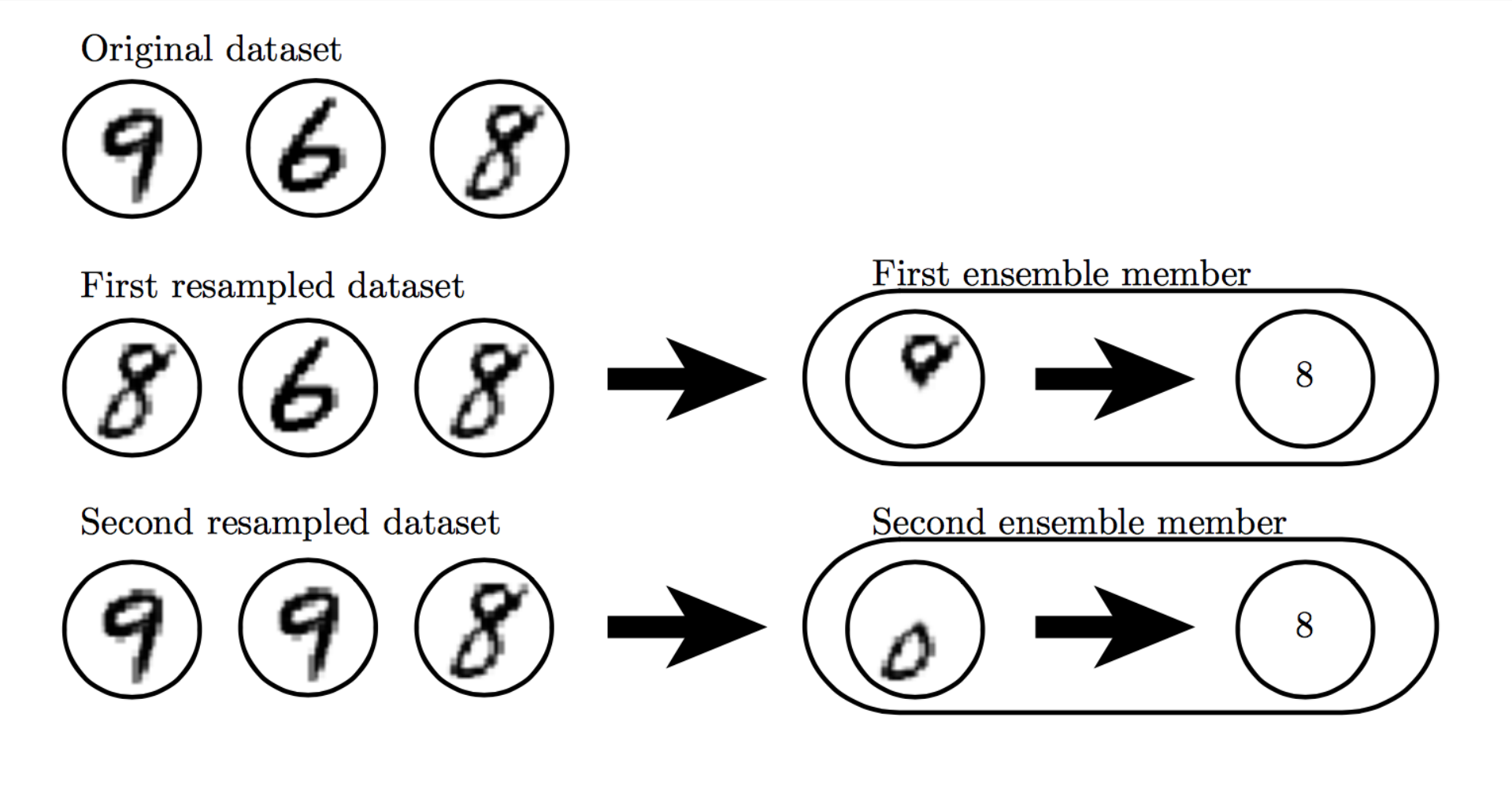
Random perturbation of network weights

- Gaussian noise: Equivalent to minimizing loss with regularization term
- Encourages smooth function: small perturbation in weights leads to small changes in output

Injecting noise in output labels

- Better convergence: prevents pursuit of hard probabilities

$$\mathbf{E}\left[\left\|\nabla_W y(x)\right\|\right]$$


Original dataset
First resampled dataset
First ensemble member
8
Second resampled dataset
Second ensemble member
8

Outline

Norm Penalties

Early Stopping

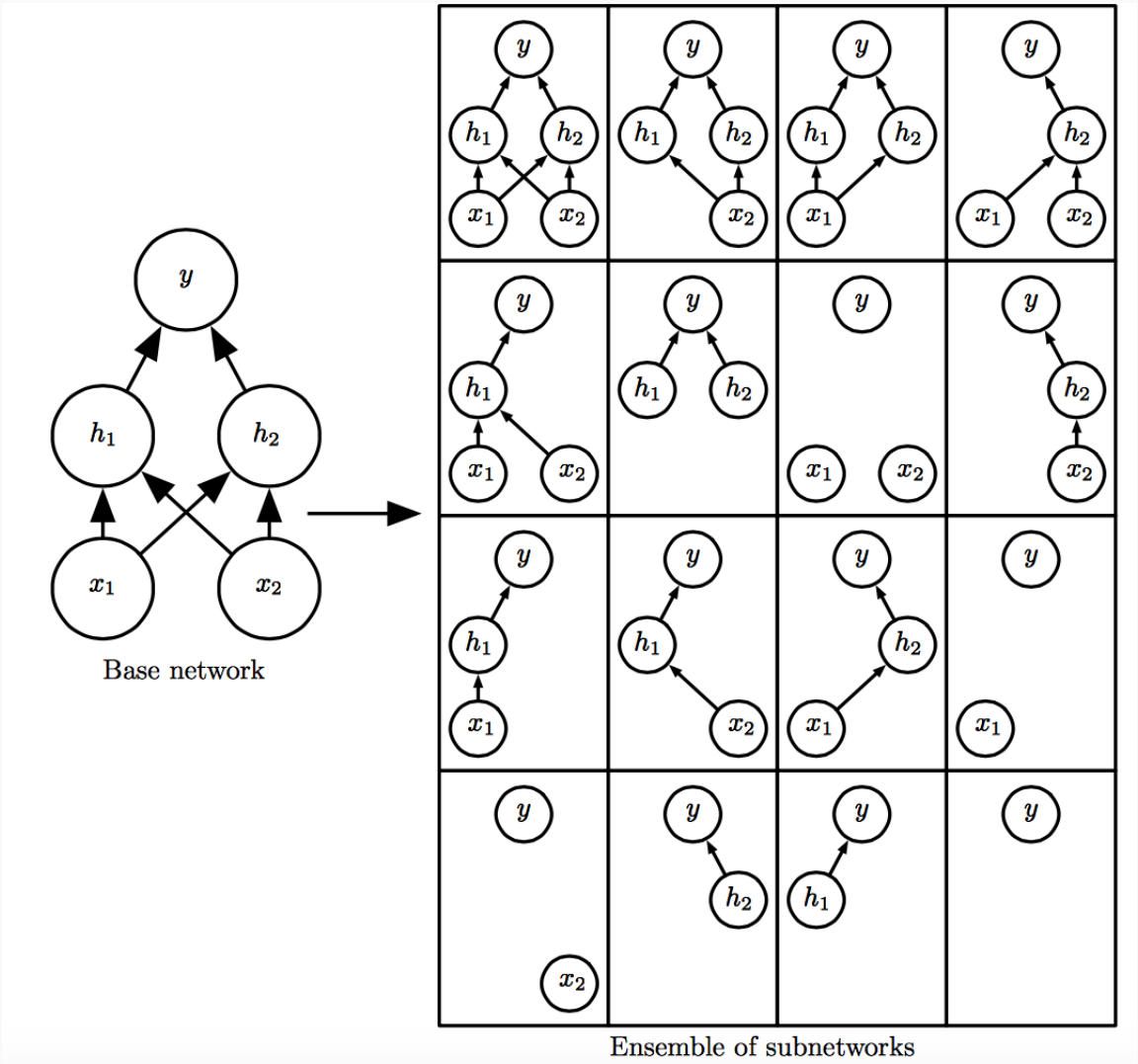
Data Augmentation

Bagging

Dropout

Dropout

Train all sub-networks obtained by removing non-output units from base network



Dropout: Stochastic GD

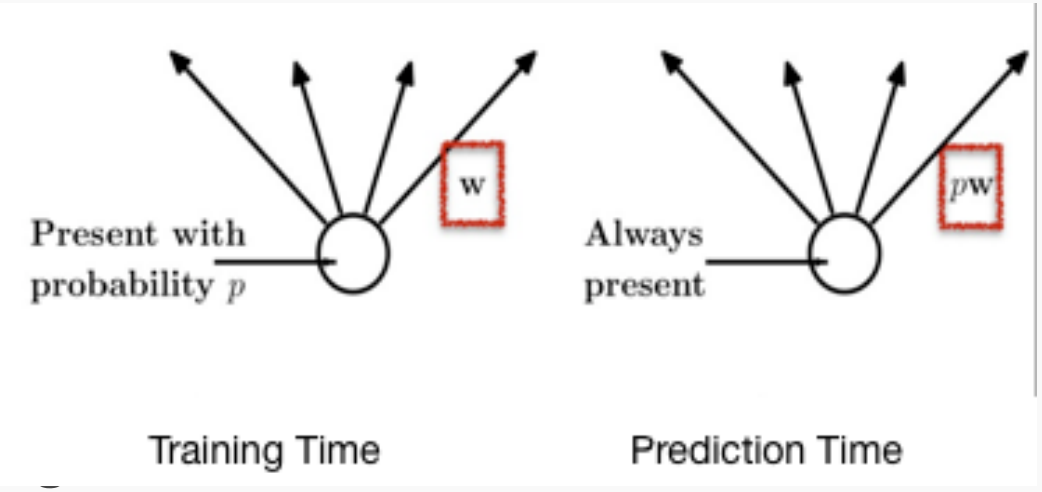
For each new example/mini-batch:

- Randomly sample a binary mask μ independently, where μ_i indicates if input/hidden node i is included
- Multiply output of node i with μ_i, and perform gradient update

Typically, an input node is included with prob=0.8, hidden node with prob=0.5

Dropout: Weight Scaling

During prediction time use all units, but scale weights with probability of inclusion

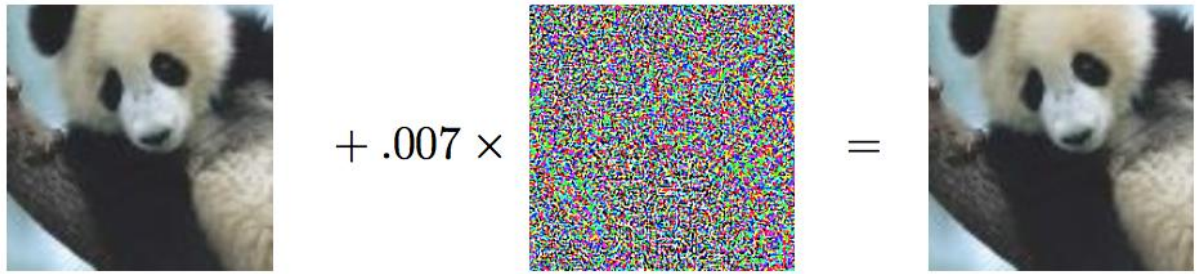


Approximates the followi

$$\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \boldsymbol{x}, \boldsymbol{\mu})}$$

Cristina Scheau (2016)

Adversarial Examples



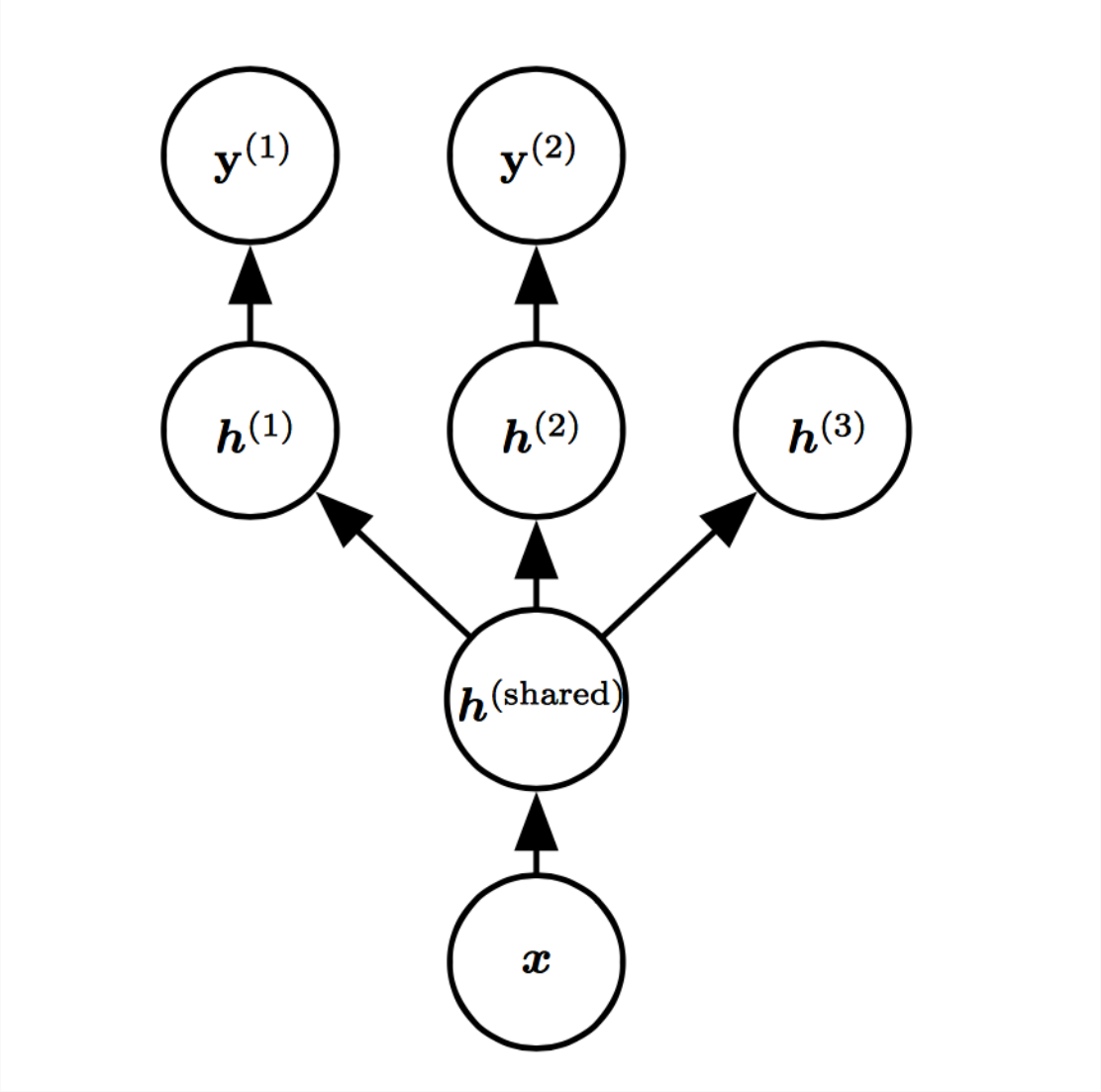
$+ .007 \times$ $=$

$\boldsymbol{x}$	$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$	$\boldsymbol{x} + \epsilon\,\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
$y =$"panda" w/ 57.7% confidence	"nematode" w/ 8.2% confidence	"gibbon" w/ 99.3 % confidence

Training on adversarial examples is mostly intended to improve security, but can sometimes provide generic regularization.

deeplearningbook.org

Multi-task Learning



Goodfellow et al. (2016)

Optimization

Learning vs. Optimization

Goal of learning: minimize generalization error

In practice, empirical risk minimization:

$$J(q) = \mathbf{E}_{(x,y)\sim p_{data}}\left[L(f(x;q),y)\right]$$

$$\hat{J}(q) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};q),y^{(i)})$$

Quantity optimized different from the quantity we care about

Batch vs. Stochastic Algorithms

Batch algorithms

- Optimize empirical risk using exact gradients

Stochastic algorithms

- Estimates gradient from a small random sample

$$\nabla J(q) = \mathbf{E}_{(x,y)\sim p_{data}}\left[\nabla L(f(x;q), y)\right]$$

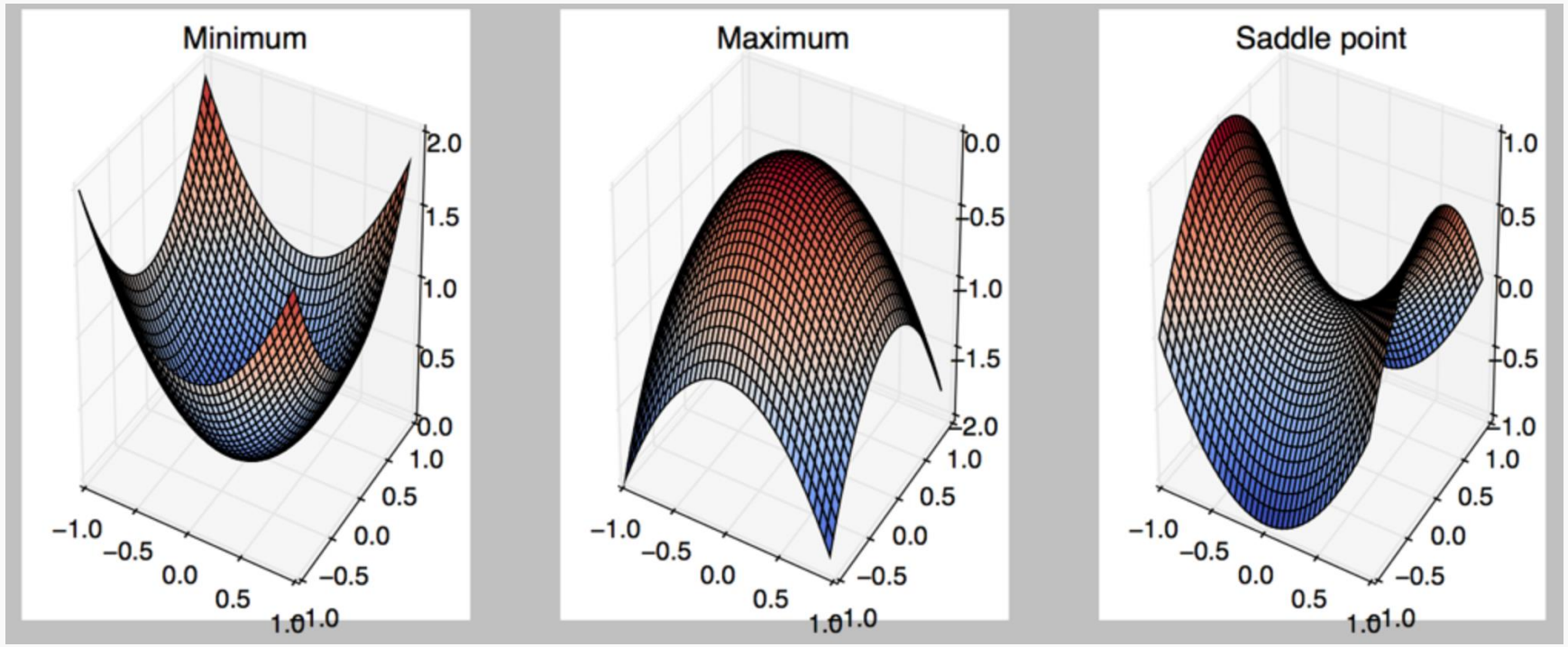
Large mini-batch: gradient computation expensive

Small mini-batch: greater variance in estimate, longer steps for convergence

Critical Points

Points with zero gradient

2nd-derivate (Hessian) determines curvature



Goodfellow et al. (2016)

Stochastic Gradient Descent

Take small steps in direction of negative gradient

Sample *m* examples from training set and compute:

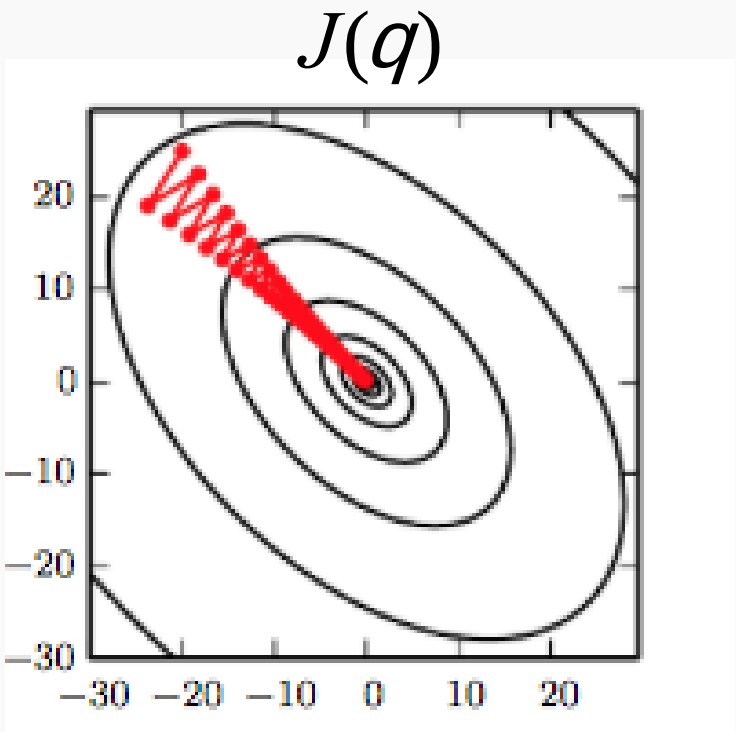
Update parameters:

$$g = \frac{1}{m}\sum_i \nabla L(f(x^{(i)};q), y^{(i)})$$

$$q = q - e_k g$$

In practice: shuffle training set once and pass through multiple times

Stochastic Gradient Descent



Oscillations because updates do not exploit curvature information

Goodfellow et al. (2016)

Outline

Challenges in Optimization

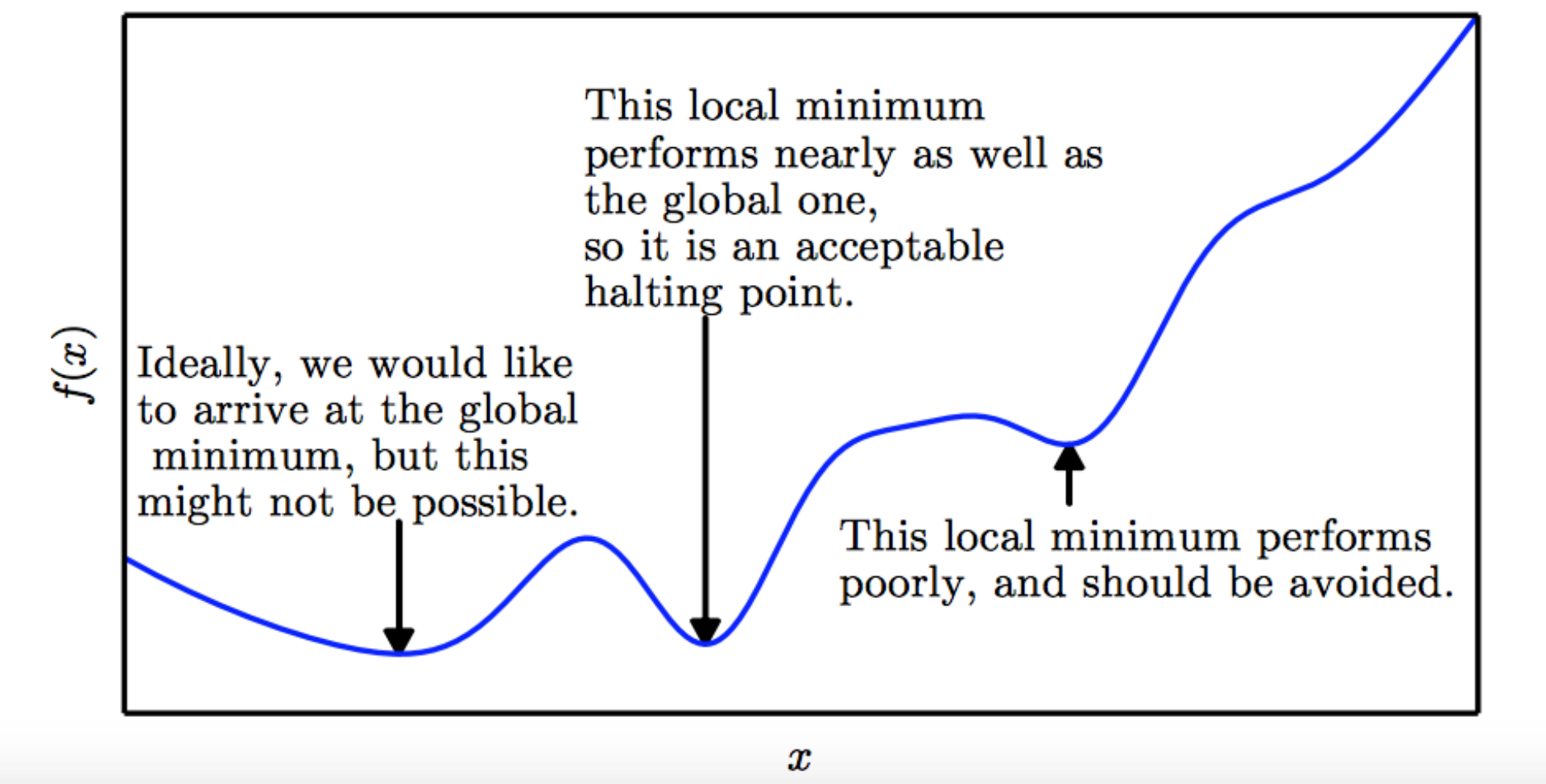
Momentum

Adaptive Learning Rate

Parameter Initialization

Batch Normalization

Local Minima



Goodfellow et al. (2016)

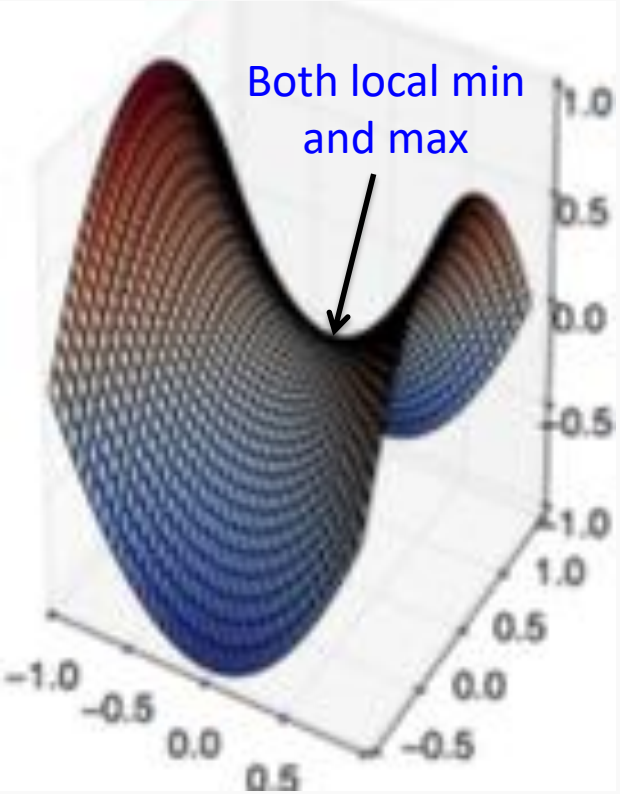
Local Minima

Old view: local minima is major problem in neural network training

Recent view:

- For sufficiently large neural networks, most local minima incur low cost
- Not important to find true global minimum

Saddle Points



Recent studies indicate that in high dim, saddle points are more likely than local min

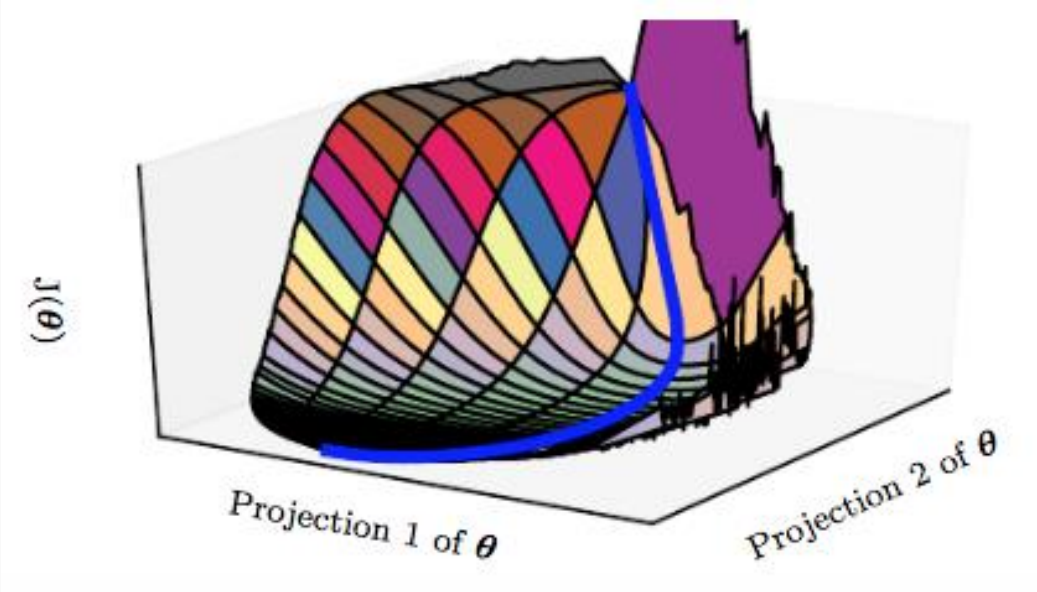
Gradient can be very small near saddle points

Goodfellow et al. (2016)

Saddle Points

SGD is seen to escape saddle points

– Moves down-hill, uses noisy gradients



Second-order methods get stuck

– solves for a point with zero gradient

Goodfellow et al. (2016)

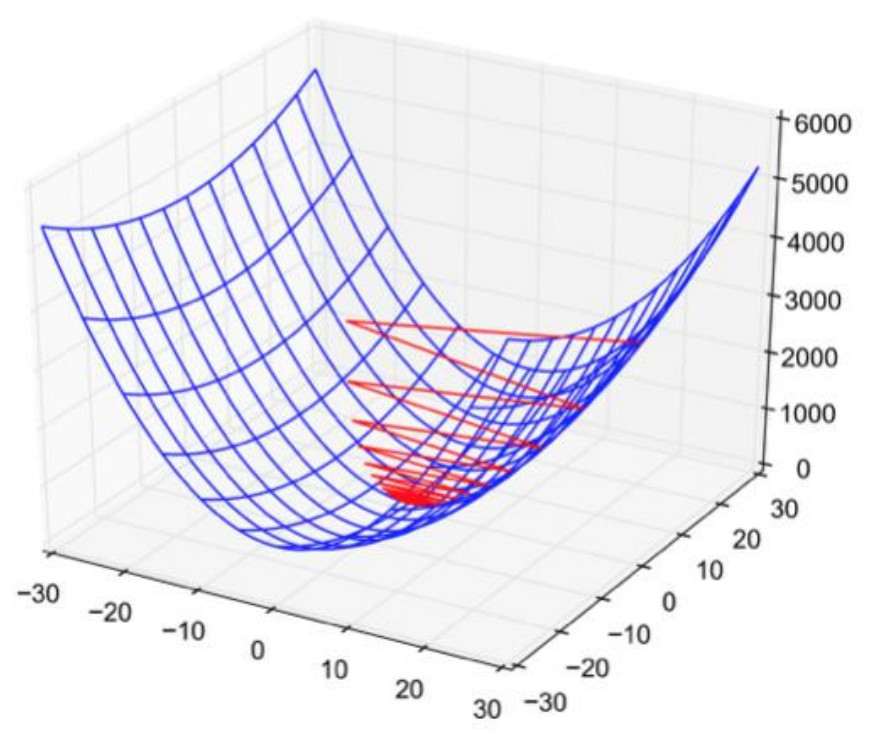
Poor Conditioning

Poorly conditioned Hessian matrix

- High curvature: small steps leads to huge increase

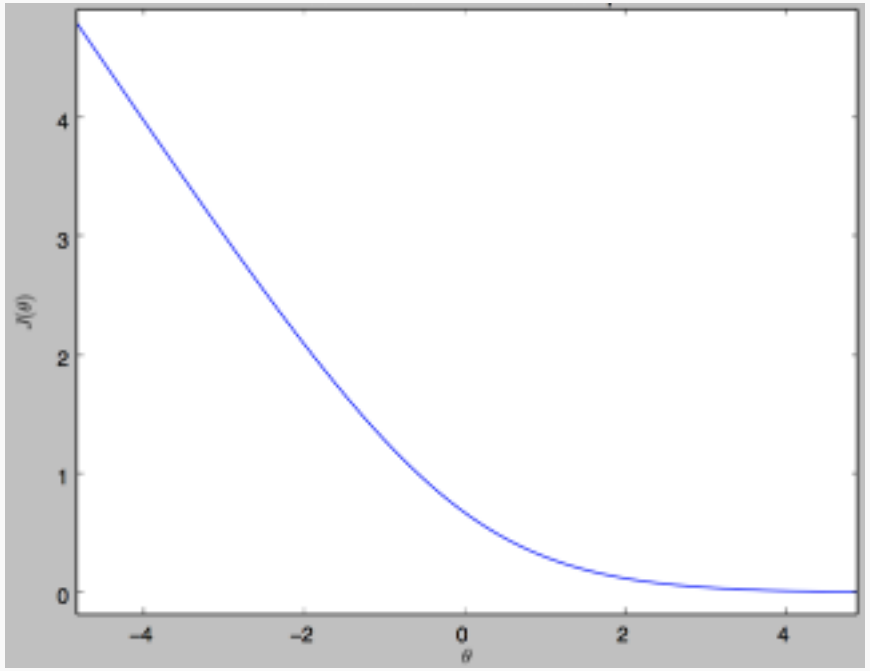
Learning is slow despite strong gradients

Oscillations slow down progress



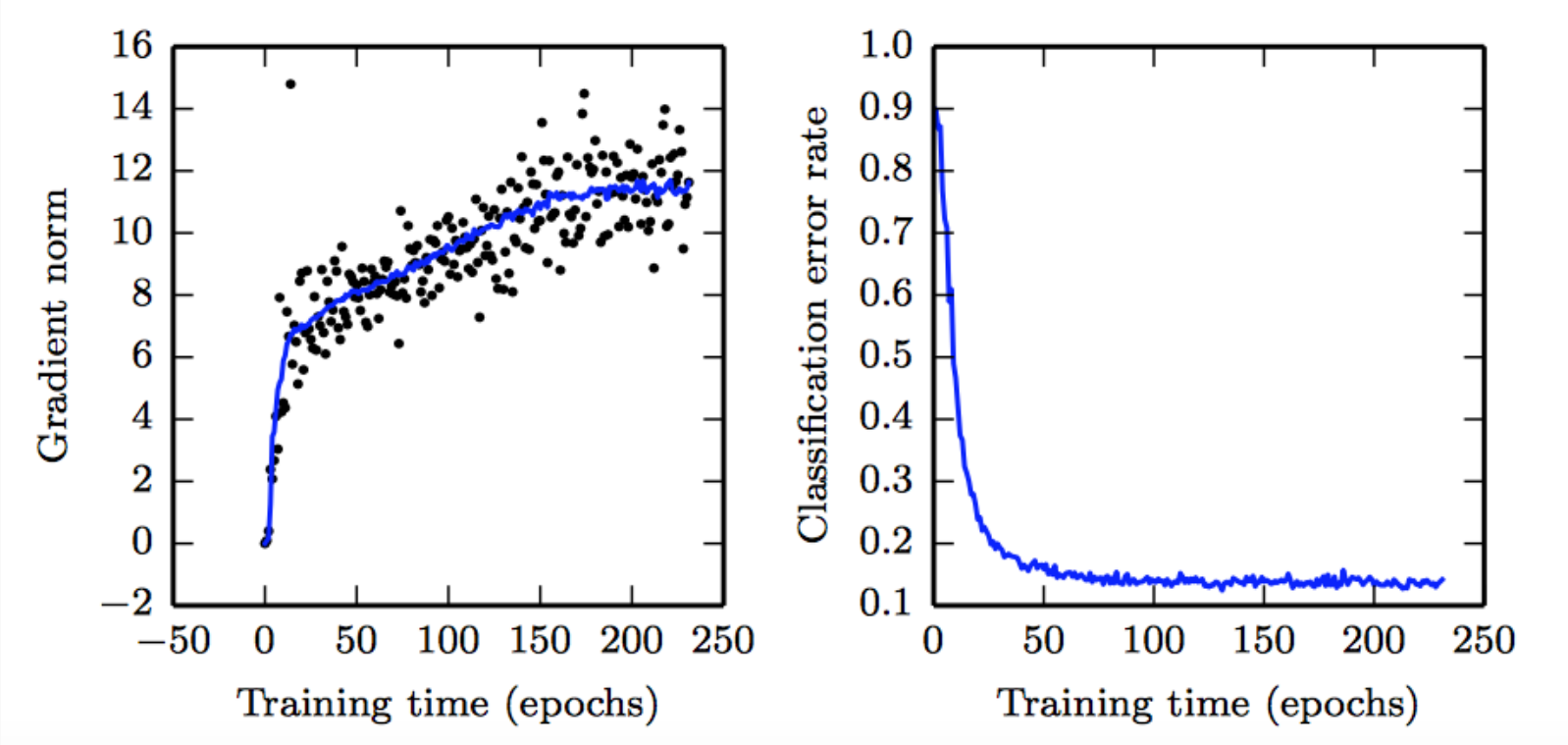
No Critical Points

Some cost functions do not have critical points



No Critical Points

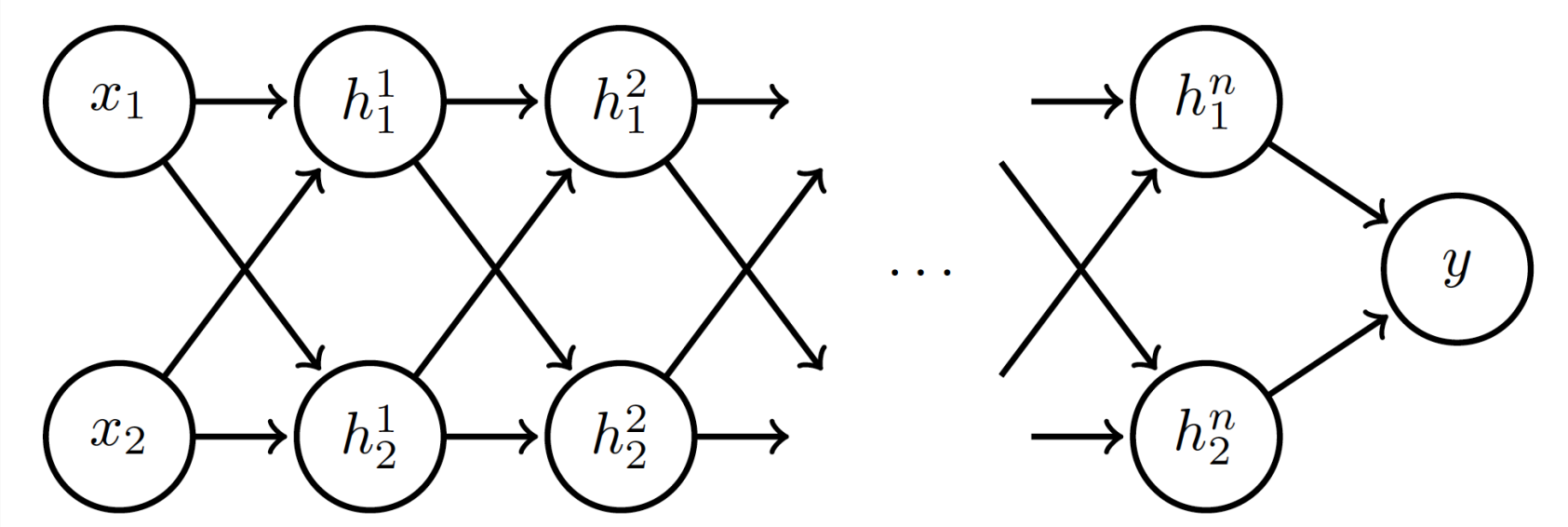
Gradient norm increases, but validation error decreases



Convolution Nets for Object Detection

Goodfellow et al. (2016)

Exploding and Vanishing Gradients



Linear activation

$$\mathbf{h}_1 = \mathbf{W}\mathbf{x}$$

$$\mathbf{h}_i = \mathbf{W}\mathbf{h}_{i-1}, \quad i = 2 \ldots n$$

$$y = \sigma(h_1^n + h_2^n), \quad \text{where } \sigma(s) = \frac{1}{1+e^{-s}}$$

Exploding and Vanishing Gradients

Suppose $\mathbf{W} = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$:

$$\begin{bmatrix} h_1^1 \\ h_2^1 \end{bmatrix} = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \cdots \quad \begin{bmatrix} h_1^n \\ h_2^n \end{bmatrix} = \begin{bmatrix} a^n & 0 \\ 0 & b^n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$y = s(a^n x_1 + b^n x_2)$$

$$\nabla y = s'(a^n x_1 + b^n x_2) \begin{bmatrix} n a^{n-1} x_1 \\ n b^{n-1} x_2 \end{bmatrix}$$

Exploding and Vanishing Gradients

Suppose $x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Case 1: $a = 1,\ b = 2$:

$$y \to 1, \quad \nabla y \to \begin{bmatrix} n \\ n2^{n-1} \end{bmatrix}$$

Explodes!

Case 2: $a = 0.5,\ b = 0.9$:

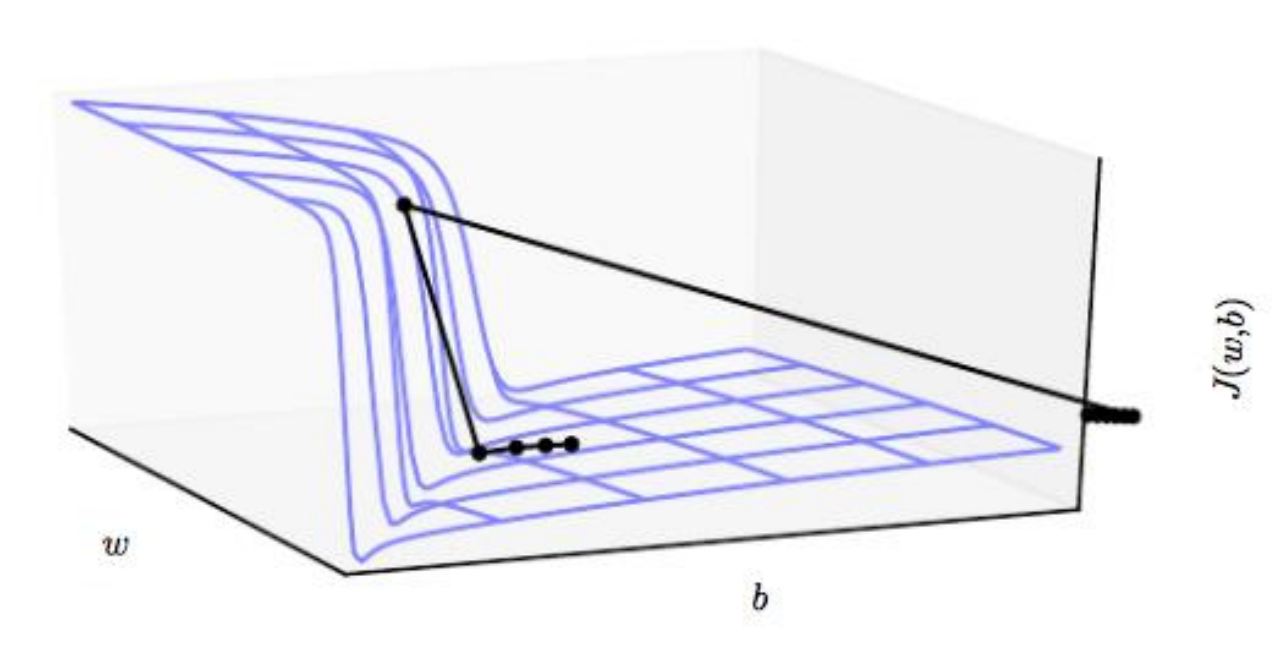
$$y \to 0, \quad \nabla y \to \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Vanishes!

Exploding and Vanishing Gradients

Exploding gradients lead to cliffs

Can be mitigated using gradient clipping



Goodfellow et al. (2016)

Outline

Challenges in Optimization

Momentum

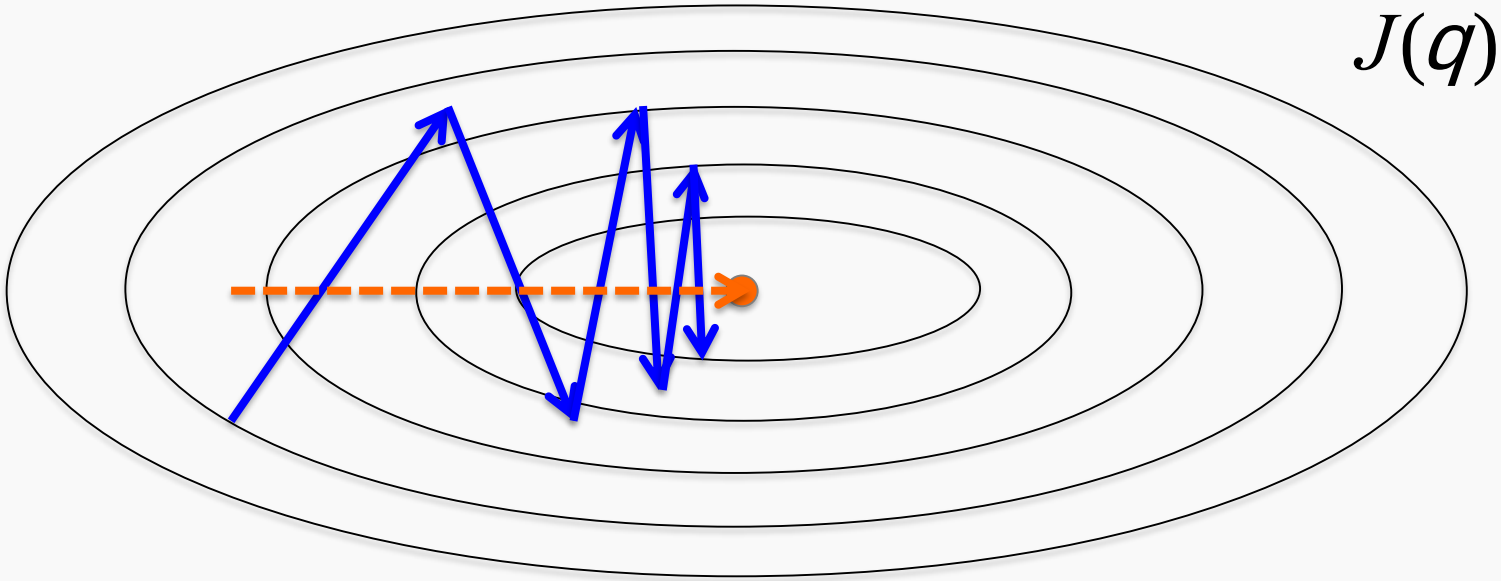
Adaptive Learning Rate

Parameter Initialization

Batch Normalization

Momentum

SGD is slow when there is high curvature



Average gradient presents faster path to opt:

– vertical components cancel out

Momentum

Uses past gradients for update

Maintains a new quantity: 'velocity'

Exponentially decaying average of gradients:

Current gradient update

$$v = av + (-eg)$$

$a \hat{I} [0,1)$ controls how quickly effect of past gradients decay

Momentum

Compute gradient estimate:

$$g = \frac{1}{m} \sum_i \nabla_q L(f(x^{(i)}; q), y^{(i)})$$

Update velocity:

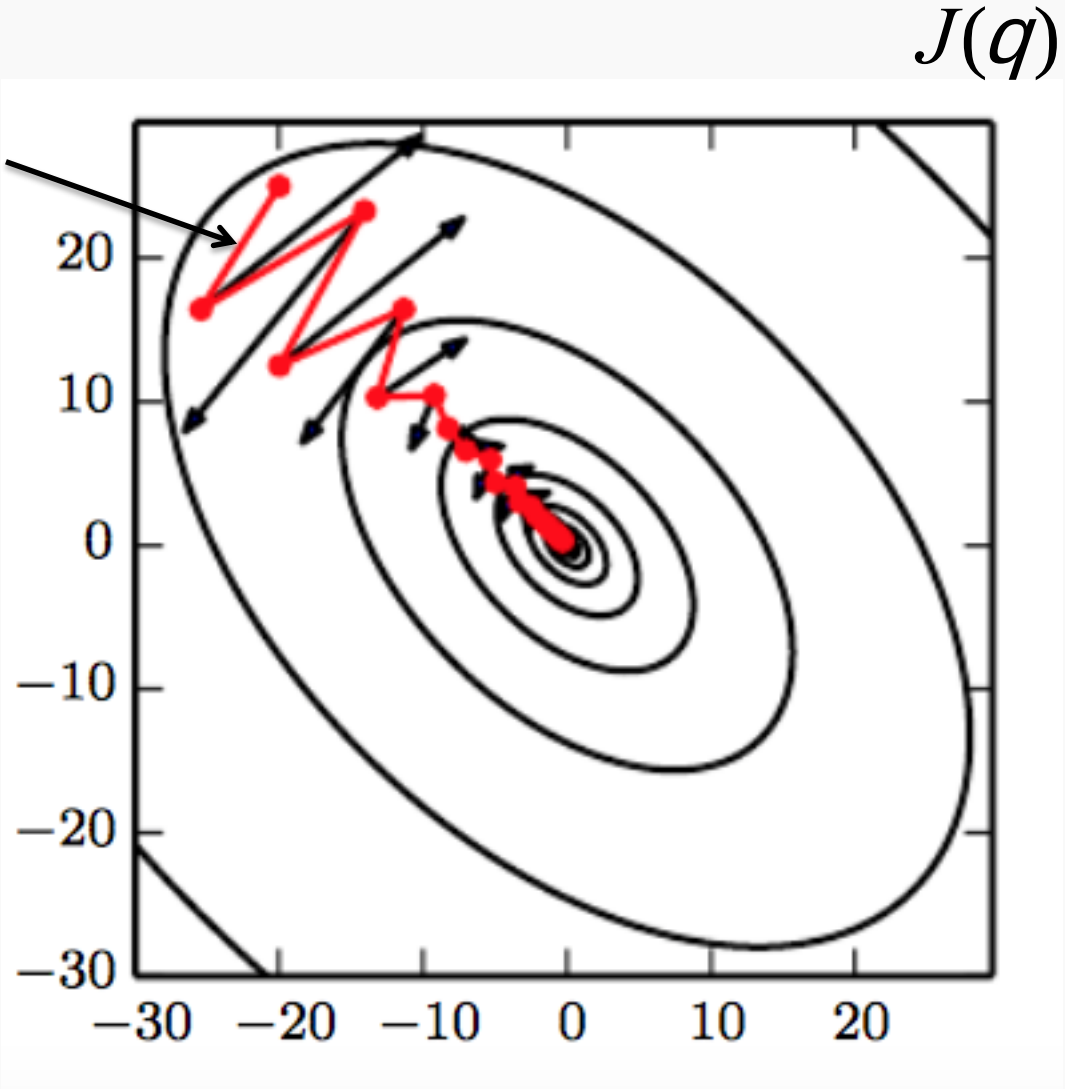
$$v = \alpha v - \epsilon g$$

Update parameters:

$$q = q + v$$

Momentum

Damped oscillations: gradients in opposite directions get cancelled out



Goodfellow et al. (2016)

Outline

Challenges in Optimization

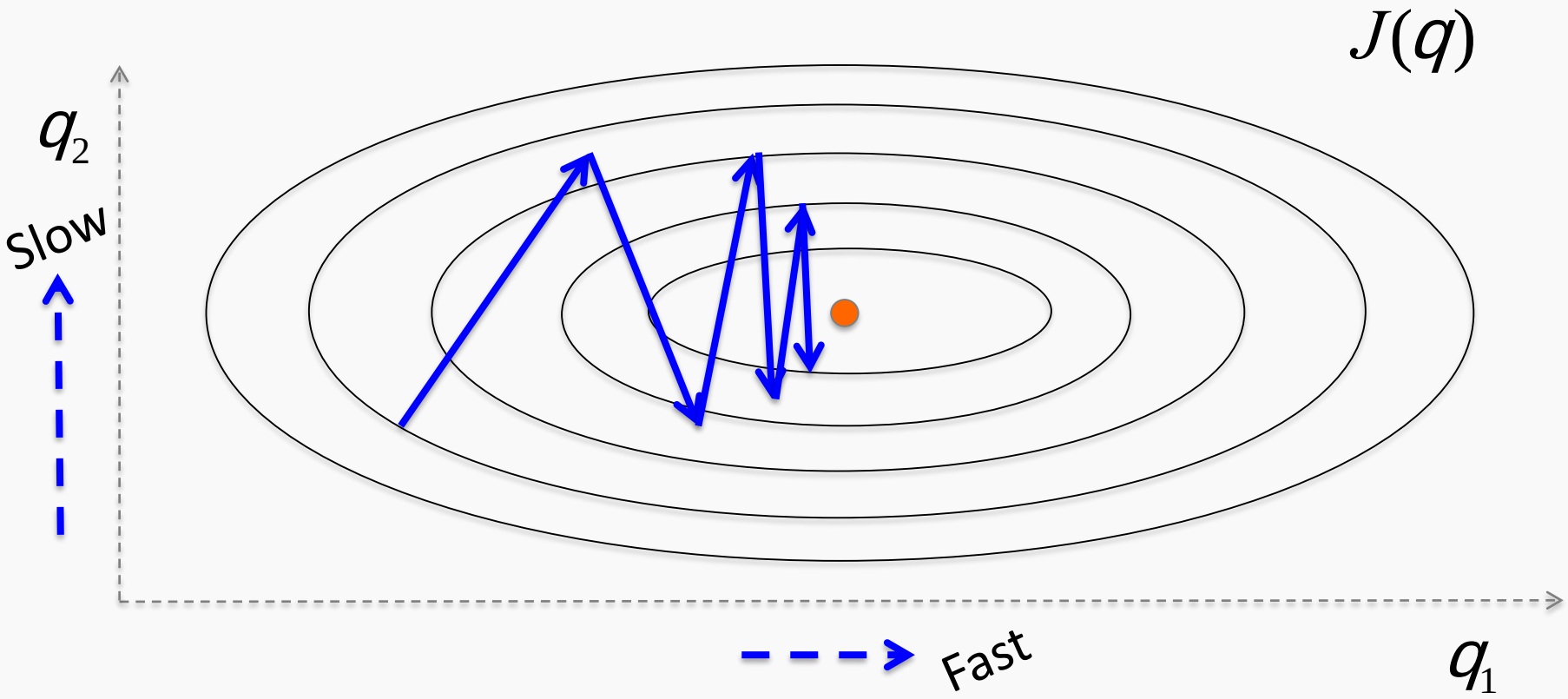
Momentum

Adaptive Learning Rate

Parameter Initialization

Batch Normalization

Adaptive Learning Rates



Oscillations along vertical direction

– Learning must be slower along parameter 2

Use a different learning rate for each parameter?

AdaGrad

- Accumulate squared gradients:

$$r_i = r_i + g_i^2$$

- Update each parameter:

$$q_i = q_i - \frac{e}{d + \sqrt{r_i}} g_i$$

Inversely proportional to cumulative squared gradient

- Greater progress along gently sloped directions

RMSProp

- For non-convex problems, AdaGrad can prematurely decrease learning rate
- Use exponentially weighted average for gradient accumulation

$$r_i = \rho r_i + (1-\rho) g_i^2$$

$$q_i = q_i - \frac{\epsilon}{\delta + \sqrt{r_i}} g_i$$

Adam

- RMSProp + Momentum
- Estimate first moment:

$$v_i = r_1 v_i + (1 - r_1) g_i$$

- Estimate second moment:

$$r_i = r_2 r_i + (1 - r_2) g_i^2$$

- Update parameters:

$$q_i = q_i - \frac{e}{d + \sqrt{r_i}} v_i$$

Also applies bias correction to v and r

Works well in practice, is fairly robust to hyper-parameters

Outline

Challenges in Optimization

Momentum

Adaptive Learning Rate

Parameter Initialization

Batch Normalization

Parameter Initialization

- Goal: break symmetry between units
  - so that each unit computes a different function
- Initialize all weights (not biases) randomly
  - Gaussian or uniform distribution
- Scale of initialization?
  - *Large* -> grad explosion, *Small* -> grad vanishing

Xavier Initialization

- Heuristic for all outputs to have unit variance
- For a fully-connected layer with m inputs:

$$W_{ij} \sim N\left(0, \frac{1}{m}\right)$$

- For ReLU units, it is recommended:

$$W_{ij} \sim N\left(0, \frac{2}{m}\right)$$

Outline

Challenges in Optimization

Momentum

Adaptive Learning Rate

Parameter Initialization

Batch Normalization

Feature Normalization

Good practice to normalize features before applying learning algorithm:

Feature vector

Vector of mean feature values

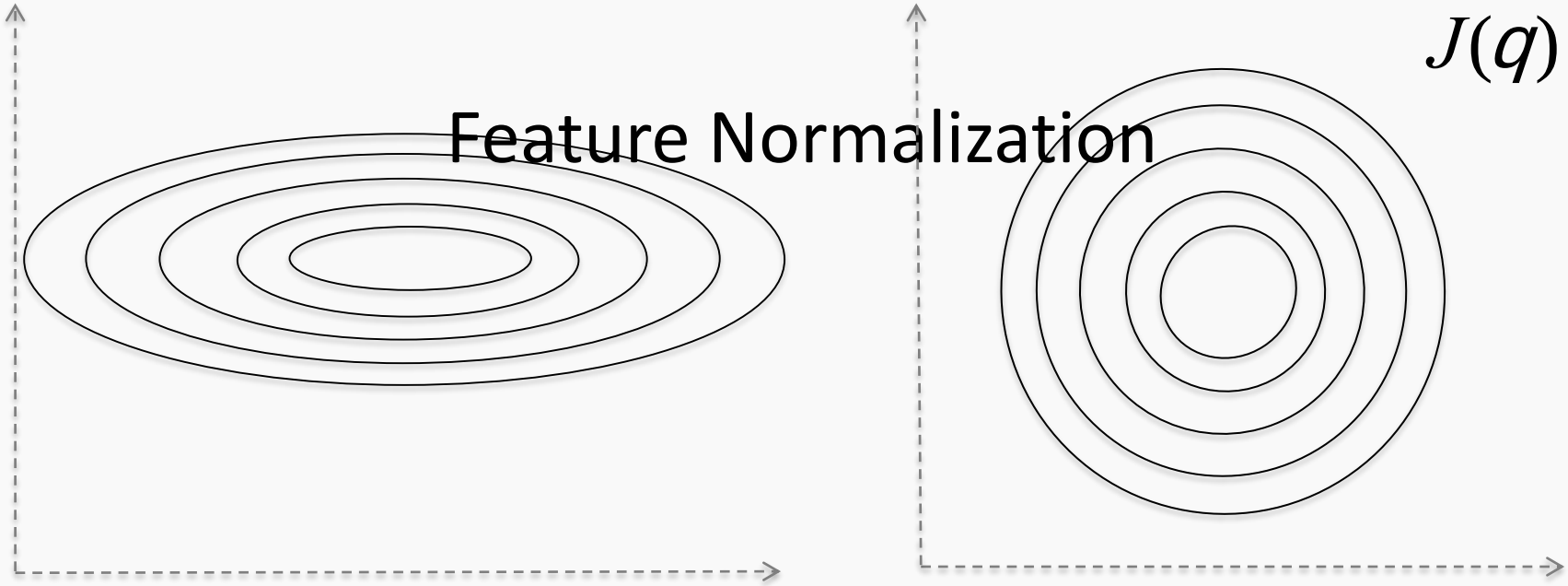
$$x' = \frac{x - m}{s}$$

Vector of SD of feature values

Features in same scale: mean 0 and variance 1

- Speeds up learning

Feature Normalization

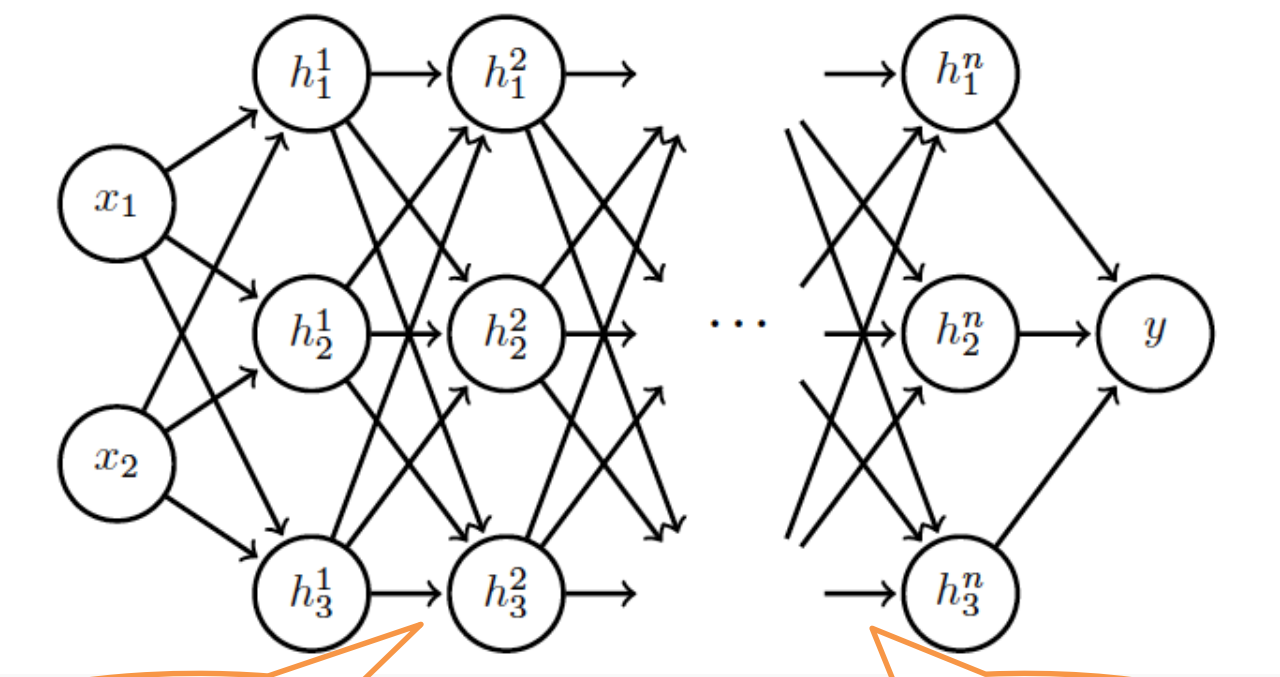


Before normalization

After normalization

Internal Covariance Shift

Each hidden layer changes distribution of inputs to next layer: *slows down learning*



Normalize inputs to layer 2

...

Normalize inputs to layer n

Batch Normalization

Training time:

- Mini-batch of activations for layer to normalize

$$H = \begin{bmatrix} H_{11} & \cdots & H_{1K} \\ \vdots & \ddots & \vdots \\ H_{N1} & \cdots & H_{NK} \end{bmatrix}$$

K hidden layer activations

N data points in mini-batch

Batch Normalization

Training time:

- Mini-batch of activations for layer to normalize

where

$$H' = \frac{H - \mu}{\sigma}$$

$$\mu = \frac{1}{m} \sum_i H_{i,:} \qquad \sigma = \sqrt{\frac{1}{m} \sum_i (H - \mu)_i^2 + \delta}$$

Vector of mean activations across mini-batch

Vector of SD of each unit across mini-batch

Batch Normalization

Training time:

- Normalization can reduce expressive power
- Instead use:

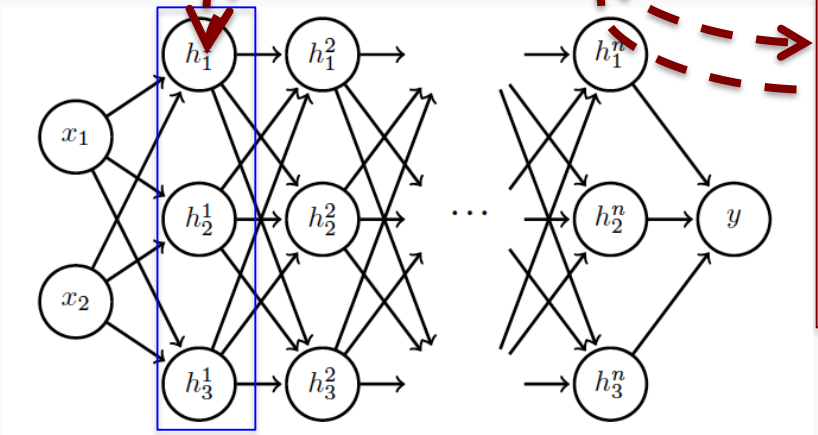
$$gH + b$$

Learnable parameters

- Allows network to control range of normalization

Batch Normalization

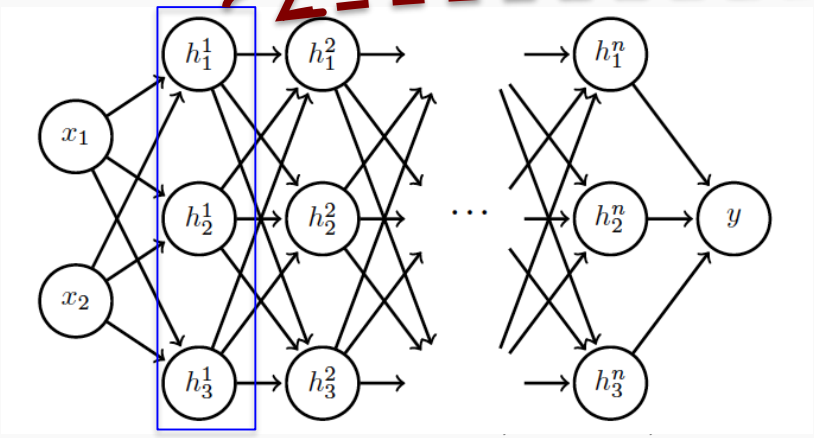
Batch 1



$$\mu^1 = \frac{1}{m}\sum_i H_{i,:}$$

$$\sigma^1 = \sqrt{\frac{1}{m}\sum_i (H - \mu)_i^2 + \delta}$$

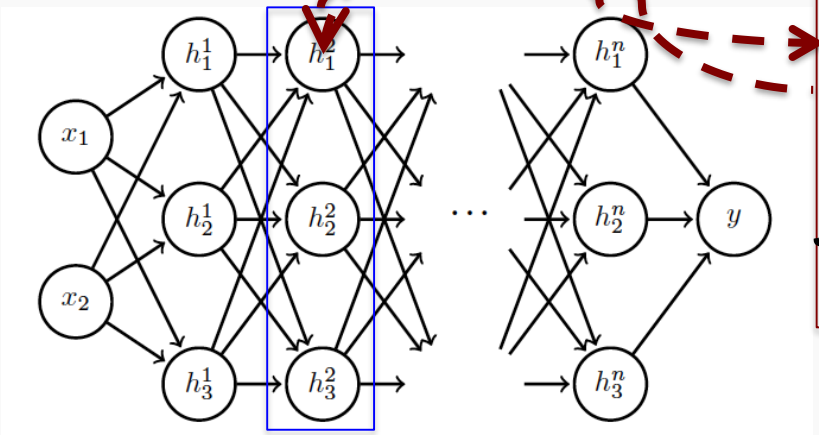
Batch *N*



Add normalization operations for layer 1

Batch Normalization

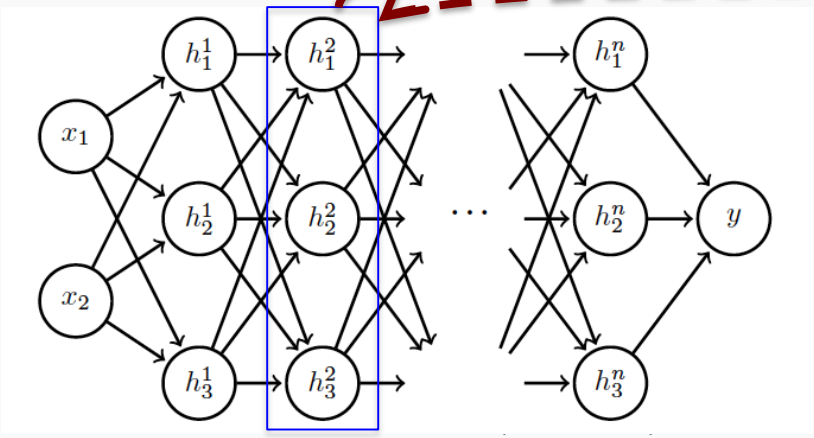
Batch 1



$$\mu = \frac{1}{m}\sum_i H_{i,:}$$

$$\sigma = \sqrt{\frac{1}{m}\sum_i (H - \mu)_i^2 + \delta}$$

Batch *N*



Add normalization operations for layer 2 and so on …

Batch Normalization

Differentiate the joint loss for N mini-batches

Back-propagate *through* the norm operations

Test time:

- Model needs to be evaluated on a *single example*
- Replace μ and σ with running averages collected during training