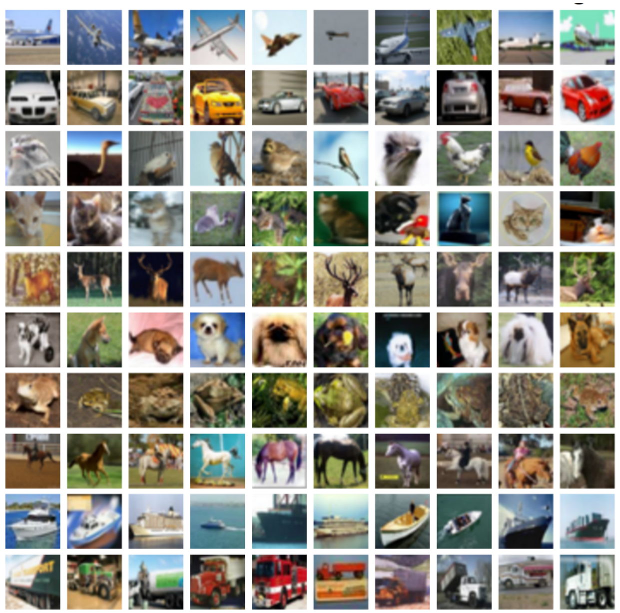
Feedforward Neural Networks

Jian Tang
HEC Montreal
Mila-Quebec AI Institute
Email: jian.tang@hec.ca

The task

- The goal is to learn a mapping function $y = f(x; \theta)$ (e.g., for classification $f: R^d \rightarrow C$).



airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Example: image classification

Traditional Machine Learning

Hand-crafted
Feature Extractor

Simple Trainable Classifier
e.g., SVM, LR

Domain experts

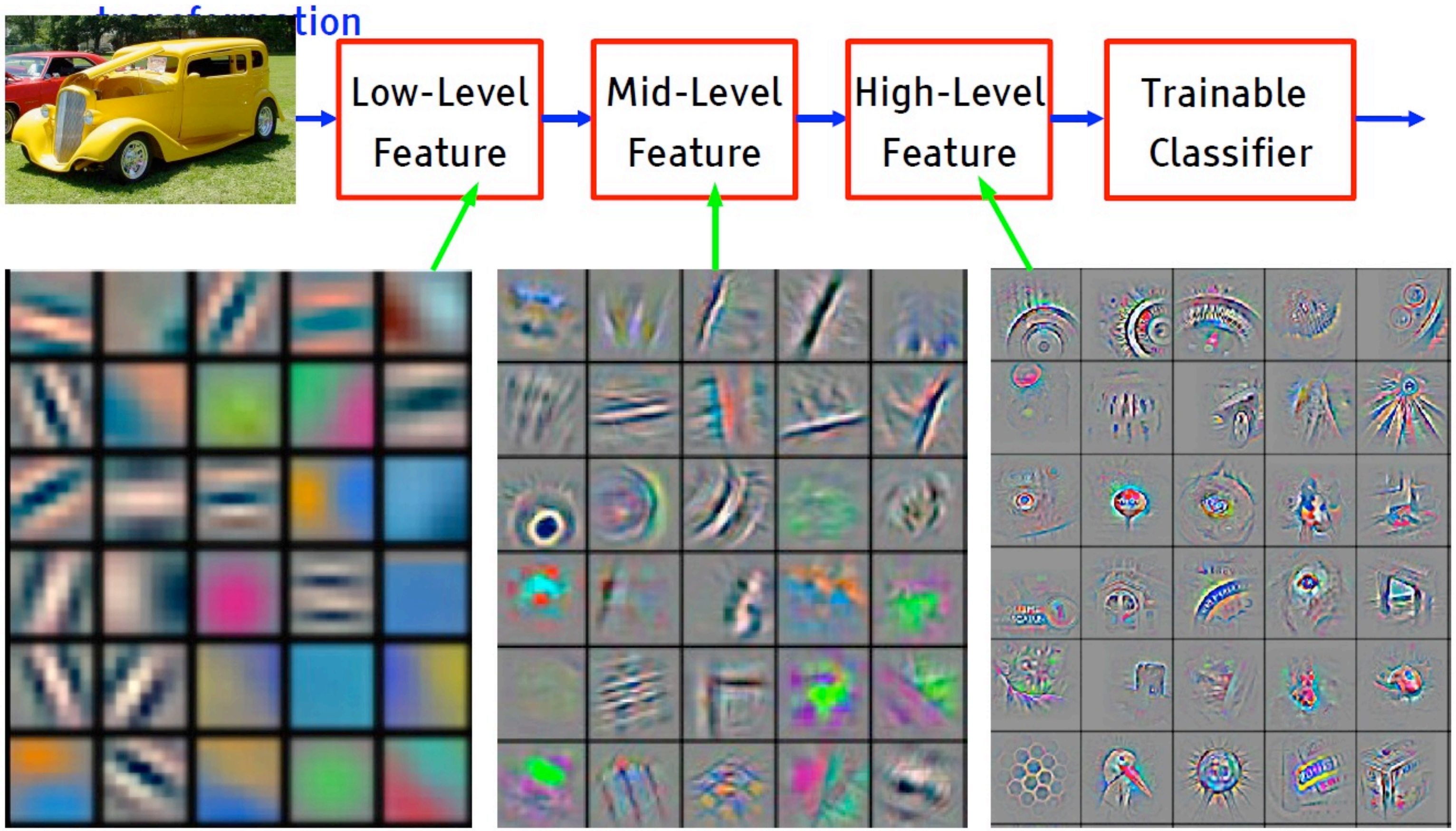
Deep Learning= End-to-end Learning/Feature Learning

Trainable
Feature Extractor

Trainable Classifier
e.g., SVM, LR

Domain expert

Deep Learning=
Learning Hierarchical representations



Low-Level Feature

Mid-Level Feature

High-Level Feature

Trainable Classifier

(Figure from LeCun)

Hierarchical representations with increasing level of abstraction

- Image recognition
  - Pixel -> edge -> texton-> motif -> part -> object
- Speech
  - Sample -> spectral band -> sound -> phone -> word…
- Text
  - Character -> word -> phrase->clause-> sentence ->paragraph-> document

(slides from LeCun)

Outline

- Network Components
    - Neurons (Hidden Units)
    - Output units
    - Cost functions
- Architecture design
    - Capacity of neural networks
- Training
    - Backpropagation with stochastic gradient descent

Neuron: Nonlinear Functions

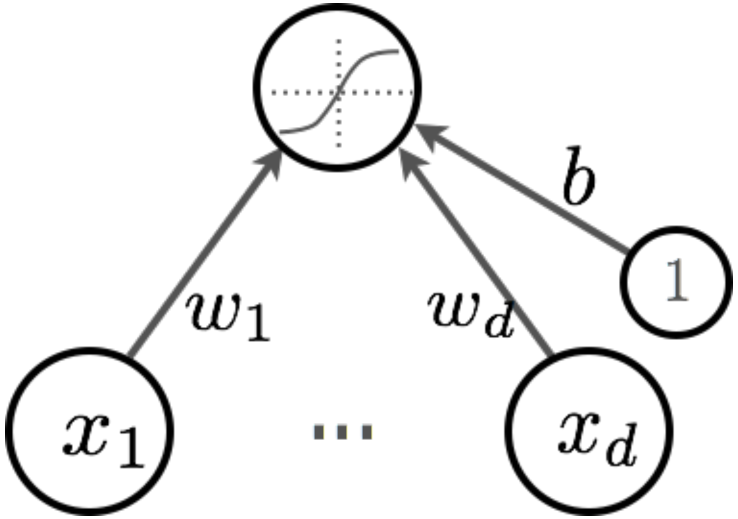
- Input: linear combination:

$$a(\boldsymbol{x}) = b + \sum_i w_i x_i = \boldsymbol{w}^T \boldsymbol{x} + b$$

- Output: nonlinear transformation:

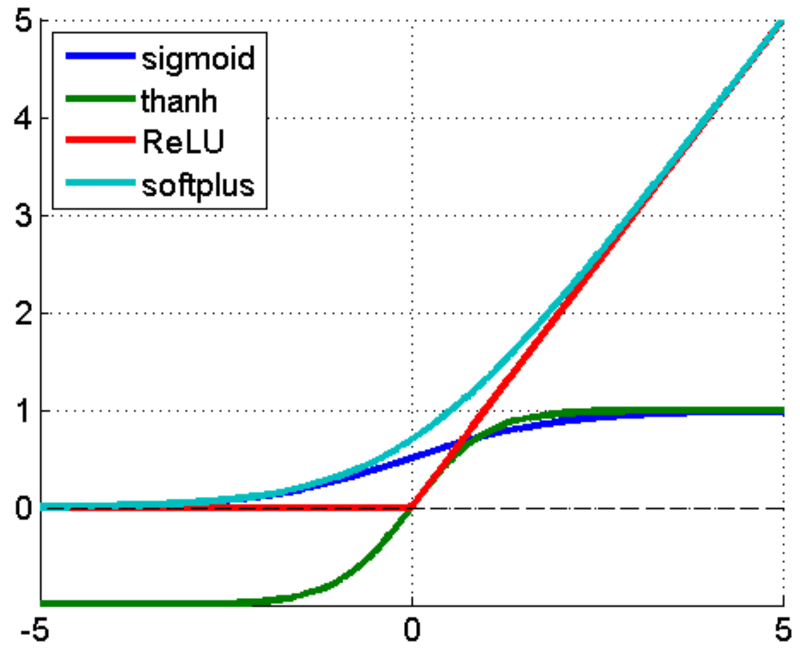
$$h(\boldsymbol{x}) = g(a(\boldsymbol{x})) = g(\boldsymbol{w}^T \boldsymbol{x} + b)$$

- **w**: are the weights (parameters)
- b is the bias term
- g(.) is called the activation function



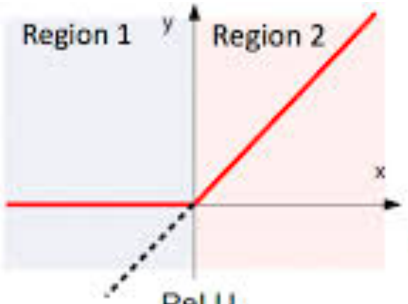
Activation functions/Hidden Units

- Sigmoid function
  - g(x) = 1/(1+exp(-x))
  - Map the input to (0,1)
- Tanh function
  - g(x) = (1-exp(-2x))/(1+exp(-2x))
  - Map the input to (-1,1)
- Rectified linear (ReLU) function
  - g(x) =max(0,x)
  - No upper bounded



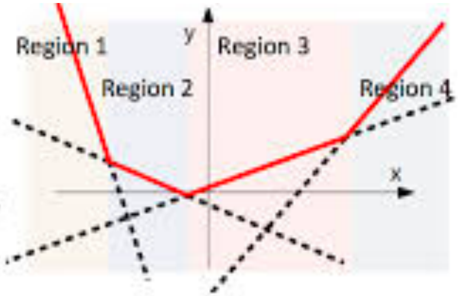
Other activation functions

- Leaky ReLU (Maas et al. 2013)
    - $g(x) = \max(0, x) + \alpha \min(0, x)$
    - Fix α to a small value, e.g., 0.01
- Parametric ReLU (He et al. 2015)
    - Treat α as a parameter to learn
- Maxout units (Goodfellow et al. ,2013)
    - Generalize rectified linear units
    - Divide the output units into groups of k values, and output the maximum value in each group
    - Provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.



ReLU

LReLU and PReLU



Maxout (k=4)

One Hidden layer Neural Networks

- Input of the hidden layer:

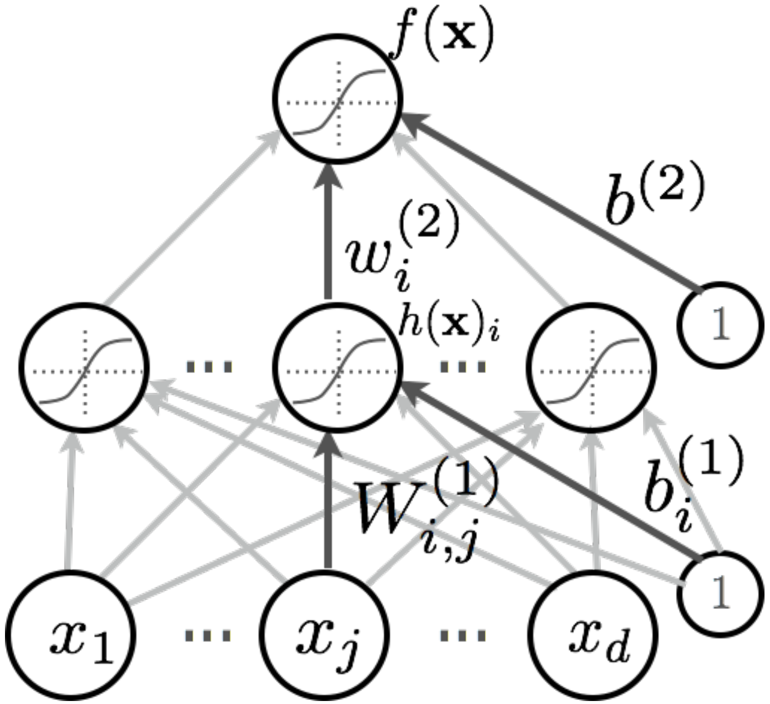
$$a(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$$

- Nonlinear transformation:

$$h(\boldsymbol{x}) = g_1(a(\boldsymbol{x}))$$

- Output layer

$$\mathrm{f}(\boldsymbol{x}) = o\ (h(\boldsymbol{x}))$$



Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost functions
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Linear Units for Gaussian Output Distributions

- Given the hidden units **h**, a layer of linear output units produces $\hat{\boldsymbol{y}} = \boldsymbol{W}^T\boldsymbol{h} + \boldsymbol{b}$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution

$$p(\boldsymbol{y}|\boldsymbol{x}) = \mathrm{N}(\boldsymbol{y}|\hat{\boldsymbol{y}}, \boldsymbol{I})$$

Sigmoid Units for Bernoulli Output Distributions

- Bernoulli output distributions: binary classification
- The goal is to define $p(y = 1|\boldsymbol{x})$, which can be defined as follows:

$$p(y = 1|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

Softmax Units for Multinomial Output Distributions

- Multinomial output distributions: multi-class classification
- First, define a linear layer to predict the unnormalized log probabilities of softmax:

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$$

- where $z_i = \log p(y = i|\boldsymbol{x})$. Formally, the softmax function is given by
-

$$p(y = i|\boldsymbol{x}) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Multilayer Neural Networks

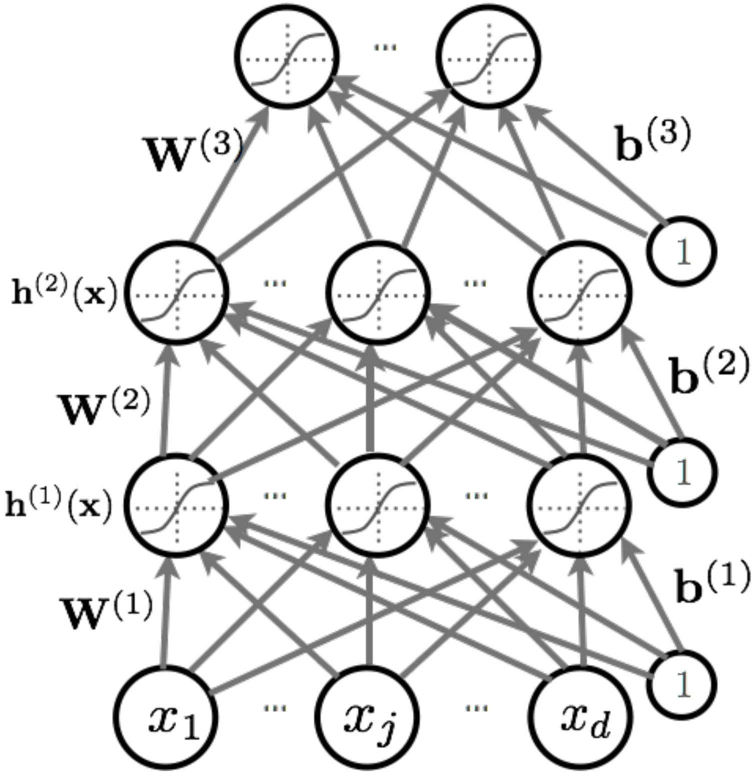
- Neural network with multiple hidden layers
- The output of previous layer as the input of next layer: (k=1…, L)

$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

- Final output layer

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



$(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

Outline

- Network Components
    - Neurons (Hidden Units)
    - Output units
    - Cost function
- Architecture design
    - Capacity of neural networks
- Training
    - Backpropagation with stochastic gradient descent

Maximum Likelihood

- Most of the time, neural networks are used to define a distribution $\mathrm{p}(y^t|\boldsymbol{x}^t;\boldsymbol{\theta})$. Therefore, the overall objective is defined as:

$$argmax_{\boldsymbol{\theta}} \frac{1}{T} \sum_t \log p(y^t|\boldsymbol{x}^t;\boldsymbol{\theta}) - \lambda \Omega(\boldsymbol{\theta})$$

- Or equivalently we can minimize the **cross-entropy error.**

Outline

- Network Components
    - Neurons (Hidden Units)
    - Output units
    - Cost functions
- Architecture design
    - Capacity of neural networks
- Training
    - Backpropagation with stochastic gradient descent

Universal Approximation

- Universal Approximation Theorem (Hornik, 1991)
  - "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units"
- However, we may not be able to find the right parameters ….
  - The layer may be infeasibly large
  - Optimizing neural networks is difficult …

Deeper Networks are Preferred

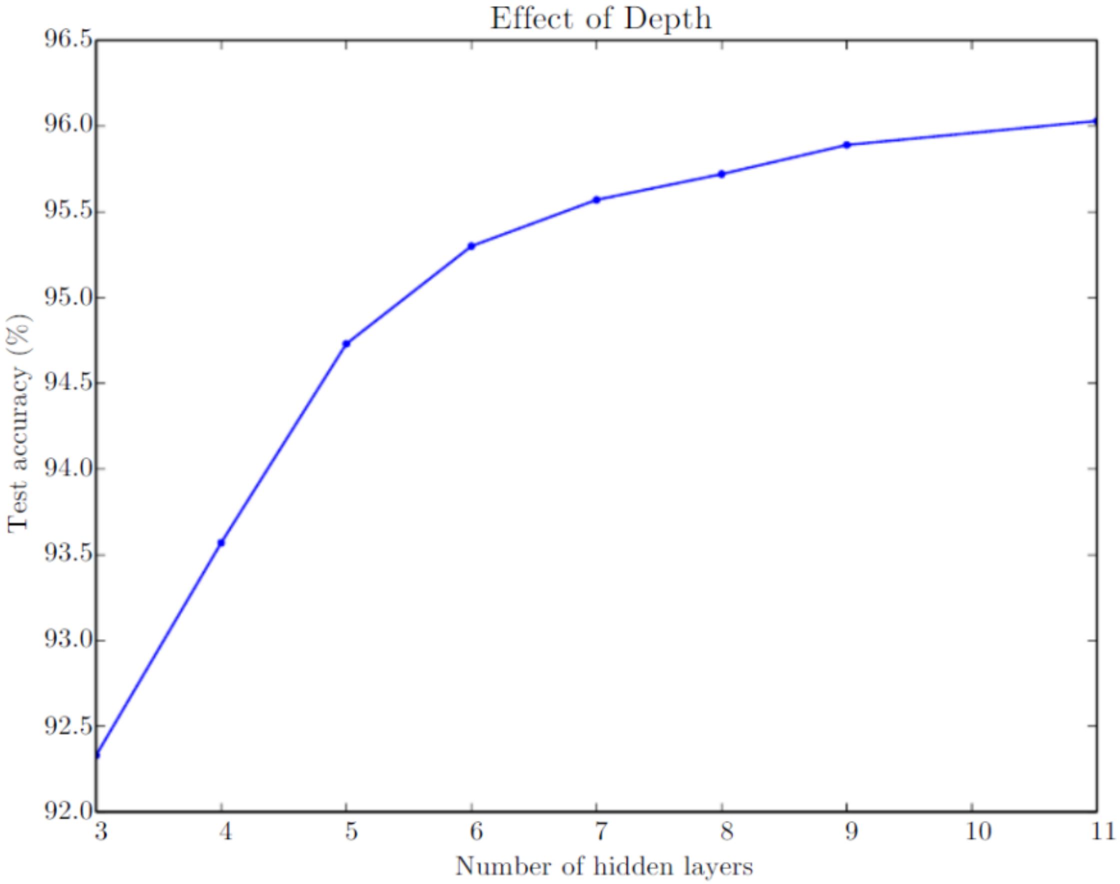


Figure: Empirical results showing that deeper networks generalize better

Deeper Networks are Preferred

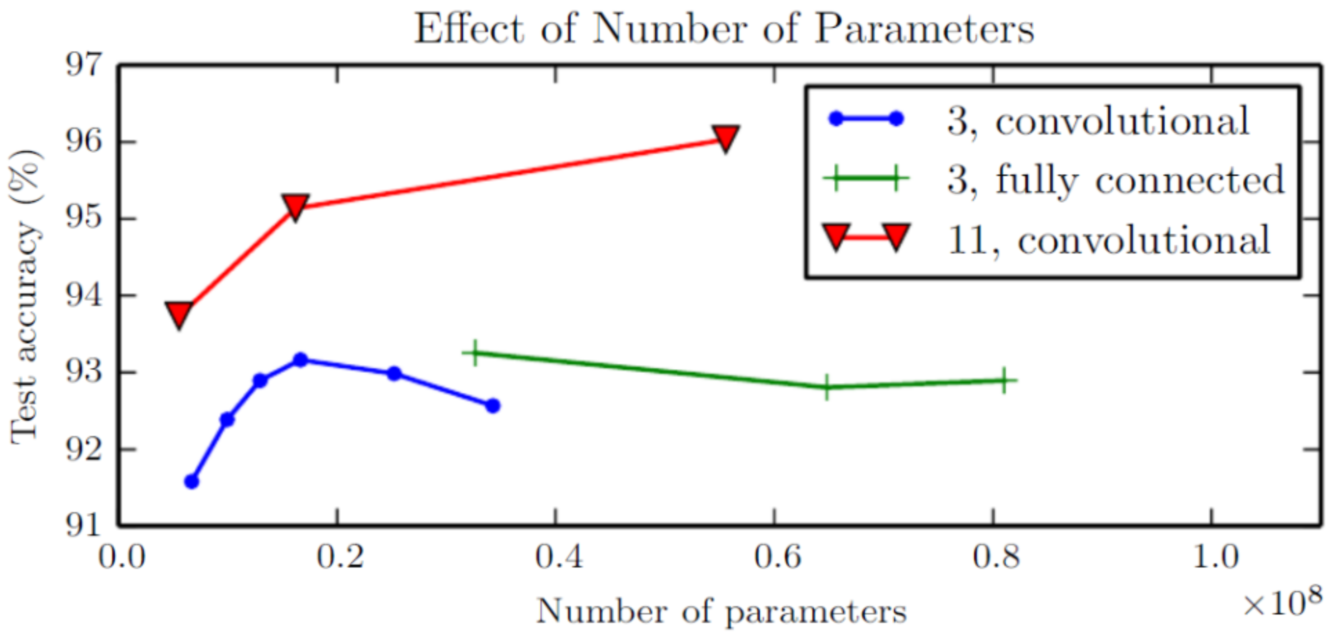


Figure: Deeper models tend to perform better with the same number of parameter

Deeper Networks are Preferred

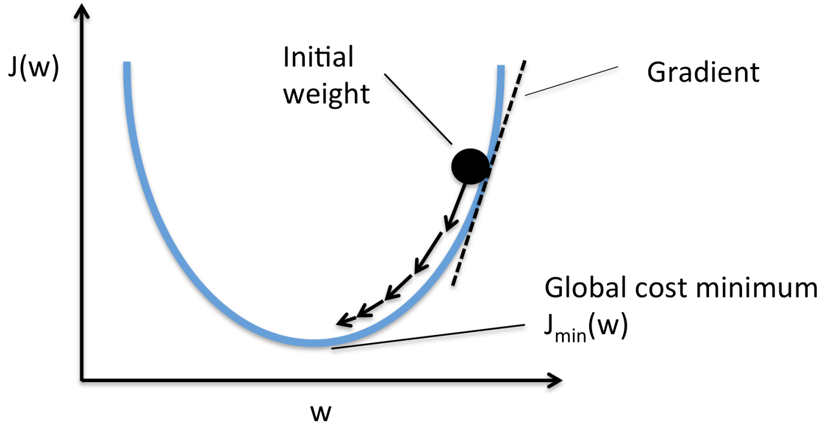
- There exist families of functions which can be approximated efficiently with deep networks but require a much larger model for shallow networks
- Statistical reasons
  - a deep model encodes a very general belief that the function we want to learn should involve composition of several simple functions
  - Or we believe the learning problem consists of discovering different levels of variations, with the high-level ones defined on the low-level (simple) ones (e.g., Pixel -> edge -> texton-> motif -> part -> object).

Outline

- Network Components
  - Neurons (Hidden Units)
  - Output units
  - Cost functions
- Architecture design
  - Capacity of neural networks
- Training
  - Backpropagation with stochastic gradient descent

Backpropagation with Stochastic Gradient Descent

- Gradient descent:
  - Update the parameters in the direction of gradients
  - Need to iterate over all the examples for every update
- Stochastic gradient descent
  - Perform updates after seeing each example



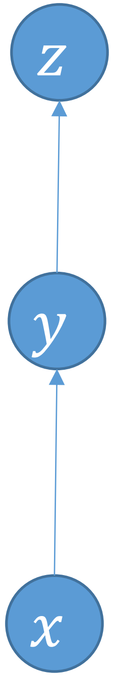
- Initialize: $\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$
- For t=1:T
  - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$

$$\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\, \Delta$$

Training epoch
=
Iteration of all examples

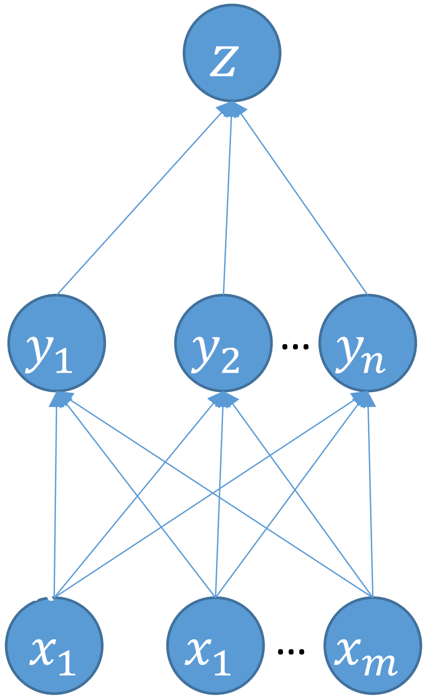
BackPropagation: Simple Chain Rule



$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

$$y = g(x)$$
$$z = f(y) = f(g(x))$$

BackPropagation: Simple Chain Rule



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

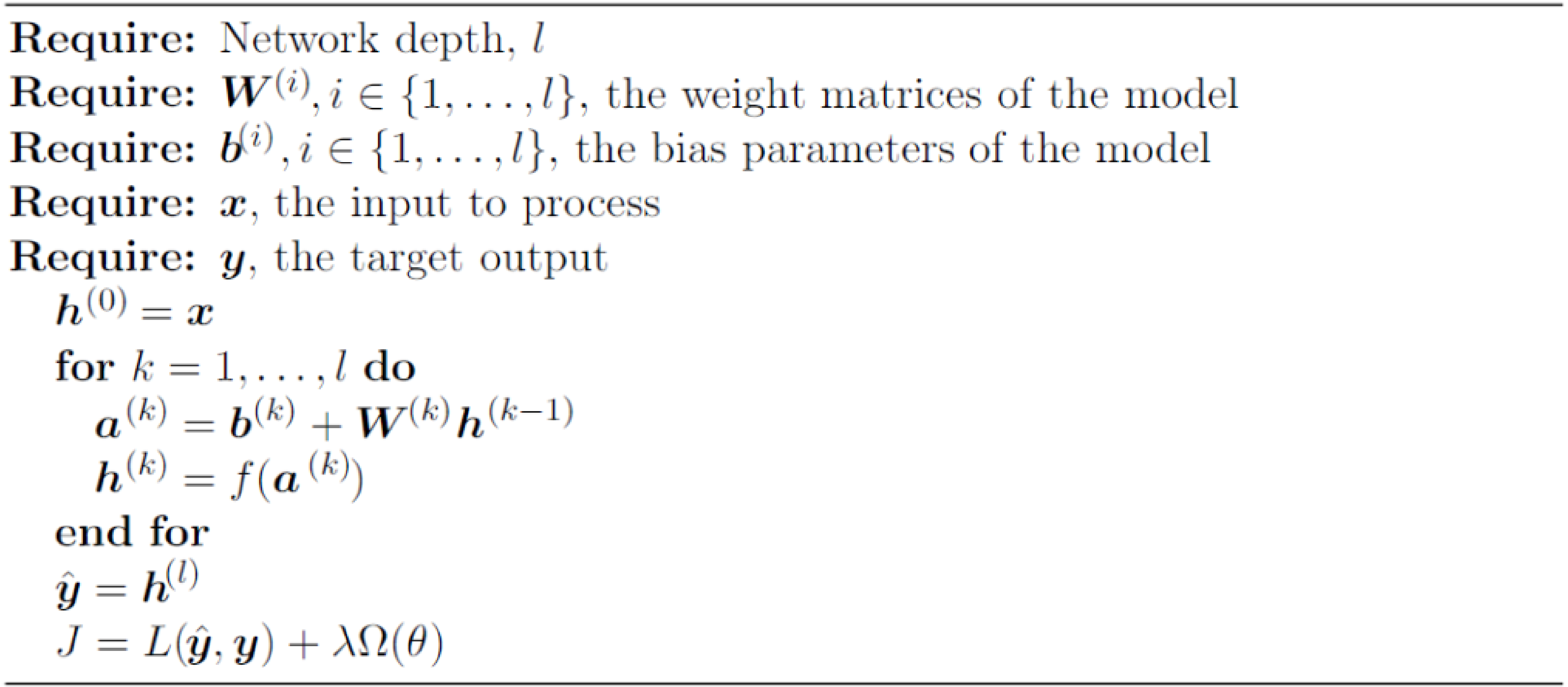
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

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

$\dfrac{\partial \vec{y}}{\partial \vec{x}}$ is the n x m Jacobian matrix of g

Forward Propagation

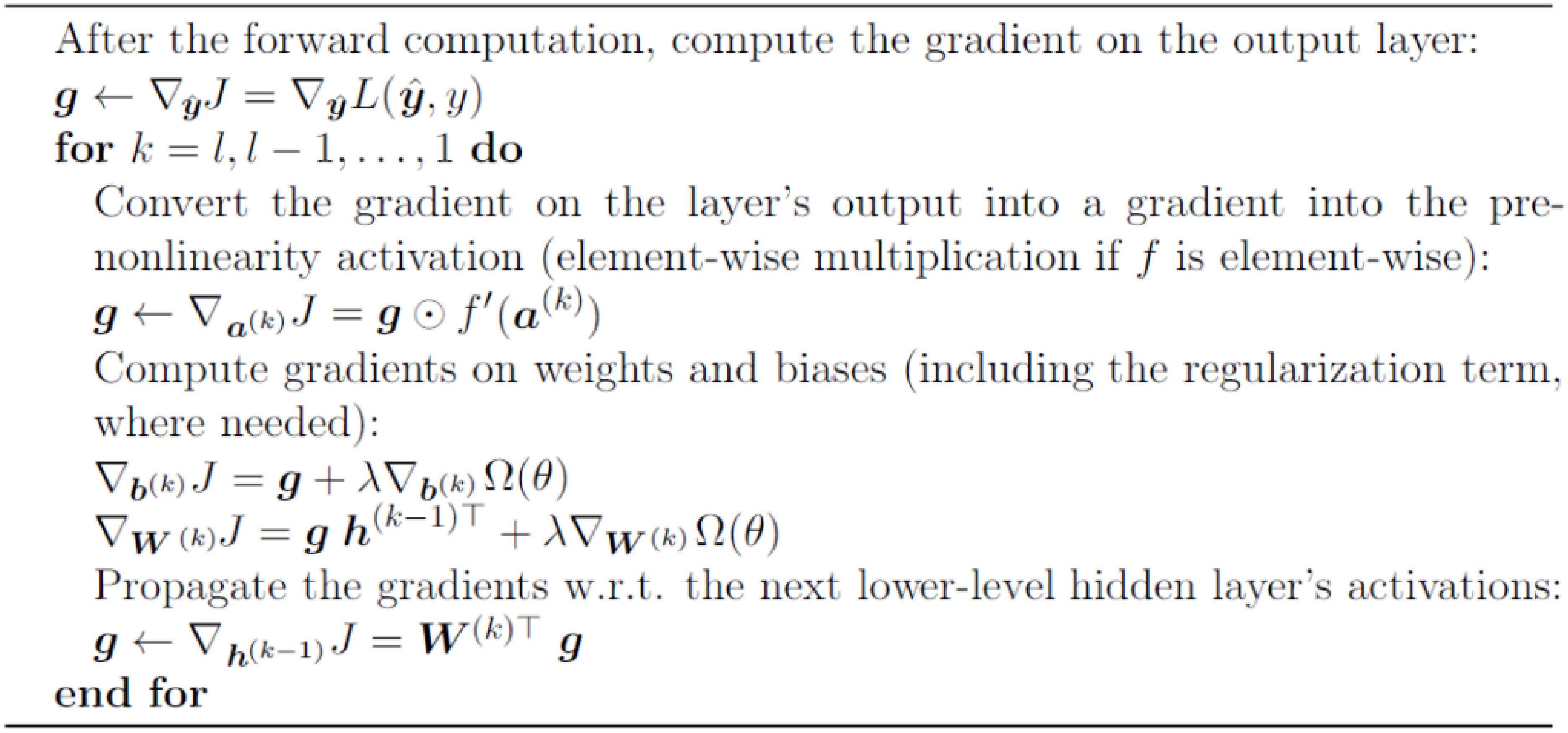
- For each training example (x, y), calculate the output based on current neural networks $\hat{y}$ and the supervised loss $\text{loss}(y, \hat{y})$



```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$
```

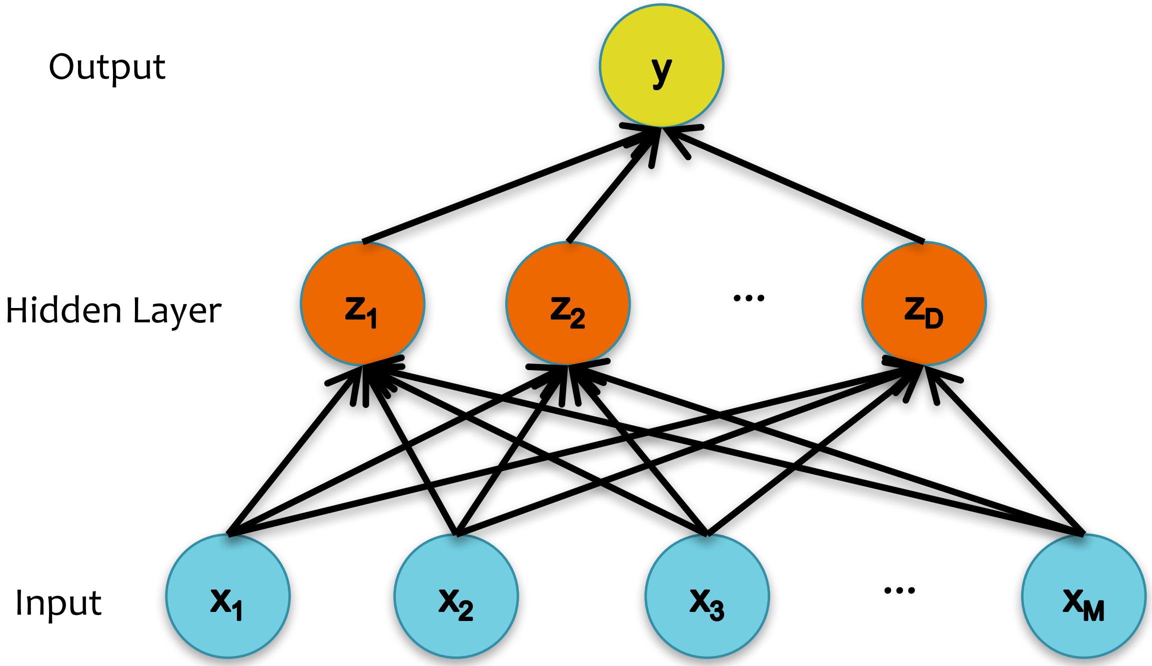
Backward Propagation

- Calculate the gradients w.r.t. the parameters in each layer
  - Backward the errors in the output to the parameter in each layer



```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ do
    Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
    Compute gradients on weights and biases (including the regularization term, where needed):
    $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
    $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
    Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
    $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for
```

Exercise



$$z_i = \sigma\left(\sum_{j=1}^{M} w_{ij}^{1} x_j\right)$$

$$o_i = \sum_{j=1}^{D} w_{ij}^{2} z_j$$

$$p(y = k) = \frac{\exp(o_k)}{\sum_{i=1}^{K} \exp(o_i)}$$

Regularization and Optimization

What is regularization

- The goal of machine learning algorithm is to perform well on the training data and generalize well to new data
- Regularization are the techniques to improve the generalization ability
  - i.e., avoid overfitting

Outline

- Regularization
    - Parameter Norm Penalties
    - Data set Augmentation
    - Noise Robustness
    - Semi-supervised Learning
    - Multi-task Learning
    - Early Stopping
    - Dropout

Parameter Norm Penalties

- Adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. The regularized objective function is denoted as:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

- $\alpha \in [0, \infty)$ is a hyperparameter that controls the weights of the regularization term
- For regularization neural networks
  - Only the weights of the linear transformation at each layer are regularized
  - The biases are not regularized (requires less data than the weights to fit accurately)

L^2 Parameter Regularization

- $\Omega(\theta) = \frac{1}{2}||\mathbf{w}||^2$, also know as weight decay or ridge regression
- The objective function:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- Update **w** with SGD:

$$\boldsymbol{w} = (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon\nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- Push **w** towards zero

L^1 Parameter Regularization

- $\Omega(\theta) = ||\mathbf{w}||_1 = \sum_i w_i,$
- The objective function:

$$\tilde{J}(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y}) = \alpha ||\mathbf{w}||_1 + J(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y})$$

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y}) = \alpha \operatorname{sign}(\mathbf{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \mathbf{X}, \boldsymbol{y})$$

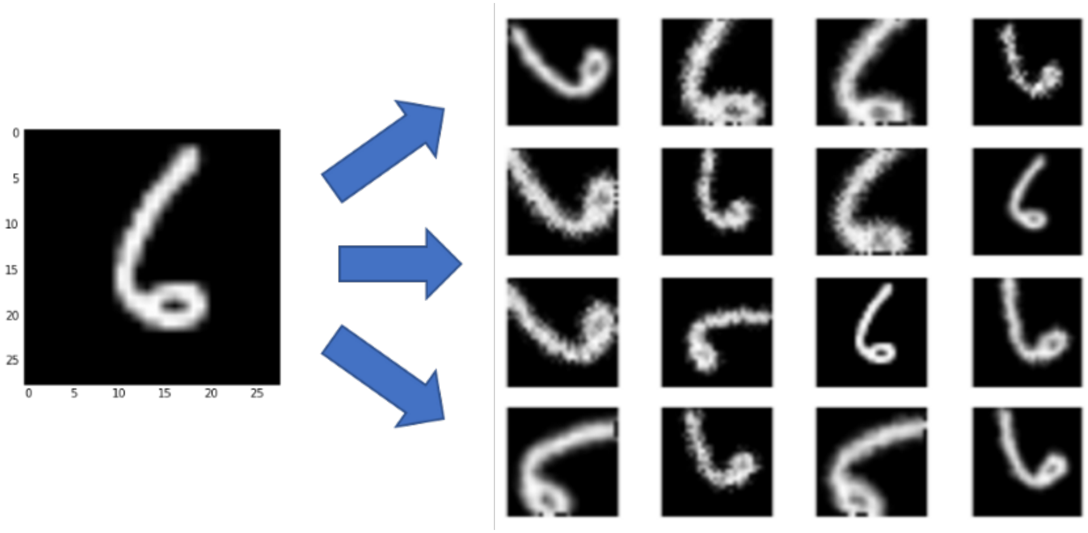
- Compare to L2 regularization, L1 regularization results in a solution that is more sparse
  - Some parameters have an optimal value of zero
  - Can be used for feature selection

Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Data Augmentation

- Best way to improve the performance of machine learning
  - Train it with more data
- Create fake data and add it to the training data
  - Translation
  - Rotation
  - Random crops
  - Inject noise in both the input and output
  - …



Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Noise Robustness

- Adding noise to the weights
  - Push the model into regions where the model is relatively insensitive to small variations in the weights
  - Find points that are not merely minima, but minima surrounded by flat regions.
- Adding noise at the output targets
  - Most data sets have some amount of mistakes in the output labels: y
  - Explicitly model the noise on the labels
  - For example, the training label y is correct with probability $1 - \epsilon$, and any of the other labels with probability ϵ

Outline

- Regularization
    - Parameter Norm Penalties
    - Dataset Augmentation
    - Noise Robustness
    - Semi-supervised Learning
    - Multi-task Learning
    - Early Stopping
    - Dropout

Semi-supervised Learning

- Semi-supervised learning: both unlabeled examples from p(x) and labeled examples p(x,y) are used to estimate p(y|x)
- Share parameters between the unsupervised objective p(x) and supervised objective p(y|x)
  - E.g., for both objectives, the goal is to learn a representation h = f(x), which can be shared across the two objectives
- A very hot topic now
  - Especially in pretraining language models in NLP.

Example:

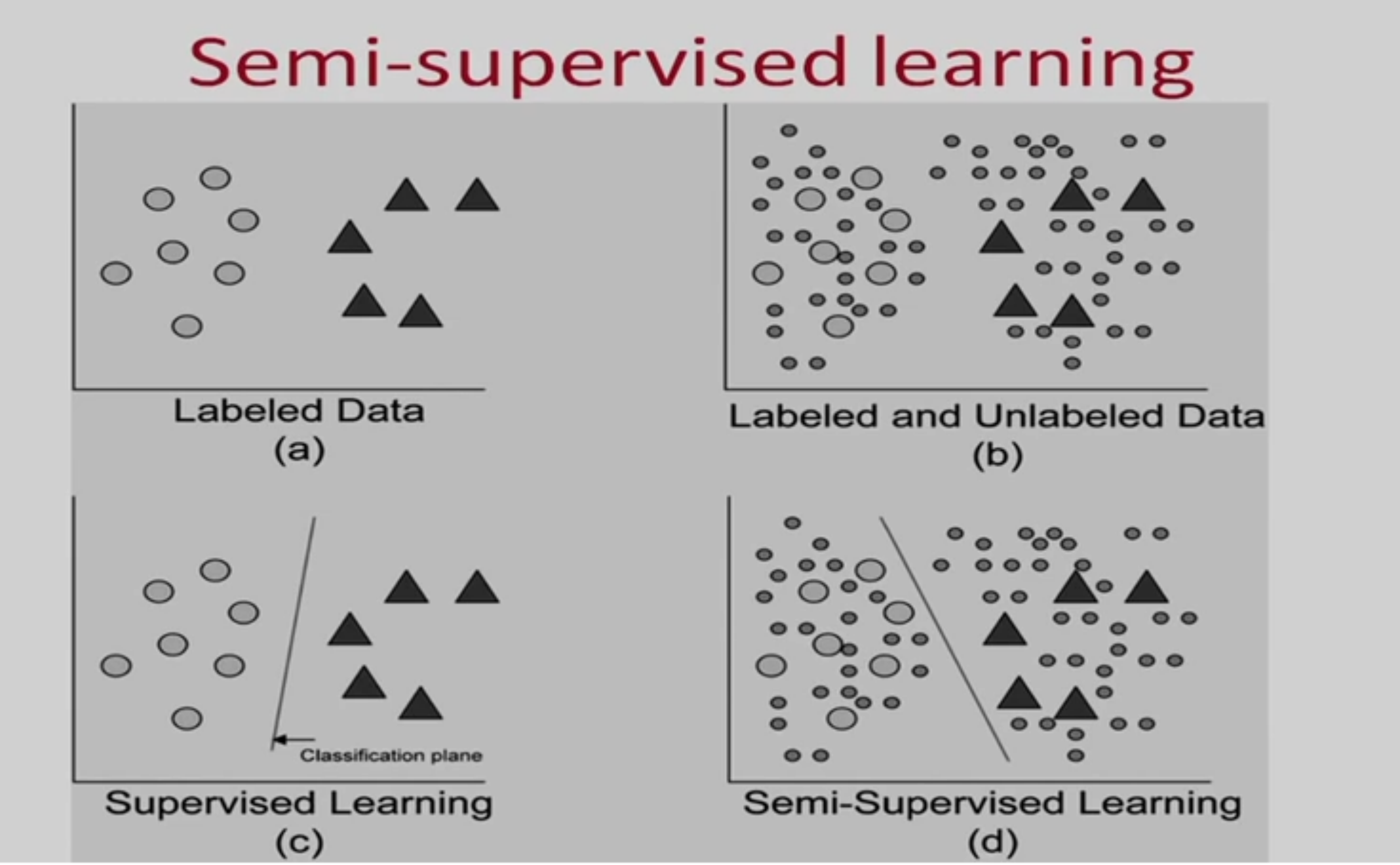


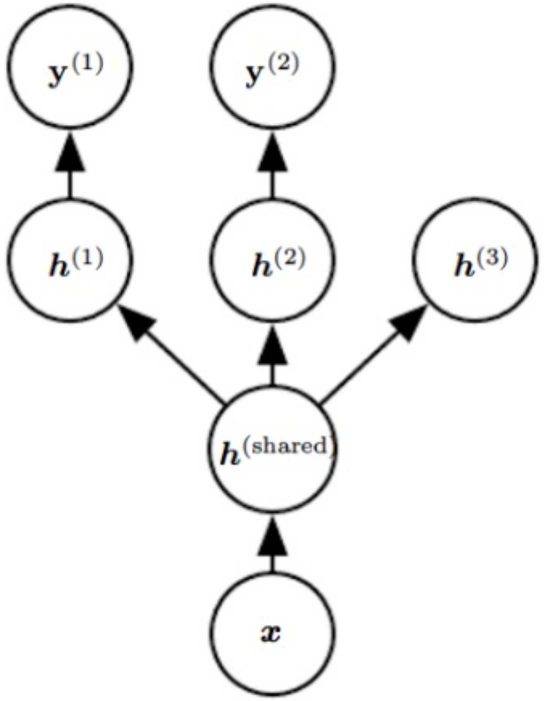
Image from Internet

Outline

- Regularization
  - Parameter Norm Penalties
  - Dataset Augmentation
  - Noise Robustness
  - Semi-supervised Learning
  - Multi-task Learning
  - Early Stopping
  - Dropout

Multi-task Learning

- Jointly learning multi-tasks by sharing the same inputs and some intermediate representations, which capture a common pool of factors
- Model
  - Task-specific parameters
  - Generic parameters shared across all the tasks

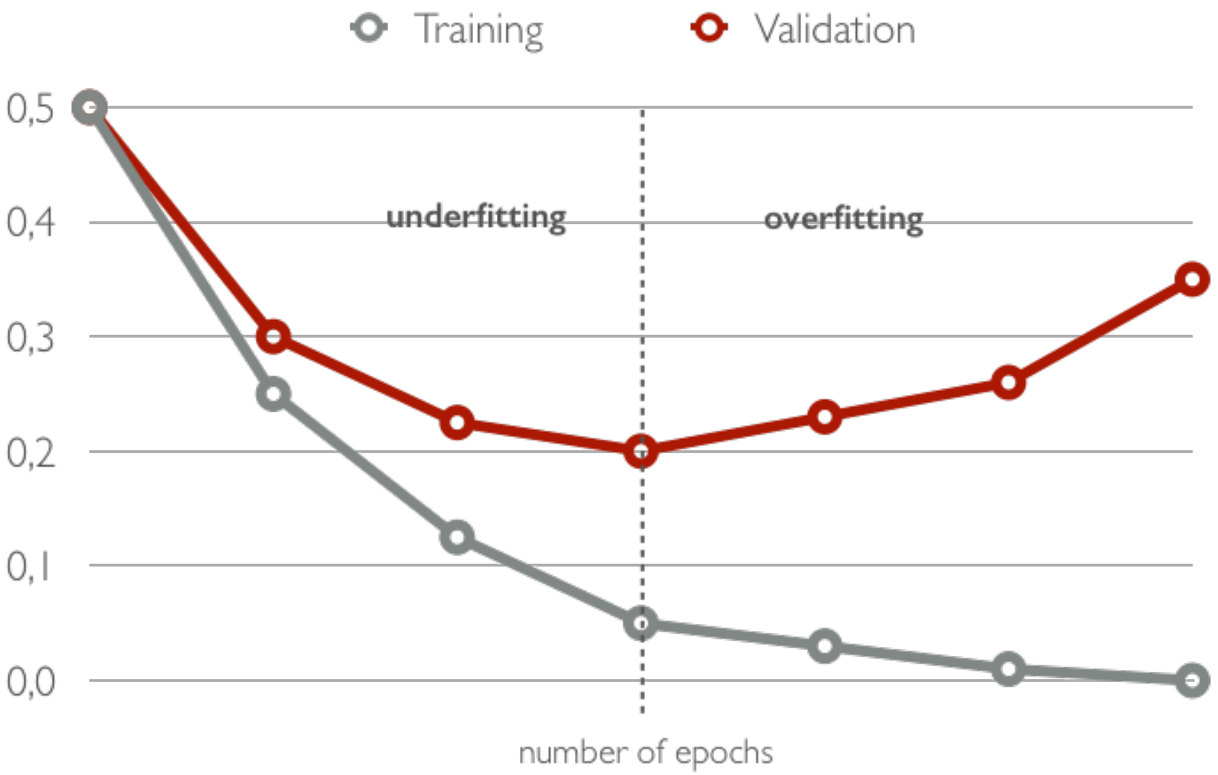


Outline

- Regularization
    - Parameter Norm Penalties
    - Dataset Augmentation
    - Noise Robustness
    - Semi-supervised Learning
    - Multi-task Learning
    - Early Stopping
    - Dropout

Early Stopping

- To select the number of epochs, stop training when validation set error increases (with some look ahead).



Outline

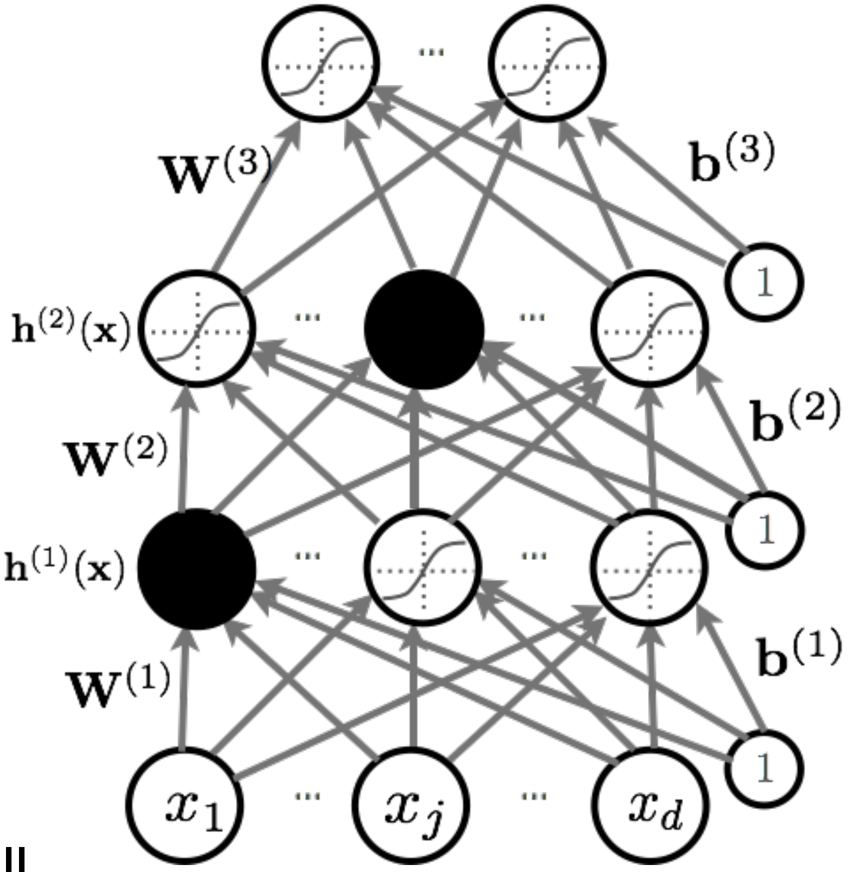
- Regularization
    - Parameter Norm Penalties
    - Dataset Augmentation
    - Noise Robustness
    - Semi-supervised Learning
    - Multi-task Learning
    - Early Stopping
    - Dropout

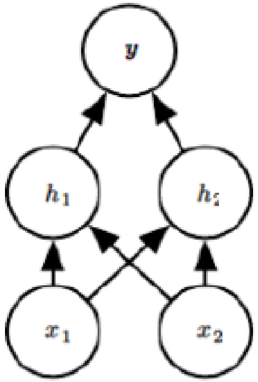
Dropout

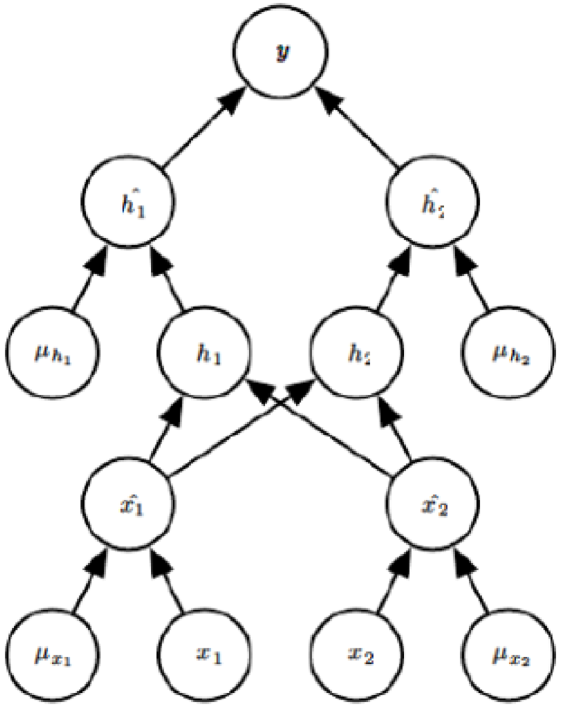
- Overcome overfitting by an ensemble of multiple different models
  - Trained with different architectures
  - Trained on different data sets
- Too expensive on deep neural networks
- Dropout:
  - Training multiple networks together by parameter sharing

Dropout

- Key idea: Cripple neural network by removing hidden units stochastically
  - each hidden unit is set to 0 with probability 0.5
  - hidden units cannot co-adapt to other units
  - hidden units must be more generally useful
- Could use a different dropout probability, but 0.5 usually works well




y
h_1
h_2
x_1
x_2


y
$\hat{h_1}$
$\hat{h_2}$
μ_{h_1}
h_1
h_2
μ_{h_2}
$\hat{x_1}$
$\hat{x_2}$
μ_{x_1}
x_1
x_2
μ_{x_2}

Dropout

- Use random binary masks m$^{(k)}$
  - layer pre-activation for k>0

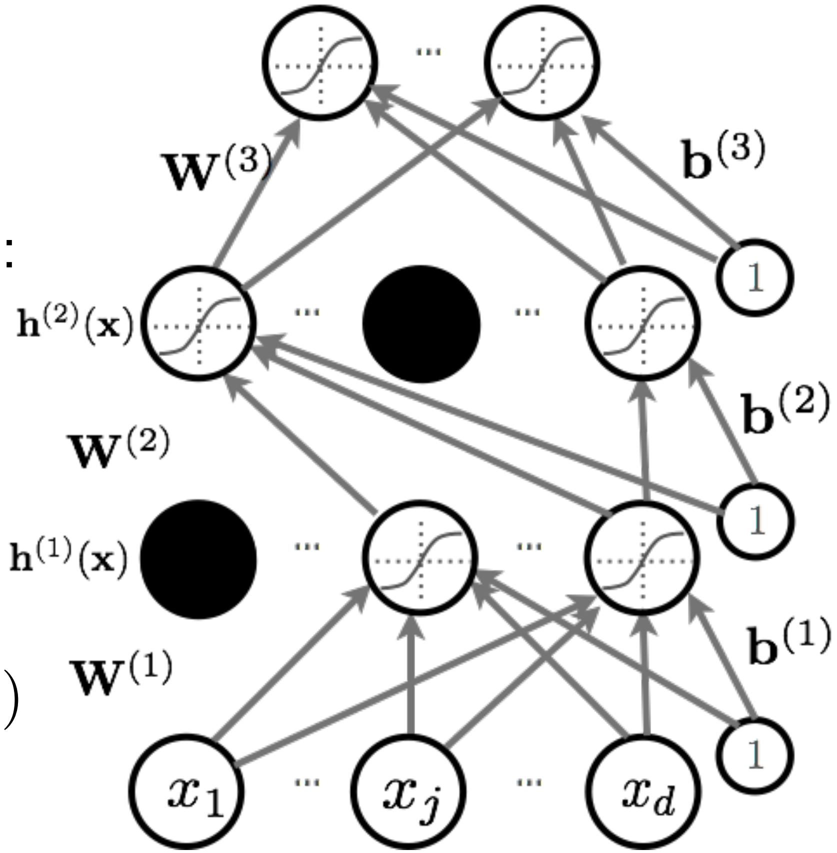
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k=1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x})) \odot \mathrm{m}^{(k)}$$

  - Output activation (k=L+1)

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



Dropout at Test Time

- At test time, we replace the masks by their expectation
  - This is simply the constant vector 0.5 if dropout probability is 0.5
  - For single hidden layer: equivalent to taking the geometric average of all neural networks, with all possible binary masks

- Can be combined with unsupervised pre-training

- Beats regular backpropagation on many datasets

- Ensemble: Can be viewed as a geometric average of exponential number of networks.

Outline

- Optimization
    - Parameter Initialization Strategies
    - Momentum
    - Adaptive Learning Rates (AdaGrad, RMSProp, Adam)
    - Batch Normalization

Parameter Initialization (Glorot and Bengio, 2010)

- For a fully connected network with *m* inputs and *n* outputs, the weights are sampled according to:

$$W_{ij} \sim U\left(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right).$$

- which aims to tradeoff between the goal of initializing all layers to have the same **activation variance** and the goal of initializing all layers to have the same **gradient variance**

Tricks of the Trade

- Normalizing your (real-valued) data:
  - for each dimension x_i subtract its training set mean
  - divide each dimension x_i by its training set standard deviation
  - this can speed up training

- Decreasing the learning rate: As we get closer to the optimum, take smaller update steps:

  i. start with large learning rate (e.g. 0.1)
  ii. maintain until validation error stops improving
  iii. divide learning rate by 2 and go back to (ii)

Mini-batch, Momentum

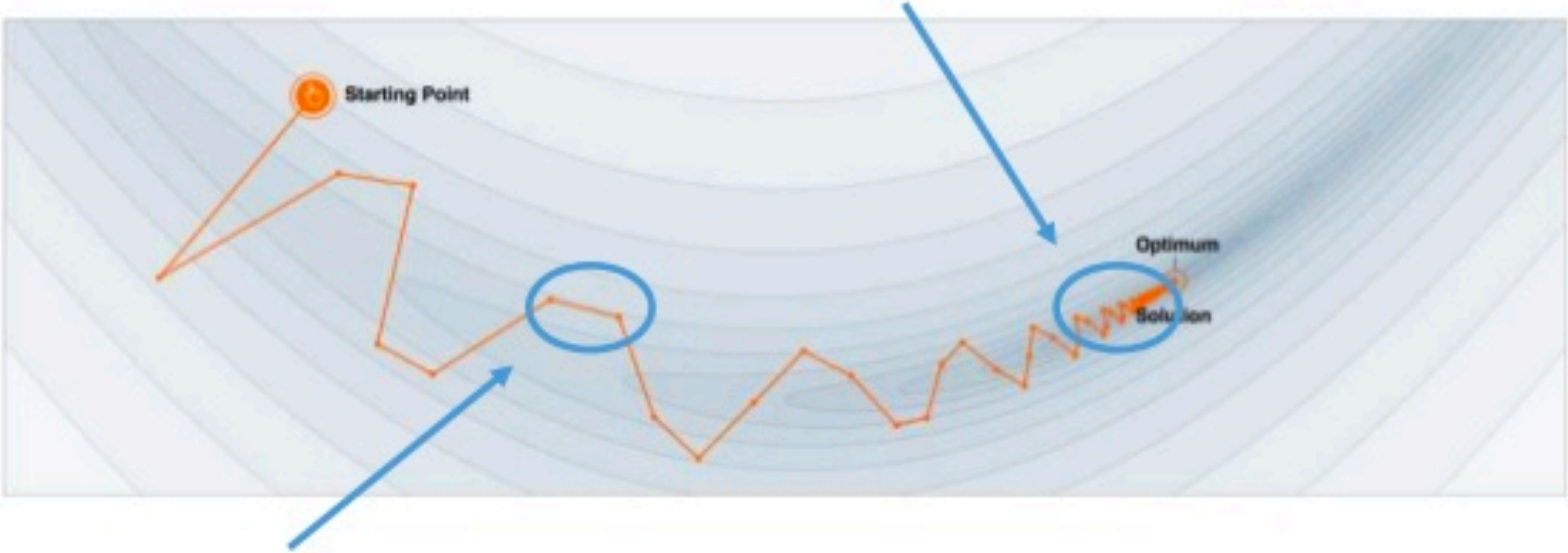
- Make updates based on a mini-batch of examples (instead of a single example):
  - the gradient is the average regularized loss for that mini-batch
  - can give a more accurate estimate of the gradient
  - can leverage matrix/matrix operations, which are more efficient

- Momentum: Can use an exponential average of previous gradients:

$$\overline{\nabla}_{\boldsymbol{\theta}}^{(t)} = \nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)}) + \beta \overline{\nabla}_{\boldsymbol{\theta}}^{(t-1)}$$

Why Momentum really works?

The momentum term **reduces updates for dimensions whose gradients change directions.**



The momentum term **increases for dimensions whose gradients point in the same directions.**

Demo : http://distill.pub/2017/momentum/

Adapting Learning Rates

- Updates with adaptive learning rates ("one learning rate per parameter")
  - **Adagrad**: learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\gamma^{(t)} = \gamma^{(t-1)} + \left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2 \qquad \overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

  - **RMSProp**: instead of cumulative sum, use exponential moving average

$$\gamma^{(t)} = \beta\gamma^{(t-1)} + (1-\beta)\left(\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})\right)^2$$

$$\overline{\nabla}_\theta^{(t)} = \frac{\nabla_\theta l(\mathbf{f}(\mathbf{x}^{(t)}), y^{(t)})}{\sqrt{\gamma^{(t)} + \epsilon}}$$

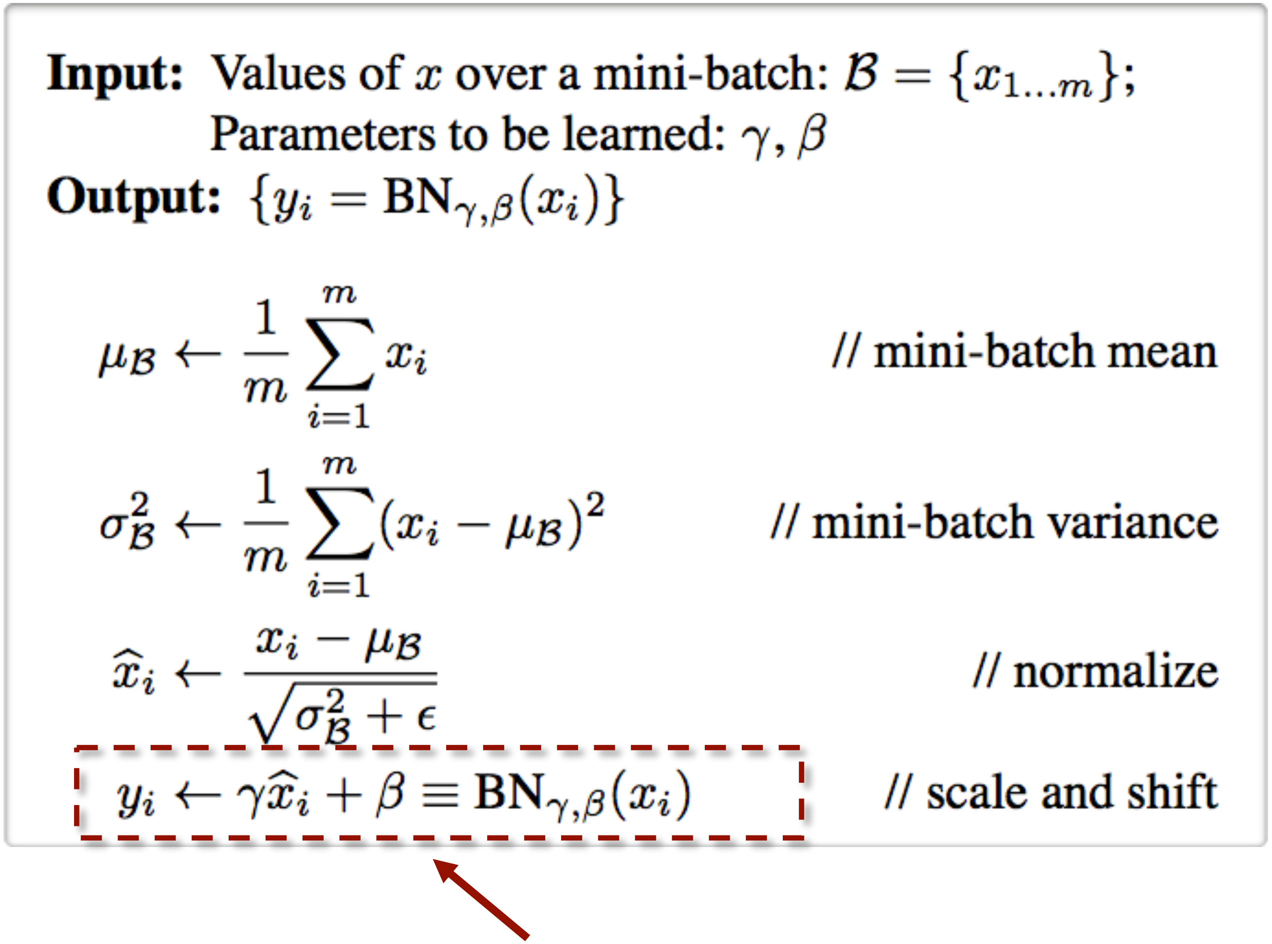
  - **Adam**: essentially combines RMSProp with momentum

Batch Normalization

- Normalizing the inputs will speed up training (Lecun et al. 1998)
  - could normalization be useful at the level of the hidden layers?

- Batch normalization is an attempt to do that (Ioffe and Szegedy, 2014)
  - each unit's pre-activation is normalized (mean subtraction, stddev division)
  - during training, mean and stddev is computed for each minibatch
  - backpropagation takes into account the normalization
  - at test time, the global mean / stddev is used

Batch Normalization



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

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

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

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

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

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

Learned linear transformation to adapt to non-linear activation function (γ and β are trained)

References

- Chapter 7-8, Deep Learning book

Disclaim

- Some slides are taken from Ruslan Salakhutdinov’s deep learning course at CMU.