Machine Learning – Lecture 10

Neural Networks

26.11.2018

Bastian Leibe
RWTH Aachen
http://www.vision.rwth-aachen.de

leibe@vision.rwth-aachen.de

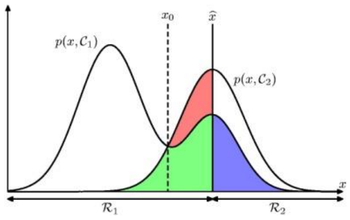
Today's Topic

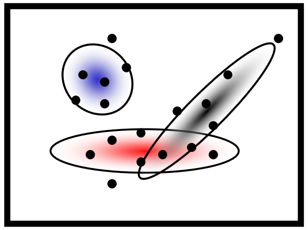
Deep Learning

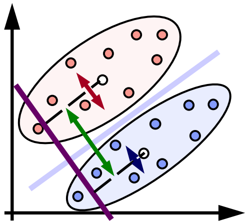
wallchan.com

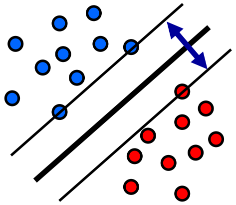
Course Outline

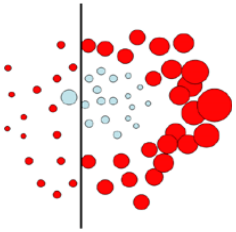
- Fundamentals
    - Bayes Decision Theory
    - Probability Density Estimation
- Classification Approaches
    - Linear Discriminants
    - Support Vector Machines
    - Ensemble Methods & Boosting
    - (Random Forests)
- Deep Learning
    - Foundations
    - Convolutional Neural Networks
    - Recurrent Neural Networks

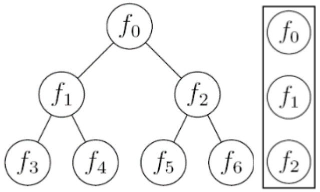


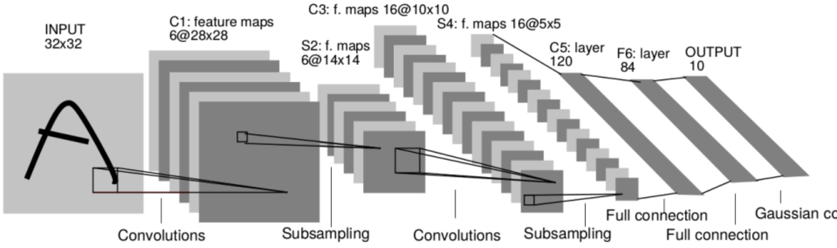












Recap: AdaBoost – "Adaptive Boosting"

- Main idea [Freund & Schapire, 1996]
  - Iteratively select an ensemble of component classifiers
  - After each iteration, reweight misclassified training examples.
    - Increase the chance of being selected in a sampled training set.
    - Or increase the misclassification cost when training on the full set.
- Components
  - $h_m(\mathbf{x})$: "weak" or base classifier
    - Condition: <50% training error over any distribution
  - $H(\mathbf{x})$: "strong" or final classifier
- AdaBoost:
  - Construct a strong classifier as a thresholded linear combination of the weighted weak classifiers:

$$H(\mathbf{x}) = sign\left(\sum_{m=1}^{M} \alpha_m h_m(\mathbf{x})\right)$$

Recap: AdaBoost – Algorithm

1. Initialization: Set $w_n^{(1)} = \frac{1}{N}$ for $n = 1,\ldots,N$.
2. For $m = 1,\ldots,M$ iterations
   a) Train a new weak classifier $h_m(\mathbf{x})$ using the current weighting coefficients $\mathbf{W}^{(m)}$ by minimizing the weighted error function

   $$J_m = \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}) \neq t_n)$$

   $$I(A) = \begin{cases} 1, & \text{if } A \text{ is true} \\ 0, & \text{else} \end{cases}$$

   b) Estimate the weighted error of this classifier on $\mathbf{X}$:

   $$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$$

   c) Calculate a weighting coefficient for $h_m(\mathbf{x})$:

   $$\alpha_m = \ ?$$

   d) Update the weighting coefficients:

   $$w_n^{(m+1)} = \ ?$$

How should we do this exactly?

Recap: Minimizing Exponential Error

- The original algorithm used an exponential error function

$$E = \sum_{n=1}^{N} \exp\{-t_n f_m(\mathbf{x}_n)\}$$

  - where $f_m(\mathbf{x})$ is a classifier defined as a linear combination of base classifiers $h_l(\mathbf{x})$:

$$f_m(\mathbf{x}) = \frac{1}{2} \sum_{l=1}^{m} \alpha_l h_l(\mathbf{x})$$

- Goal
  - Minimize E with respect to both the weighting coefficients α_l and the parameters of the base classifiers $h_l(\mathbf{x})$.

Recap: Minimizing Exponential Error

- Sequential Minimization (continuation from last lecture)
  - Only minimize with respect to α_m and $h_m(\mathbf{x})$

$$E = \sum_{n=1}^{N} \exp\left\{-t_n f_m(\mathbf{x}_n)\right\} \qquad \text{with} \quad f_m(\mathbf{x}) = \frac{1}{2}\sum_{l=1}^{m} \alpha_l h_l(\mathbf{x})$$

$$= \sum_{n=1}^{N} \exp\left\{\underbrace{-t_n f_{m-1}(\mathbf{x}_n)}_{= \textit{const.}} - \frac{1}{2} t_n \alpha_m h_m(\mathbf{x}_n)\right\}$$

$$= \sum_{n=1}^{N} w_n^{(m)} \exp\left\{-\frac{1}{2} t_n \alpha_m h_m(\mathbf{x}_n)\right\} \quad = \ \ldots$$

$$E = \left(e^{\alpha_m/2} - e^{-\alpha_m/2}\right) \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}_n) \neq t_n) + e^{-\alpha_m/2} \sum_{n=1}^{N} w_n^{(m)}$$

AdaBoost – Minimizing Exponential Error

- Minimize with respect to $h_m(\mathbf{x})$: $\frac{\partial E}{\partial h_m(\mathbf{x}_n)} \overset{!}{=} 0$

$$E = \underbrace{\left(e^{\alpha_m/2} - e^{-\alpha_m/2}\right)}_{= \textit{const.}} \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}_n) \neq t_n) + \underbrace{e^{-\alpha_m/2} \sum_{n=1}^{N} w_n^{(m)}}_{= \textit{const.}}$$

⇒ This is equivalent to minimizing

$$J_m = \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}) \neq t_n)$$

(our weighted error function from step 2a) of the algorithm)

⇒ *We're on the right track. Let's continue…*

AdaBoost – Minimizing Exponential Error

- Minimize with respect to α_m: $\quad \dfrac{\partial E}{\partial \alpha_m} \overset{!}{=} 0$

$$E = \left(e^{\alpha_m/2} - e^{-\alpha_m/2}\right) \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}_n) \neq t_n) + e^{-\alpha_m/2} \sum_{n=1}^{N} w_n^{(m)}$$

$$\left(\frac{1}{\cancel{2}} e^{\alpha_m/2} + \frac{1}{\cancel{2}} e^{-\alpha_m/2}\right) \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}_n) \neq t_n) \overset{!}{=} \frac{1}{\cancel{2}} e^{-\alpha_m/2} \sum_{n=1}^{N} w_n^{(m)}$$

weighted error $\epsilon_m :=$

$$\boxed{\frac{\sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}_n) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}} = \frac{e^{-\alpha_m/2}}{e^{\alpha_m/2} + e^{-\alpha_m/2}}$$

$$\epsilon_m = \frac{1}{e^{\alpha_m} + 1}$$

$\Rightarrow$ *Update for the α coefficients:*

$$\alpha_m = \ln\left\{\frac{1-\epsilon_m}{\epsilon_m}\right\}$$

AdaBoost – Minimizing Exponential Error

- Remaining step: update the weights
  - Recall that

$$E = \sum_{n=1}^{N} \underbrace{w_n^{(m)} \exp\left\{-\frac{1}{2} t_n \alpha_m h_m(\mathbf{x}_n)\right\}}_{\text{This becomes } w_n^{(m+1)} \text{ in the next iteration.}}$$

  - Therefore

$$\begin{aligned} w_n^{(m+1)} &= w_n^{(m)} \exp\left\{-\frac{1}{2} t_n \alpha_m h_m(\mathbf{x}_n)\right\} \\ &= \ldots \\ &= w_n^{(m)} \exp\left\{\alpha_m I(h_m(\mathbf{x}_n) \neq t_n)\right\} \end{aligned}$$

⇒ *Update for the weight coefficients.*

AdaBoost – Final Algorithm

1. Initialization: Set $w_n^{(1)} = \frac{1}{N}$ for $n = 1,\ldots,N$.
2. For $m = 1,\ldots,M$ iterations
   a) Train a new weak classifier $h_m(\mathbf{x})$ using the current weighting coefficients $\mathbf{W}^{(m)}$ by minimizing the weighted error function

      $$J_m = \sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}) \neq t_n)$$

   b) Estimate the weighted error of this classifier on $\mathbf{X}$:

      $$\epsilon_m = \frac{\sum_{n=1}^{N} w_n^{(m)} I(h_m(\mathbf{x}) \neq t_n)}{\sum_{n=1}^{N} w_n^{(m)}}$$

   c) Calculate a weighting coefficient for $h_m(\mathbf{x})$:

      $$\alpha_m = \ln\left\{\frac{1-\epsilon_m}{\epsilon_m}\right\}$$

   d) Update the weighting coefficients:

      $$w_n^{(m+1)} = w_n^{(m)} \exp\left\{\alpha_m I(h_m(\mathbf{x}_n) \neq t_n)\right\}$$

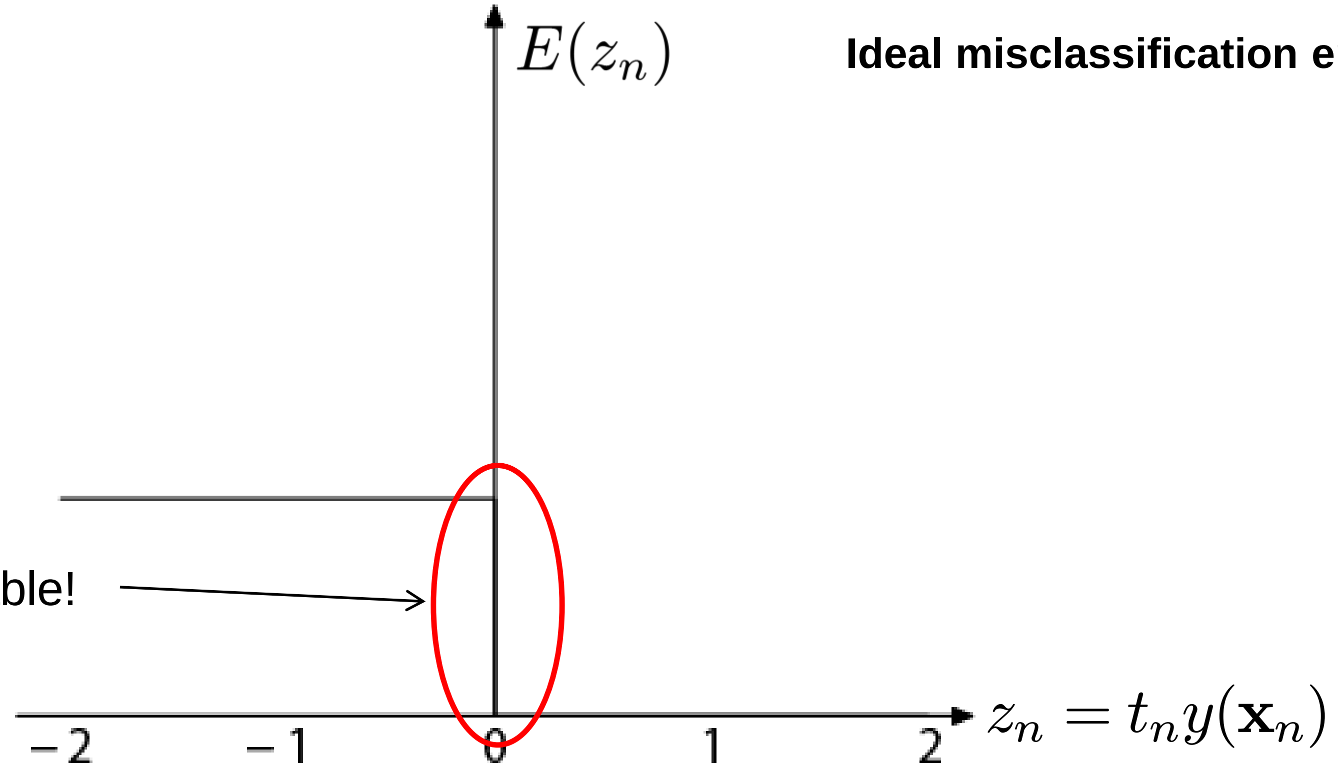
AdaBoost – Analysis

- Result of this derivation
  - We now know that AdaBoost minimizes an exponential error function in a sequential fashion.
  - This allows us to analyze AdaBoost's behavior in more detail.
  - In particular, we can see how robust it is to outlier data points.

Recap: Error Functions

$t_n \in \{-1, 1\}$

Ideal misclassification error



Not differentiable!

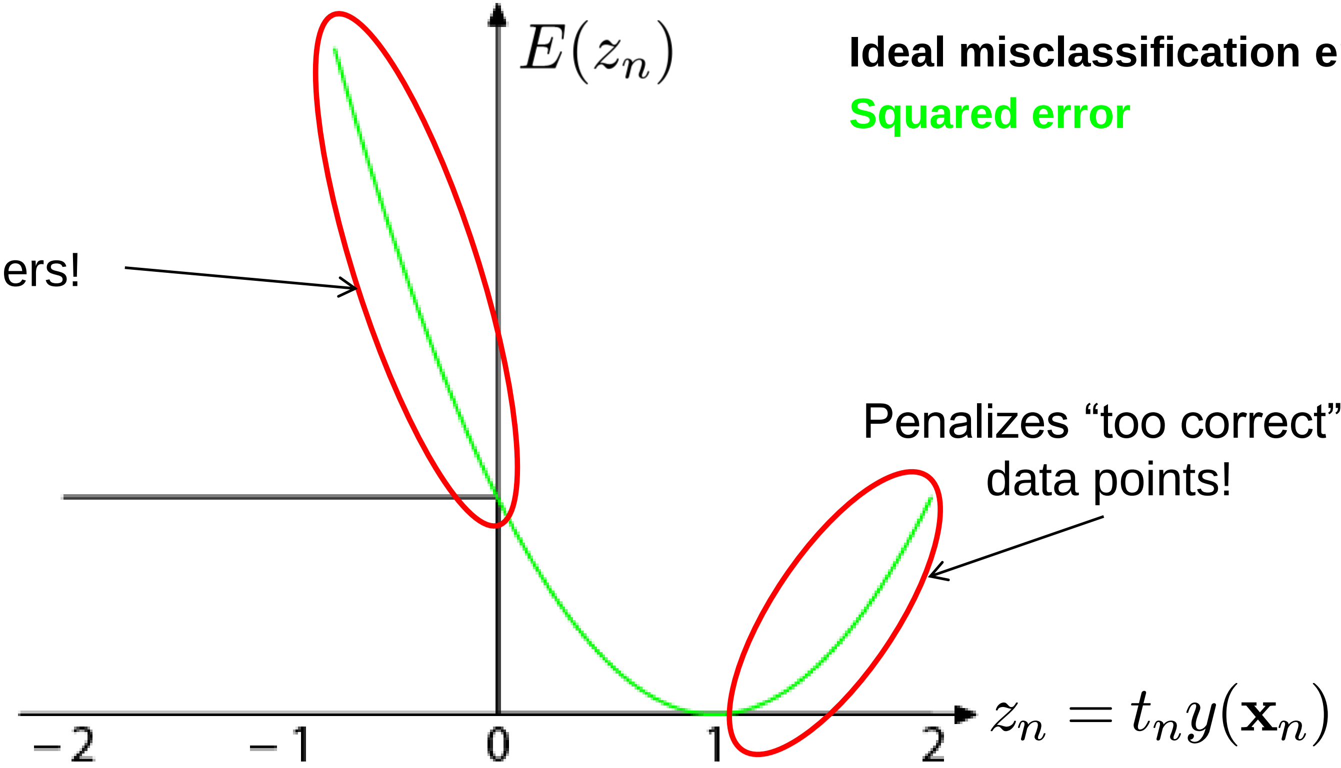
- Ideal misclassification error function (black)
  - This is what we want to approximate,
  - Unfortunately, it is not differentiable.
  - The gradient is zero for misclassified points.

$\Rightarrow$ We cannot minimize it by gradient descent.

Image source: Bishop, 2006

Recap: Error Functions

$t_n \in \{-1, 1\}$



Ideal misclassification error

Squared error

Sensitive to outliers!

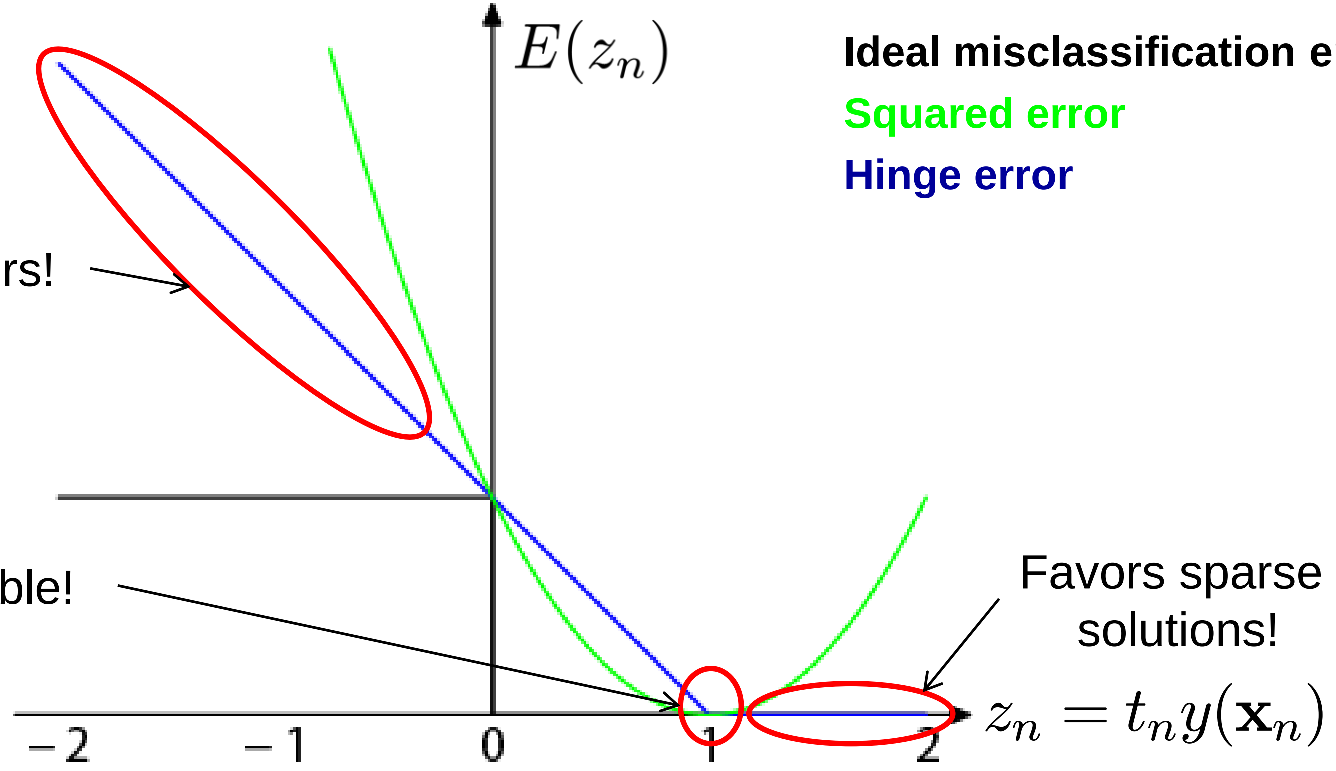
Penalizes “too correct” data points!

- Squared error used in Least-Squares Classification
  - Very popular, leads to closed-form solutions.
  - However, sensitive to outliers due to squared penalty.
  - Penalizes “too correct” data points

  $\Rightarrow$ Generally does not lead to good classifiers.

Image source: Bishop, 2006

Recap: Error Functions



Ideal misclassification error

Squared error

Hinge error

Robust to outliers!

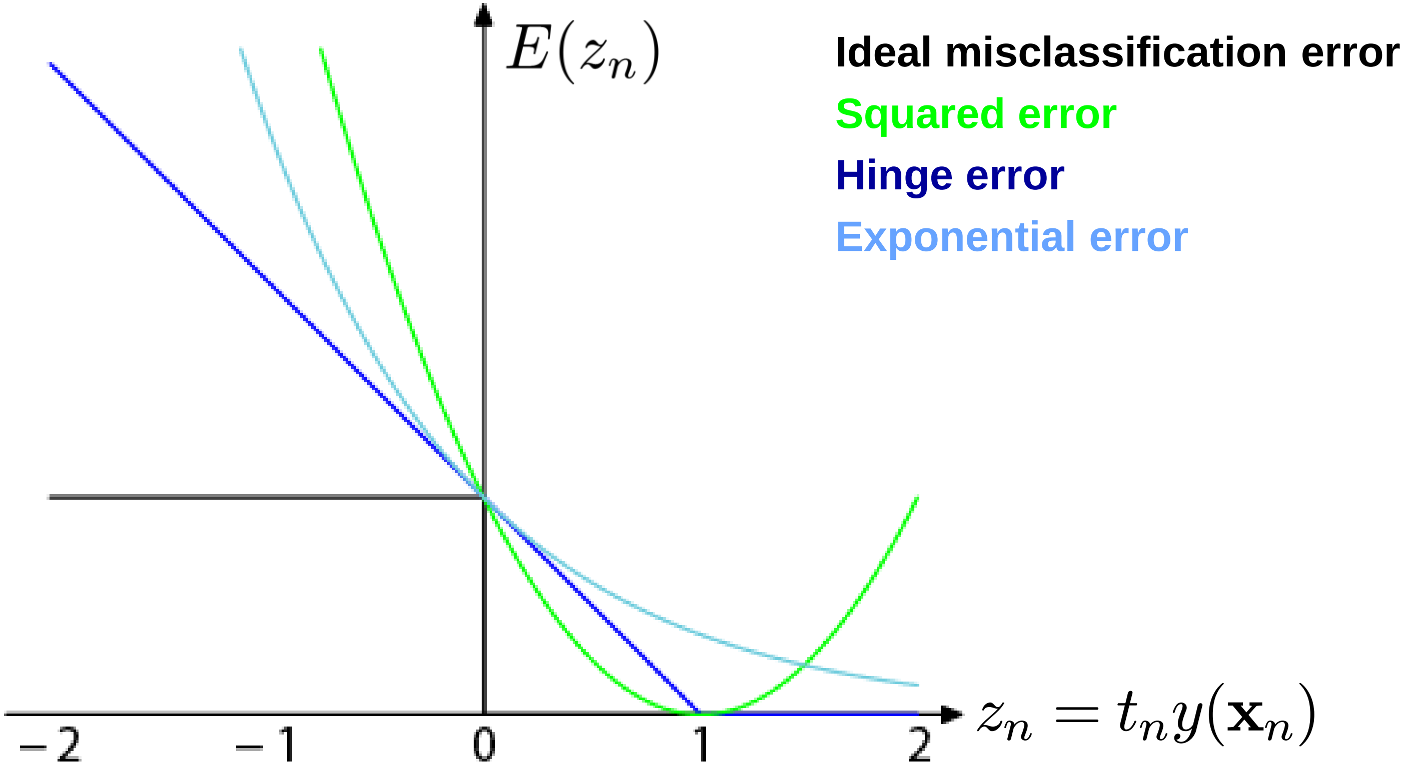
Not differentiable!

Favors sparse solutions!

- "Hinge error" used in SVMs
  - Zero error for points outside the margin ($z_n > 1$) ⇒ sparsity
  - Linear penalty for misclassified points ($z_n < 1$) ⇒ robustness
  - Not differentiable around $z_n = 1$ ⇒ Cannot be optimized directly.

Image source: Bishop, 2006

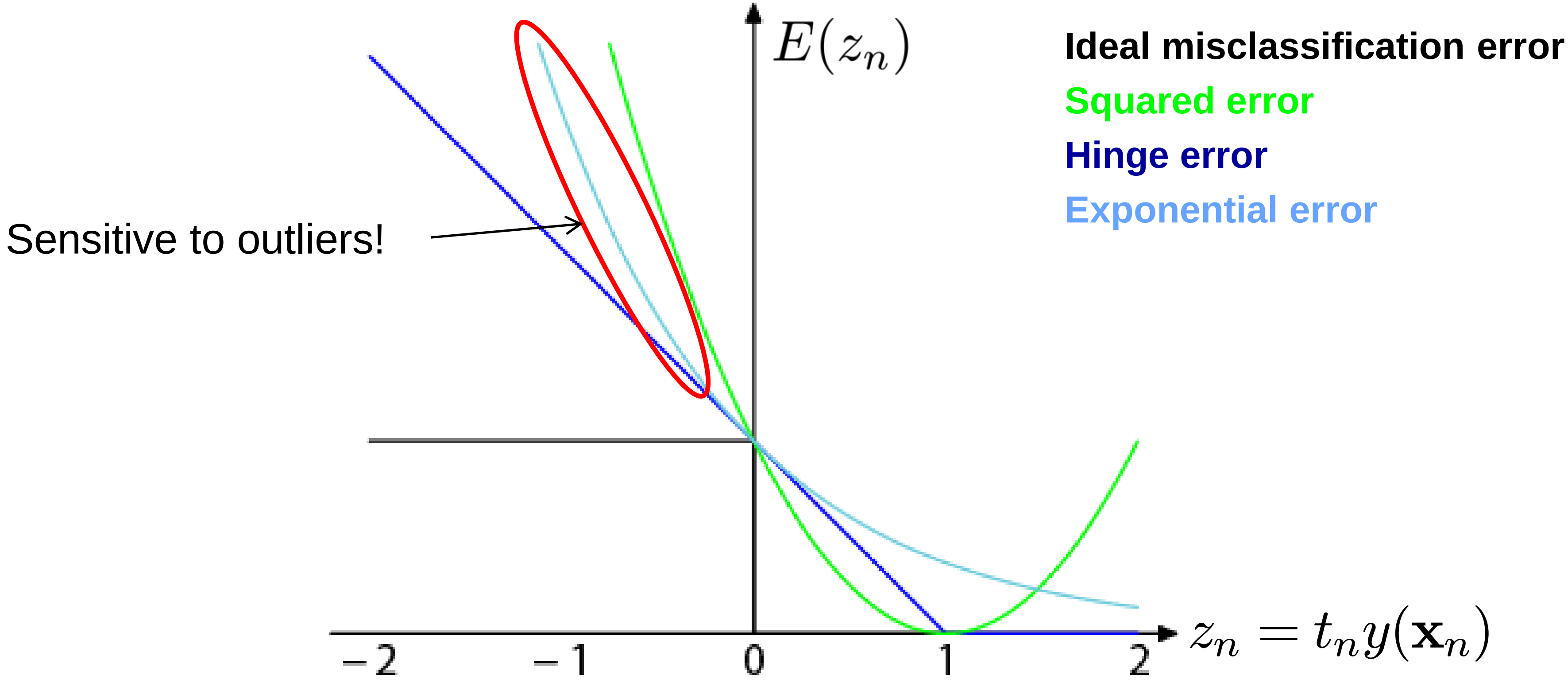
Discussion: AdaBoost Error Function



- Exponential error used in AdaBoost
  - Continuous approximation to ideal misclassification function.
  - Sequential minimization leads to simple AdaBoost scheme.
  - Properties?

Image source: Bishop, 2006

Discussion: AdaBoost Error Function



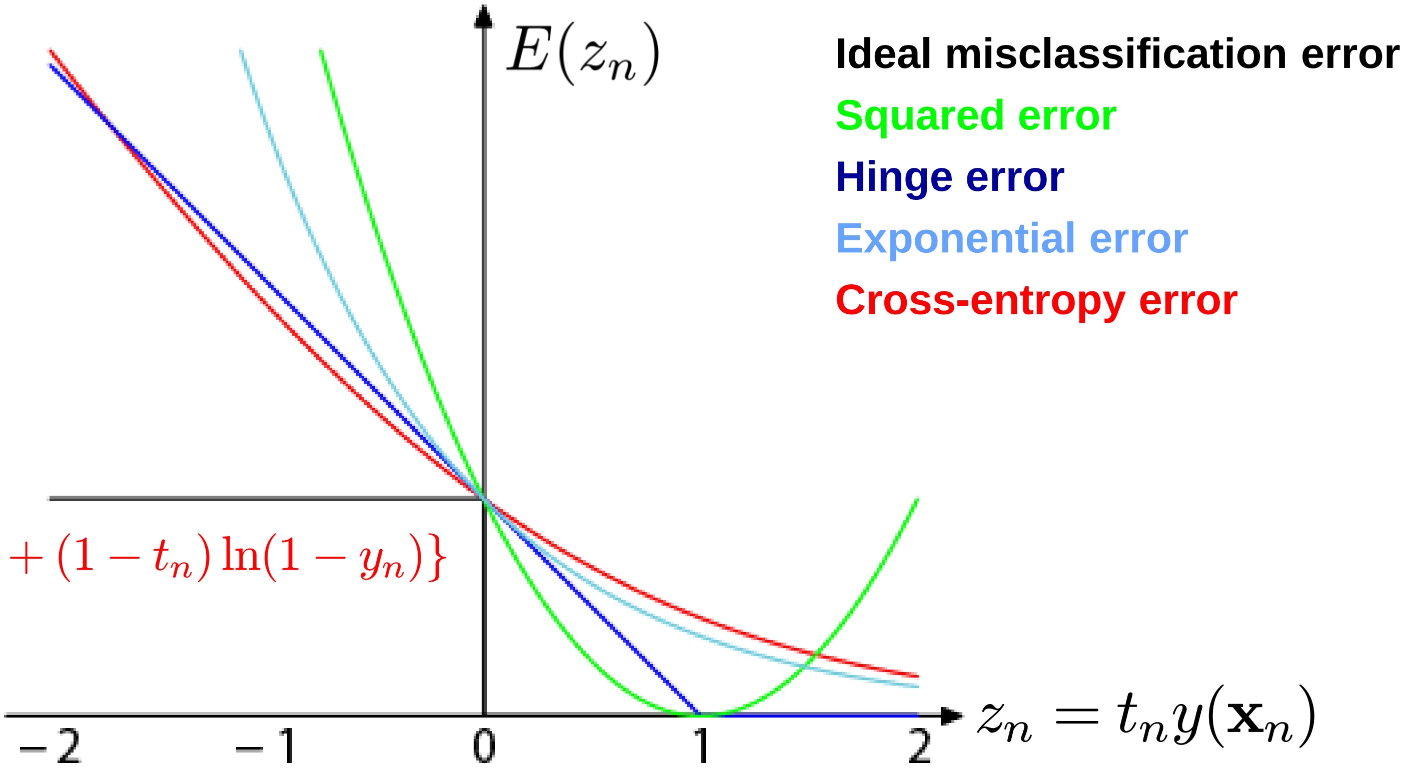
Sensitive to outliers!

- Exponential error used in AdaBoost
  - No penalty for too correct data points, fast convergence.
  - Disadvantage: exponential penalty for large negative values!

  $\Rightarrow$ Less robust to outliers or misclassified data points!

Image source: Bishop, 2006

Discussion: Other Possible Error Functions



$$E = -\sum\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\}$$

- “Cross-entropy error” used in Logistic Regression
  - Similar to exponential error for $z>0$.
  - Only grows linearly with large negative values of z.

⇒ Make AdaBoost more robust by switching to this error function.

⇒ “GentleBoost”

Image source: Bishop, 2006

Summary: AdaBoost

- Properties
  - Simple combination of multiple classifiers.
  - Easy to implement.
  - Can be used with many different types of classifiers.
    - None of them needs to be too good on its own.
    - In fact, they only have to be slightly better than chance.
  - Commonly used in many areas.
  - Empirically good generalization capabilities.
- Limitations
  - Original AdaBoost sensitive to misclassified training data points.
    - Because of exponential error function.
    - Improvement by GentleBoost
  - Single-class classifier
    - Multiclass extensions available

Today's Topic

Deep Learning

wallchan.com

Topics of This Lecture

- A Brief History of Neural Networks

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits

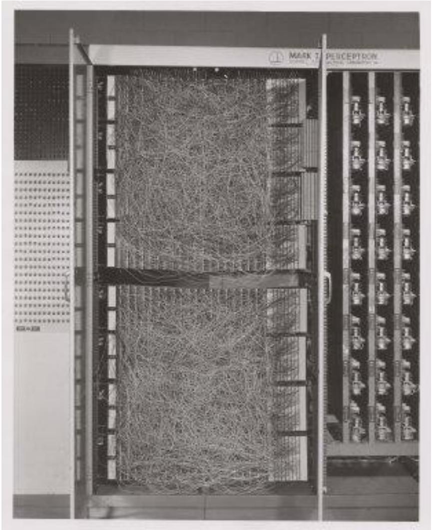
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units

- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

- And a cool learning algorithm: “Perceptron Learning”
- Hardware implementation “Mark I Perceptron” for 20×20 pixel image analysis



The New York Times

“The embryo of an electronic computer that [...] will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.”

Image source: Wikipedia, clipartpanda.com

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

- They showed that (single-layer) Perceptrons cannot solve all problems.
- This was misunderstood by many that they were worthless.

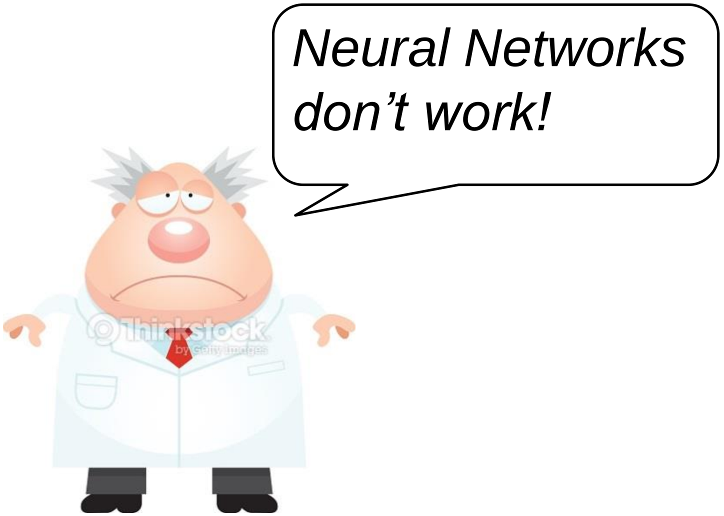


Image source: colourbox.de, thinkstock

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

- Some notable successes with multi-layer perceptrons.
- Backpropagation learning algorithm

OMG! They work like the human brain!

Oh no! Killer robots will achieve world domination!

Image sources: clipartpanda.com, cliparts.co

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

- Some notable successes with multi-layer perceptrons.
- Backpropagation learning algorithm
- But they are hard to train, tend to overfit, and have unintuitive parameters.
- So, the excitement fades again…

Image source: clipartof.com, colourbox.de

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

- Notably Support Vector Machines
- Machine Learning becomes a discipline of its own.

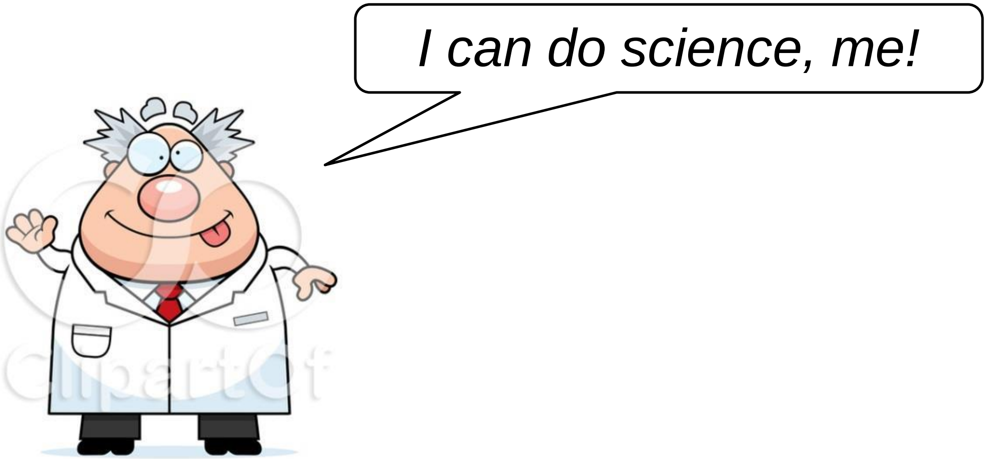


Image source: clipartof.com

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

- Notably Support Vector Machines
- Machine Learning becomes a discipline of its own.
- The general public and the press still love Neural Networks.

I'm doing Machine Learning.

So, you're using Neural Networks?

Actually...

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

2005+ Gradual progress

- Better understanding how to successfully train deep networks
- Availability of large datasets and powerful GPUs
- Still largely under the radar for many disciplines applying ML

Are you using Neural Networks?

Come on. Get real!

A Brief History of Neural Networks

1957 Rosenblatt invents the Perceptron

1969 Minsky & Papert

1980s Resurgence of Neural Networks

1995+ Interest shifts to other learning methods

2005+ Gradual progress

2012 Breakthrough results

- ImageNet Large Scale Visual Recognition Challenge
- A ConvNet halves the error rate of dedicated vision approaches.
- Deep Learning is widely adopted.

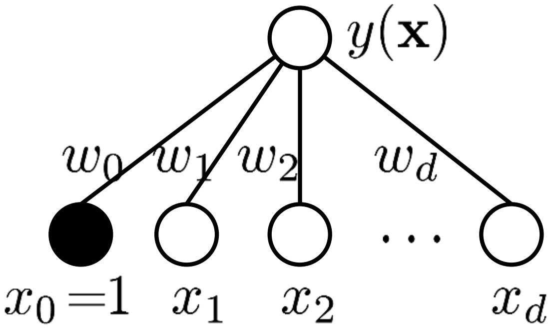
Image source: clipartpanda.com, clipartof.com

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Perceptrons (Rosenblatt 1957)

- Standard Perceptron



Output layer

Weights

Input layer

- Input Layer
  - Hand-designed features based on common sense

- Outputs
  - Linear outputs

$$y(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + w_0$$

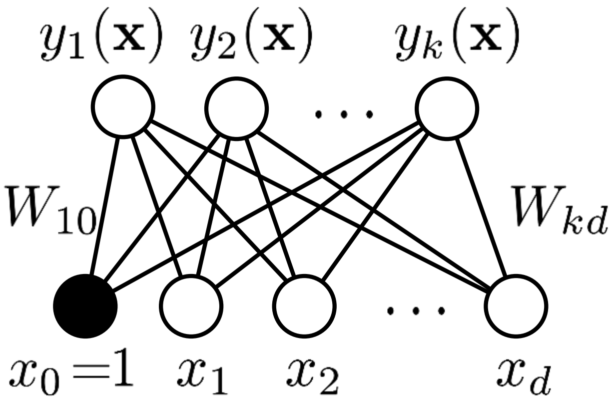
Logistic outputs

$$y(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + w_0)$$

- Learning = Determining the weights $\mathbf{w}$

Extension: Multi-Class Networks

- One output node per class



Output layer

Weights

Input layer

- Outputs
  - Linear outputs

$$y_k(\mathbf{x}) = \sum_{i=0}^{d} W_{ki} x_i$$

Logistic outputs

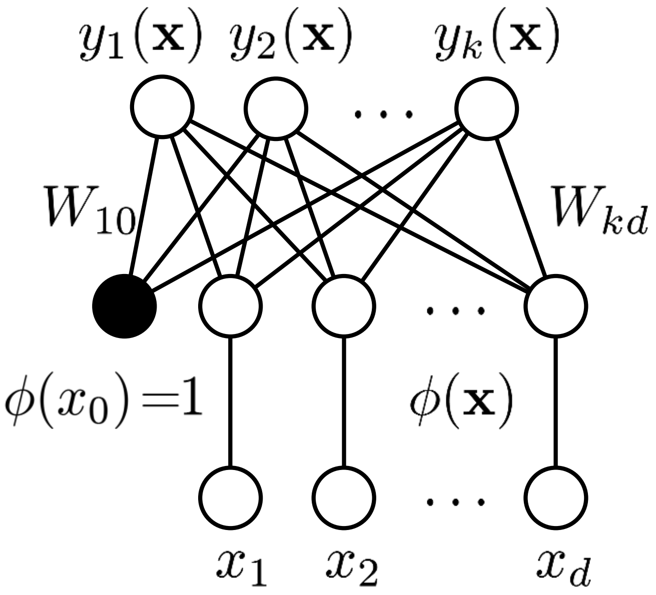
$$y_k(\mathbf{x}) = \sigma\left(\sum_{i=0}^{d} W_{ki} x_i\right)$$

$\Rightarrow$ Can be used to do multidimensional linear regression or multiclass classification.

Slide adapted from Stefan Roth

Extension: Non-Linear Basis Functions

- Straightforward generalization



Output layer

Weights

Feature layer

Mapping (fixed)

Input layer

- Outputs
  - Linear outputs

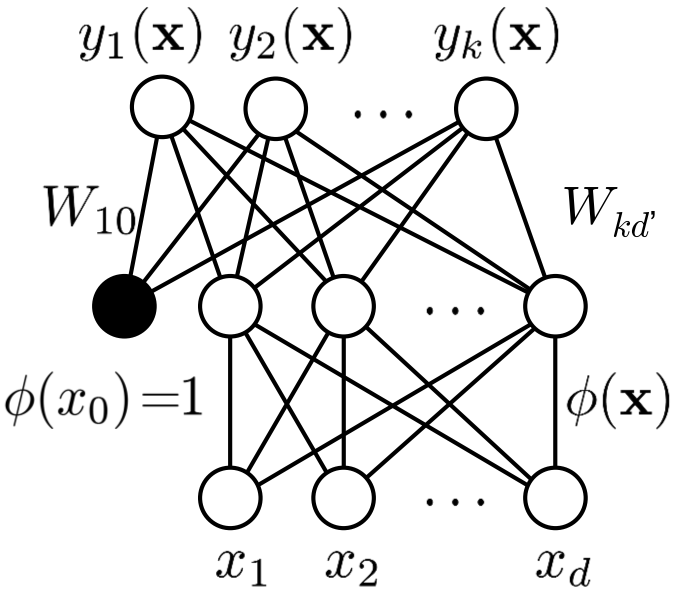
$$y_k(\mathbf{x}) = \sum_{i=0}^{d} W_{ki}\phi(x_i)$$

Logistic outputs

$$y_k(\mathbf{x}) = \sigma\left(\sum_{i=0}^{d} W_{ki}\phi(x_i)\right)$$

Extension: Non-Linear Basis Functions

- Straightforward generalization



Output layer

Weights

Feature layer

Mapping (fixed)

Input layer

- Remarks
  - Perceptrons are generalized linear discriminants!
  - Everything we know about the latter can also be applied here.
  - Note: feature functions $\phi(\mathbf{x})$ are kept fixed, not learned!

Perceptron Learning

- Very simple algorithm
- Process the training cases in some permutation
  - If the output unit is correct, leave the weights alone.
  - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.
- This is guaranteed to converge to a correct solution if such a solution exists.

Slide adapted from Geoff Hinton

Perceptron Learning

- Let’s analyze this algorithm...
- Process the training cases in some permutation
  - If the output unit is correct, leave the weights alone.
  - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.
- Translation

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)}$$

Slide adapted from Geoff Hinton

Perceptron Learning

- Let's analyze this algorithm...
- Process the training cases in some permutation
  - If the output unit is correct, leave the weights alone.
  - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.
- Translation

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \left(y_k(\mathbf{x}_n; \mathbf{w}) - t_{kn} \right) \phi_j(\mathbf{x}_n)$$

  - This is the Delta rule a.k.a. LMS rule!

⇒ Perceptron Learning corresponds to 1st-order (stochastic) Gradient Descent (e.g., of a quadratic error function)!

Slide adapted from Geoff Hinton

Loss Functions

- We can now also apply other loss functions
  - L2 loss ⇒ Least-squares regression

    $$L(t, y(\mathbf{x})) = \sum_n (y(\mathbf{x}_n) - t_n)^2$$

  - L1 loss: ⇒ Median regression

    $$L(t, y(\mathbf{x})) = \sum_n |y(\mathbf{x}_n) - t_n|$$

  - Cross-entropy loss ⇒ Logistic regression

    $$L(t, y(\mathbf{x})) = -\sum_n \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

  - Hinge loss ⇒ SVM classification

    $$L(t, y(\mathbf{x})) = \sum_n [1 - t_n y(\mathbf{x}_n)]_+$$

  - Softmax loss ⇒ Multi-class probabilistic classification

    $$L(t, y(\mathbf{x})) = -\sum_n \sum_k \left\{ \mathbb{I}(t_n = k) \ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))} \right\}$$

Regularization

- In addition, we can apply regularizers
  - E.g., an L2 regularizer

$$E(\mathbf{w}) = \sum_n L(t_n, y(\mathbf{x}_n; \mathbf{w})) + \lambda ||\mathbf{w}||^2$$

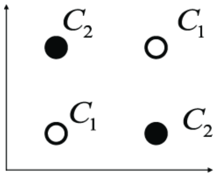
  - This is known as weight decay in Neural Networks.
  - We can also apply other regularizers, e.g. L1 ⇒ sparsity
  - Since Neural Networks often have many parameters, regularization becomes very important in practice.
  - We will see more complex regularization techniques later on...

Limitations of Perceptrons

- What makes the task difficult?
  - Perceptrons with fixed, hand-coded input features can model any separable function perfectly...
  - ...given the right input features.
  - For some tasks this requires an exponential number of input features.
    - E.g., by enumerating all possible binary input vectors as separate feature units (similar to a look-up table).
    - But this approach won't generalize to unseen test cases!

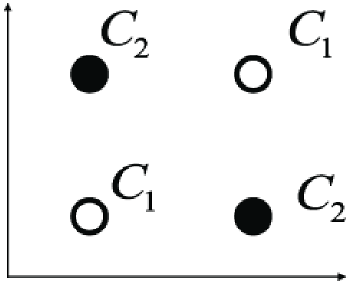
  $\Rightarrow$ *It is the feature design that solves the task!*

  - Once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.
    - Classic example: XOR function.



Wait...

- Didn't we just say that...
  - Perceptrons correspond to generalized linear discriminants
  - And Perceptrons are very limited...
  - *Doesn't this mean that what we have been doing so far in this lecture has the same problems???*
- Yes, this is the case.
  - A linear classifier cannot solve certain problems (e.g., XOR).
  - However, with a non-linear classifier based on the right kind of features, the problem becomes solvable.



$\Rightarrow$ So far, we have solved such problems by hand-designing good features ϕ and kernels $\phi^\top\phi$.

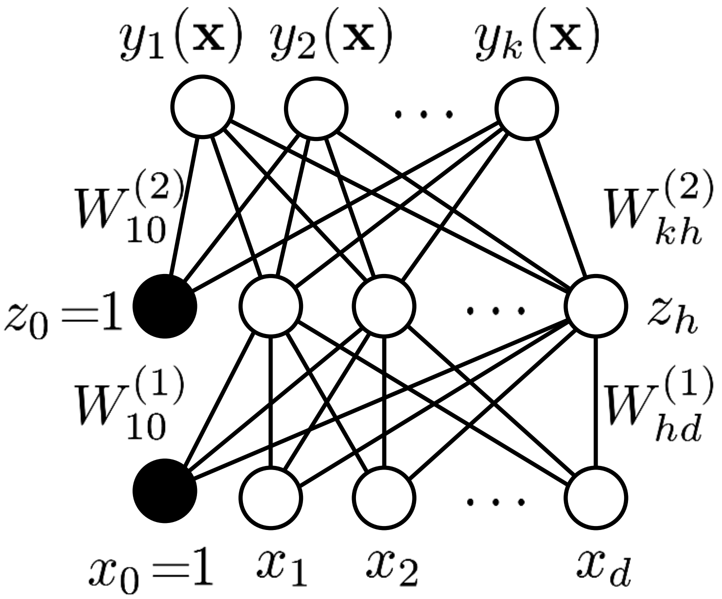
$\Rightarrow$ *Can we also learn such feature representations?*

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Multi-Layer Perceptrons

- Adding more layers



Output layer

Hidden layer

Mapping (learned!)

Input layer

- Output

$$y_k(\mathbf{x}) = g^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j\right)\right)$$

Slide adapted from Stefan Roth

Multi-Layer Perceptrons

$$y_k(\mathbf{x}) = g^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} g^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j\right)\right)$$

- Activation functions $g^{(k)}$:
  - For example: $g^{(2)}(a) = \sigma(a)$, $g^{(1)}(a) = a$
- The hidden layer can have an arbitrary number of nodes
  - There can also be multiple hidden layers.
- Universal approximators
  - A 2-layer network (1 hidden layer) can approximate any continuous function of a compact domain arbitrarily well!
    (assuming sufficient hidden nodes)

Slide credit: Stefan Roth

Learning with Hidden Units

- Networks without hidden units are very limited in what they can learn
  - More layers of linear units do not help ⇒ still linear
  - Fixed output non-linearities are not enough.

- We need multiple layers of adaptive non-linear hidden units. But how can we train such nets?
  - Need an efficient way of adapting all weights, not just the last layer.
  - Learning the weights to the hidden units = learning features
  - This is difficult, because nobody tells us what the hidden units should do.

⇒ Main challenge in deep learning.

Slide adapted from Geoff Hinton

Learning with Hidden Units

- How can we train multi-layer networks efficiently?
  - Need an efficient way of adapting all weights, not just the last layer.

- Idea: Gradient Descent
  - Set up an error function

$$E(\mathbf{W}) = \sum_n L(t_n, y(\mathbf{x}_n; \mathbf{W})) + \lambda \Omega(\mathbf{W})$$

with a loss $L(\cdot)$ and a regularizer $\Omega(\cdot)$.

  - E.g., $L(t, y(\mathbf{x}; \mathbf{W})) = \sum_n \left(y(\mathbf{x}_n; \mathbf{W}) - t_n\right)^2$ L

L$_2$ loss

$$\Omega(\mathbf{W}) = ||\mathbf{W}||_F^2$$

L$_2$ regularizer
("weight decay")

$\Rightarrow$ Update each weight $W_{ij}^{(k)}$ in the direction of the gradient $\frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(k)}}$

Gradient Descent

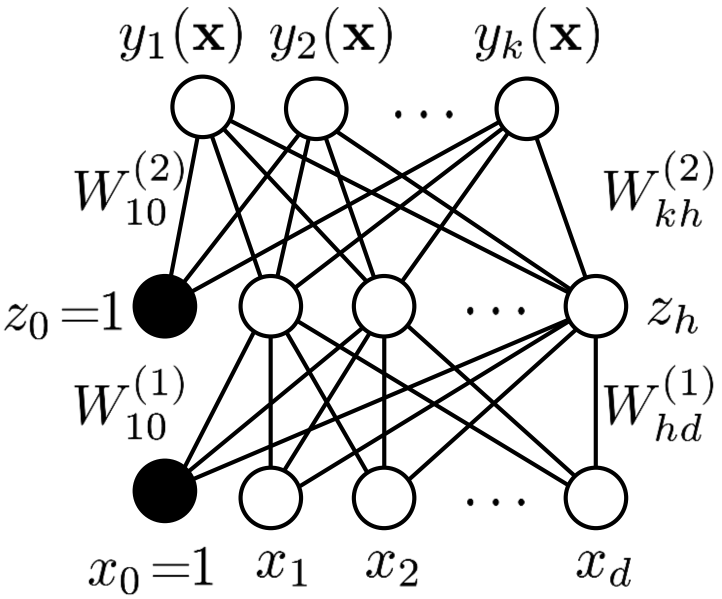
- Two main steps
  1. Computing the gradients for each weight — today
  2. Adjusting the weights in the direction of the gradient — next lecture

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



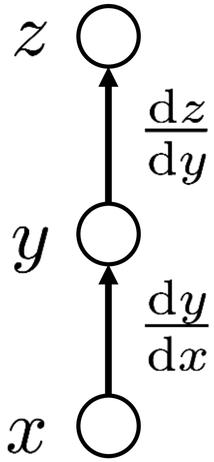
$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(1)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{hd}^{(1)}}$$

  - Compute the gradients for each variable analytically.
  - *What is the problem when doing this?*

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions



$$\Delta z = \frac{\mathrm{d}z}{\mathrm{d}y}\Delta y$$

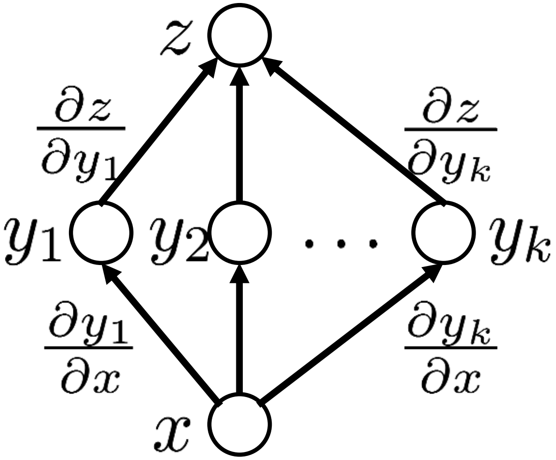
$$\Delta y = \frac{\mathrm{d}y}{\mathrm{d}x}\Delta x$$

$$\Delta z = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}\Delta x$$

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}$$

Excursion: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative



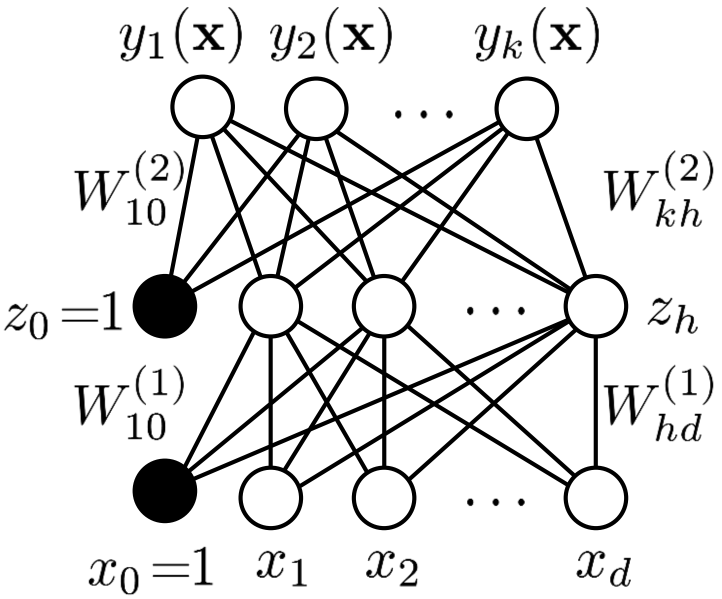
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \dots$$

$$= \sum_{i=1}^{k} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$

$\Rightarrow$ Need to sum over all paths that lead to the target variable x.

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(1)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{hd}^{(1)}}$$

  - Compute the gradients for each variable analytically.
  - *What is the problem when doing this?*

$\Rightarrow$ With increasing depth, there will be exponentially many paths!

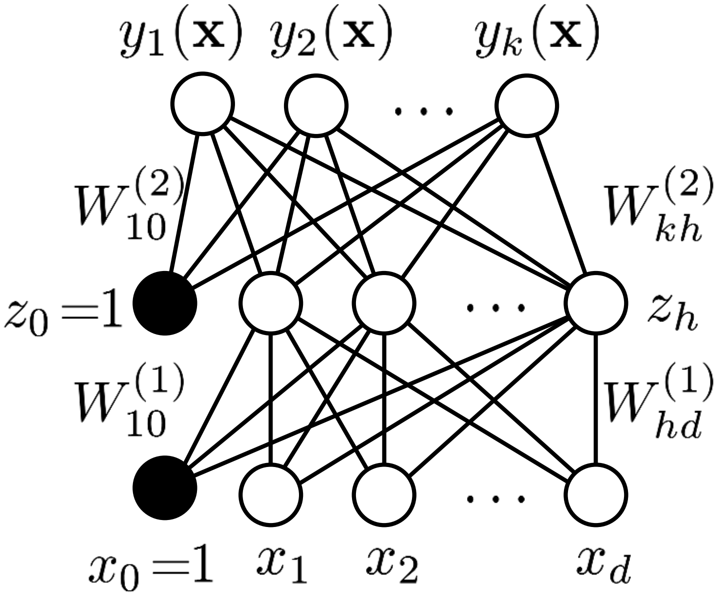
$\Rightarrow$ Infeasible to compute this way.

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Obtaining the Gradients

- Approach 2: Numerical Differentiation



  - Given the current state $\mathbf{W}^{(\tau)}$, we can evaluate $E(\mathbf{W}^{(\tau)})$.
  - Idea: Make small changes to $\mathbf{W}^{(\tau)}$ and accept those that improve $E(\mathbf{W}^{(\tau)})$.

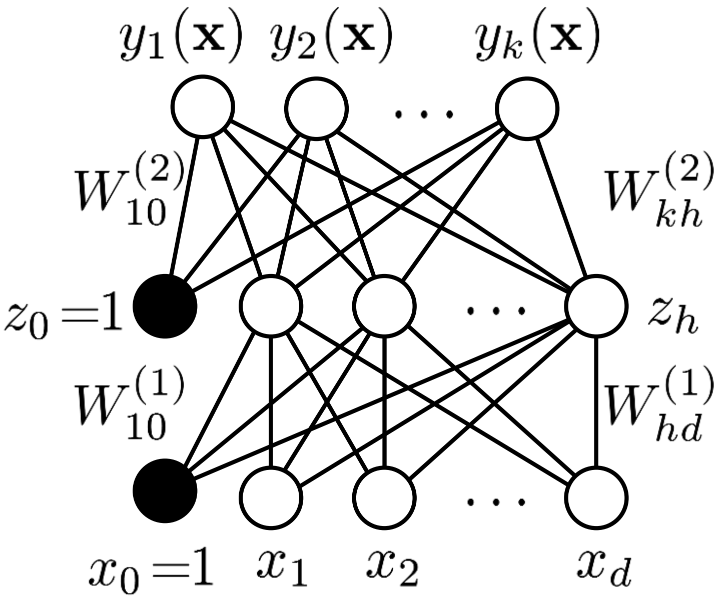
$\Rightarrow$ Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

Topics of This Lecture

- A Brief History of Neural Networks
- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation

Obtaining the Gradients

- Approach 3: Incremental Analytical Differentiation



$$\begin{tikzcd}
\frac{\partial E(\mathbf{W})}{\partial y_j} \arrow[dd] \arrow[dr] & \\
& \frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(2)}} \\
\frac{\partial E(\mathbf{W})}{\partial z_i} \arrow[dd] \arrow[dr] & \\
& \frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(1)}} \\
\frac{\partial E(\mathbf{W})}{\partial x_i} &
\end{tikzcd}$$

  - Idea: Compute the gradients layer by layer.
  - Each layer below builds upon the results of the layer above.

$\Rightarrow$ The gradient is propagated backwards through the layers.

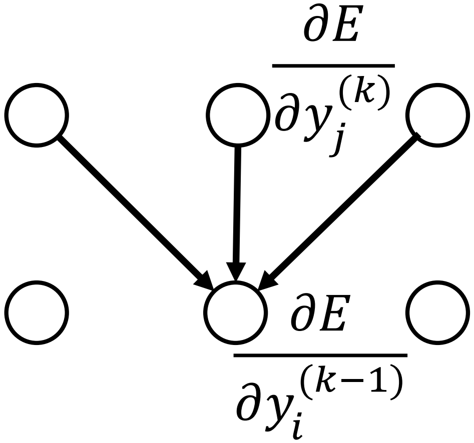
$\Rightarrow$ Backpropagation algorithm

Backpropagation Algorithm

- Core steps
  1. Convert the discrepancy between each output and its target value into an error derivate.
  2. Compute error derivatives in each hidden layer from error derivatives in the layer above.
  3. Use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights

$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

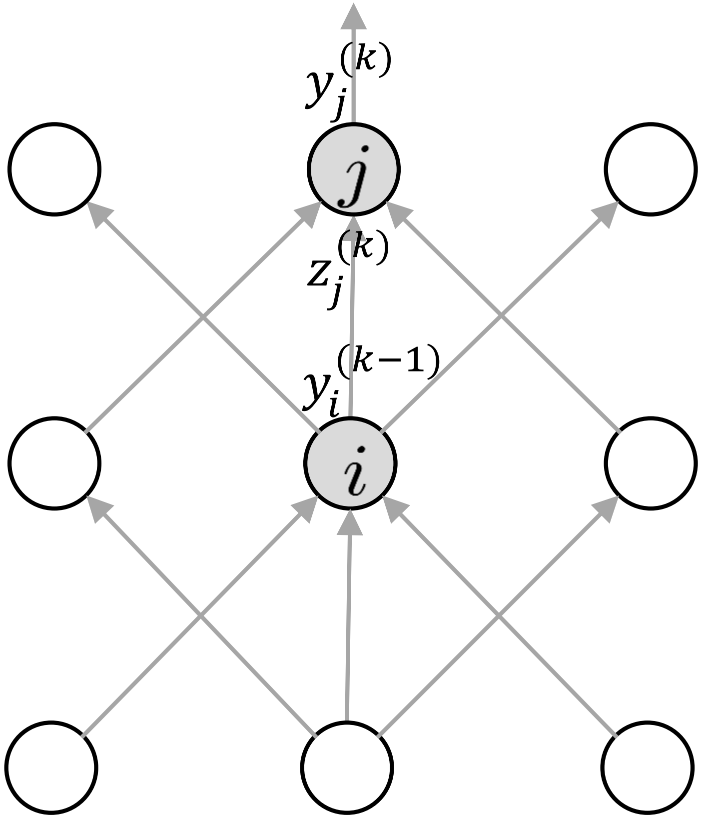
$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



$$\frac{\partial E}{\partial y_j^{(k)}} \longrightarrow \frac{\partial E}{\partial w_{ji}^{(k-1)}}$$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

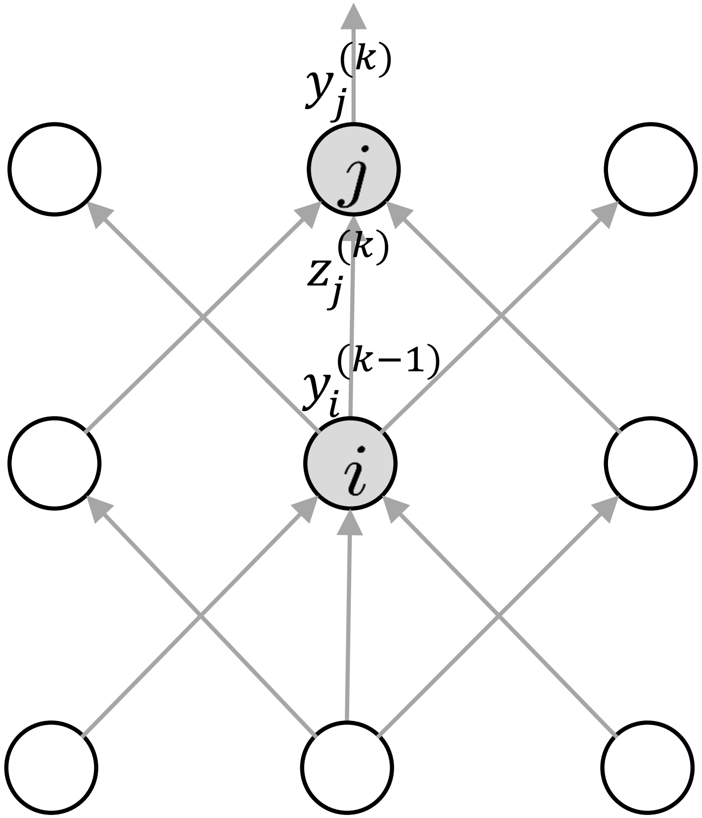
- Notation
  - $y_j^{(k)}$ Output of layer k
  - $z_j^{(k)}$ Input of layer k

Connections: $z_j^{(k)} = \sum_i w_{ji}^{(k-1)} y_i^{(k-1)}$

$y_j^{(k)} = g\left(z_j^{(k)}\right)$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

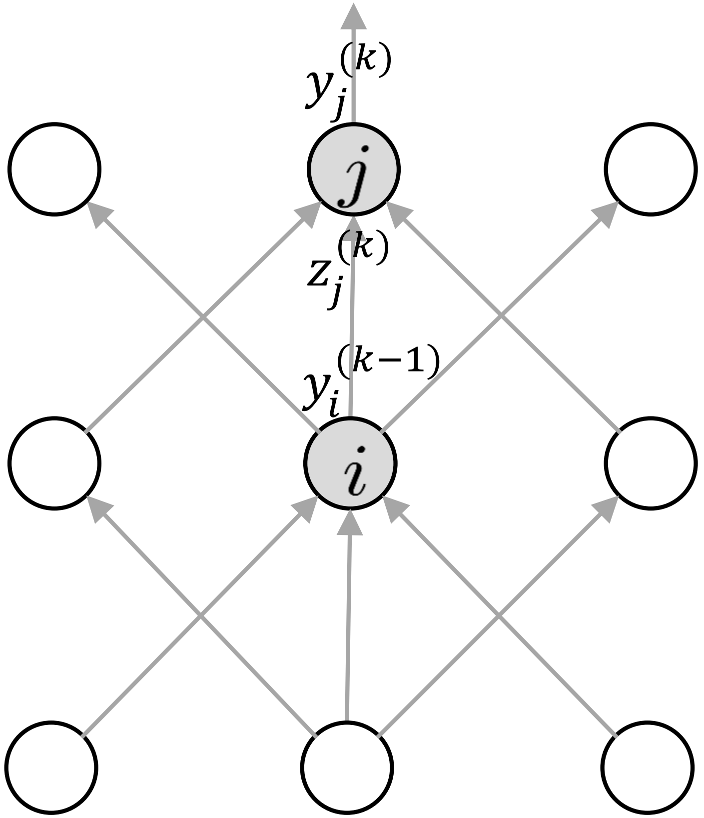
- Notation
  - $y_j^{(k)}$ Output of layer k
  - $z_j^{(k)}$ Input of layer k

Connections: $z_j^{(k)} = \sum w_{ji}^{(k-1)} y_i^{(k-1)}$

$$\boxed{\frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} = w_{ji}^{(k-1)}}$$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

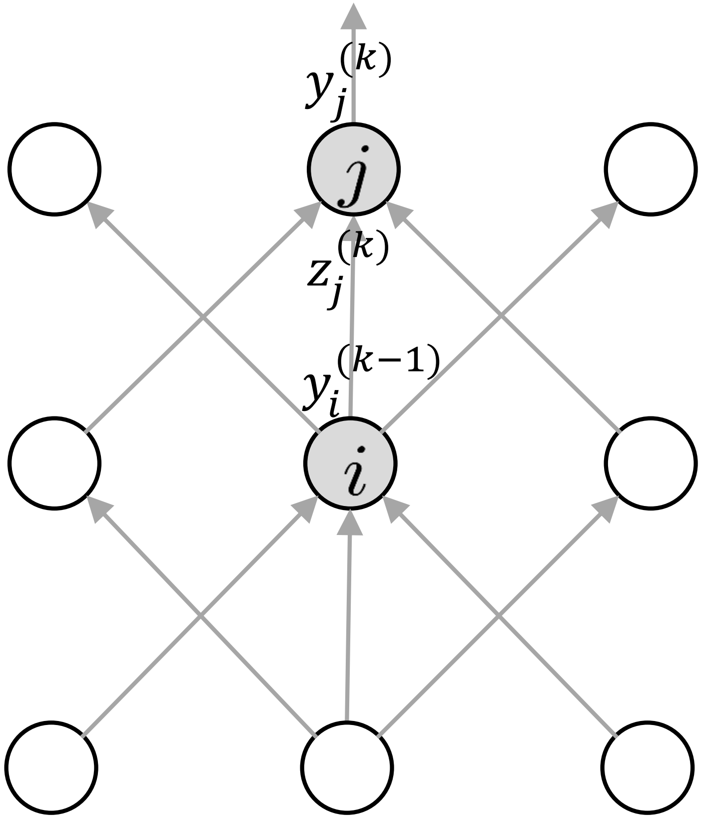
- Notation
  - $y_j^{(k)}$ Output of layer k
  - $z_j^{(k)}$ Input of layer k

Connections: $z_j^{(k)} = \sum w_{ji}^{(k-1)} y_i^{(k-1)}$

$$\boxed{\frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} = y_i^{(k-1)}}$$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j^{(k)}} = \frac{\partial y_j^{(k)}}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}} = \frac{\partial g\left(z_j^{(k)}\right)}{\partial z_j^{(k)}} \frac{\partial E}{\partial y_j^{(k)}}$$

$$\frac{\partial E}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = \sum_j w_{ji}^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

$$\frac{\partial E}{\partial w_{ji}^{(k-1)}} = \frac{\partial z_j^{(k)}}{\partial w_{ji}^{(k-1)}} \frac{\partial E}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial E}{\partial z_j^{(k)}}$$

- Efficient propagation scheme
  - $y_i^{(k-1)}$ is already known from forward pass! (Dynamic Programming)

$\Rightarrow$ Propagate back the gradient from layer k and multiply with $y_i^{(k-1)}$.

Slide adapted from Geoff Hinton

Summary: MLP Backpropagation

- Forward Pass

```
$\mathbf{y}^{(0)} = \mathbf{x}$
for $k = 1, ..., l$ do
    $\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{y}^{(k-1)}$
    $\mathbf{y}^{(k)} = g_k(\mathbf{z}^{(k)})$
endfor
$\mathbf{y} = \mathbf{y}^{(l)}$
$E = L(\mathbf{t}, \mathbf{y}) + \lambda\Omega(\mathbf{W})$
```

- Backward Pass

```
$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}} L(\mathbf{t}, \mathbf{y}) + \lambda \frac{\partial}{\partial \mathbf{y}} \Omega$
for $k = l, l\text{-}1, ..., 1$ do
    $\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{z}^{(k)}} = \mathbf{h} \odot g'(\mathbf{y}^{(k)})$
    $\frac{\partial E}{\partial \mathbf{W}^{(k)}} = \mathbf{h}\mathbf{y}^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial \mathbf{W}^{(k)}}$
    $\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}^{(k-1)}} = \mathbf{W}^{(k)\top}\mathbf{h}$
endfor
```

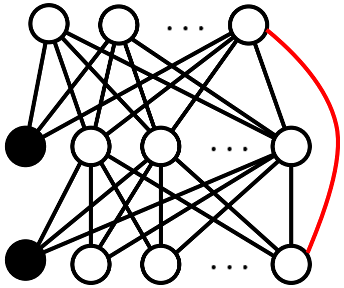
- Notes
  - For efficiency, an entire batch of data $\mathbf{X}$ is processed at once.
  - $\odot$ denotes the element-wise product

Analysis: Backpropagation

- Backpropagation is the key to make deep NNs tractable
  - However...

- The Backprop algorithm given here is specific to MLPs
  - It does not work with more complex architectures, e.g. skip connections or recurrent networks!
  - Whenever a new connection function induces a different functional form of the chain rule, you have to derive a new Backprop algorithm for it.

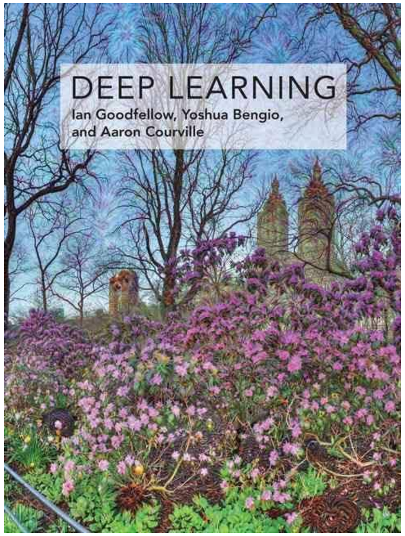
  $\Rightarrow$ Tedious...



- Let's analyze Backprop in more detail
  - This will lead us to a more flexible algorithm formulation
  - Next lecture…

References and Further Reading

- More information on Neural Networks can be found in Chapters 6 and 7 of the Goodfellow & Bengio book



I. Goodfellow, Y. Bengio, A. Courville
Deep Learning
MIT Press, 2016

https://goodfeli.github.io/dlbook/