Machine Learning – Lecture 12

Neural Networks

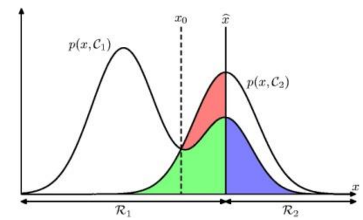
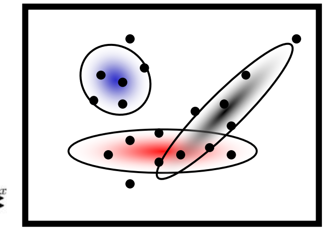
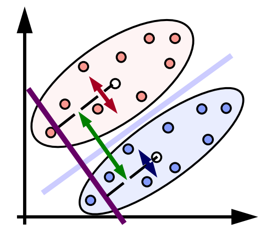
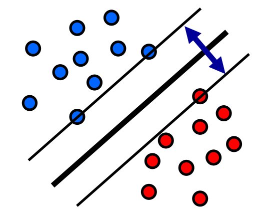
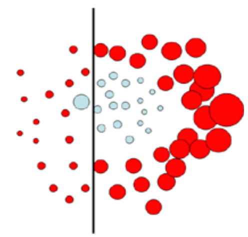
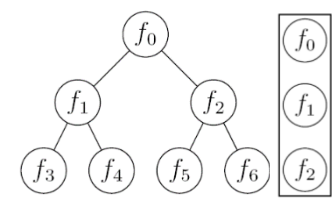
30.11.2017

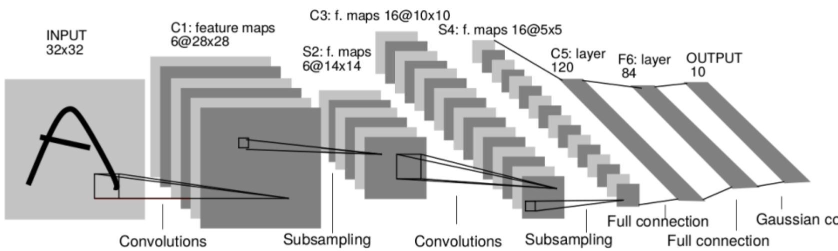
Bastian Leibe
RWTH Aachen
http://www.vision.rwth-aachen.de

leibe@vision.rwth-aachen.de

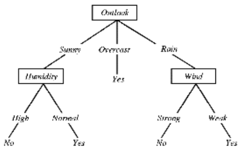
Course Outline

- 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

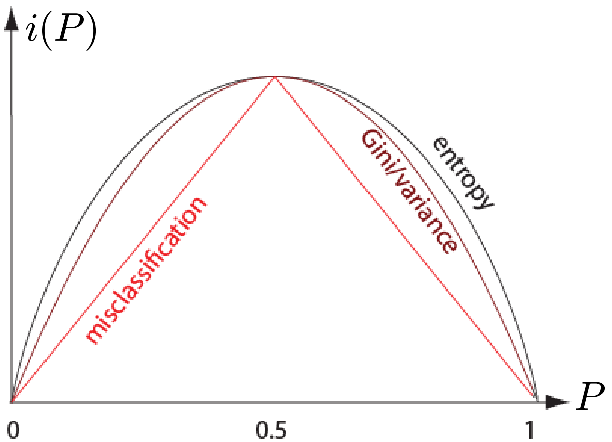


Recap: Decision Tree Training



- Goal
  - Select the query (=split) that decreases impurity the most

$$\Delta i(s_j) = i(s_j) - P_L i(s_{j,L}) - (1 - P_L) i(s_{j,R})$$



- Impurity measures
  - Entropy impurity (information gain):

$$i(s_j) = -\sum_k p(C_k|s_j) \log_2 p(C_k|s_j)$$

  - Gini impurity:

$$i(s_j) = \sum_{k \neq l} p(C_k|s_j)\, p(C_l|s_j) = \frac{1}{2}\left[1 - \sum_k p^2(C_k|s_j)\right]$$

Image source: R.O. Duda, P.E. Hart, D.G. Stork, 2001

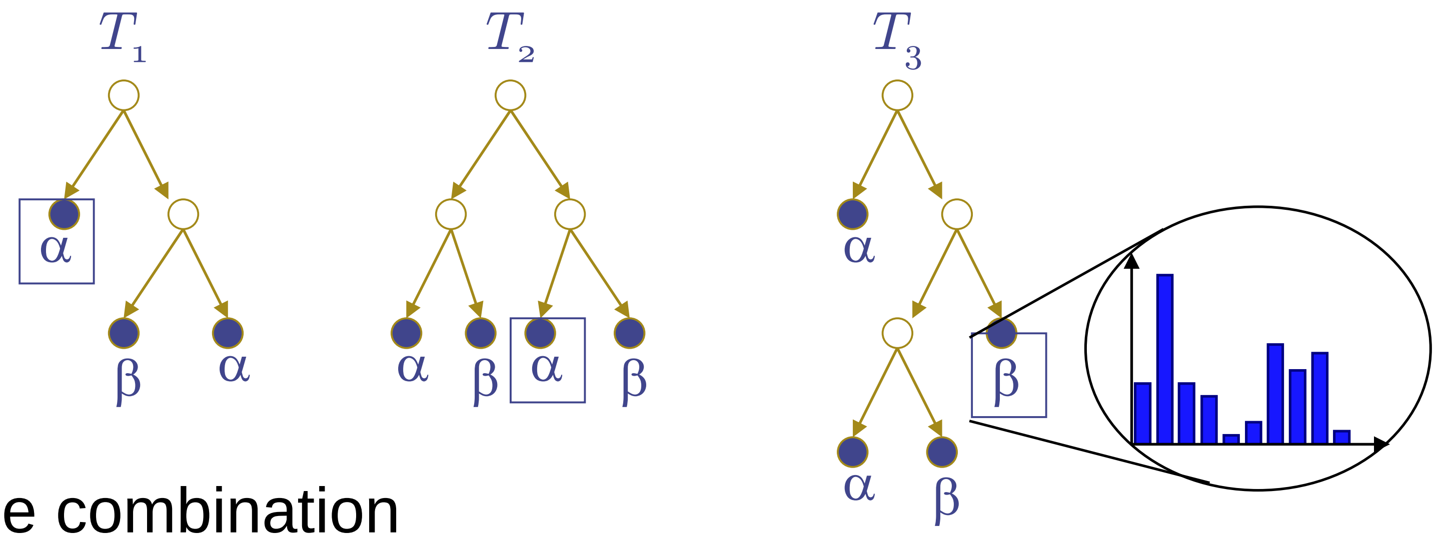
Recap: Randomized Decision Trees

- Decision trees: main effort on finding good split
  - Training runtime: $O(DN^2 \log N)$
  - This is what takes most effort in practice.
  - Especially cumbersome with many attributes (large D).

- Idea: randomize attribute selection
  - No longer look for globally optimal split.
  - Instead randomly use subset of K attributes on which to base the split.
  - Choose best splitting attribute e.g. by maximizing the information gain (= reducing entropy):

$$\triangle E = \sum_{k=1}^{K} \frac{|S_k|}{|S|} \sum_{j=1}^{N} p_j \log_2(p_j)$$

Recap: Ensemble Combination



- Ensemble combination
  - Tree leaves (l,η) store posterior probabilities of the target classes.

$$p_{l,\eta}(\mathcal{C}|\mathbf{x})$$

  - Combine the output of several trees by averaging their posteriors (Bayesian model combination)

$$p(\mathcal{C}|\mathbf{x}) = \frac{1}{L}\sum_{l=1}^{L} p_{l,\eta}(\mathcal{C}|\mathbf{x})$$

Recap: Random Forests (Breiman 2001)

- General ensemble method
  - Idea: Create ensemble of many (50 - 1,000) trees.
- Injecting randomness
  - Bootstrap sampling process
    - On average only 63% of training examples used for building the tree
    - Remaining 37% out-of-bag samples used for validation.
  - Random attribute selection
    - Randomly choose subset of K attributes to select from at each node.
    - Faster training procedure.
- Simple majority vote for tree combination
- Empirically very good results
  - Often as good as SVMs (and sometimes better)!
  - Often as good as Boosting (and sometimes better)!

Today's Topic

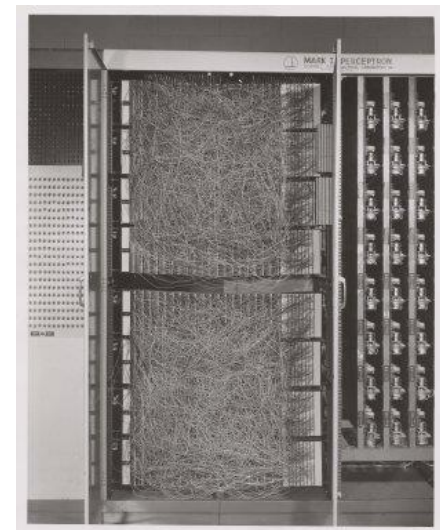
Topics of This Lecture

- 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.

Neural Networks don’t work!

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.

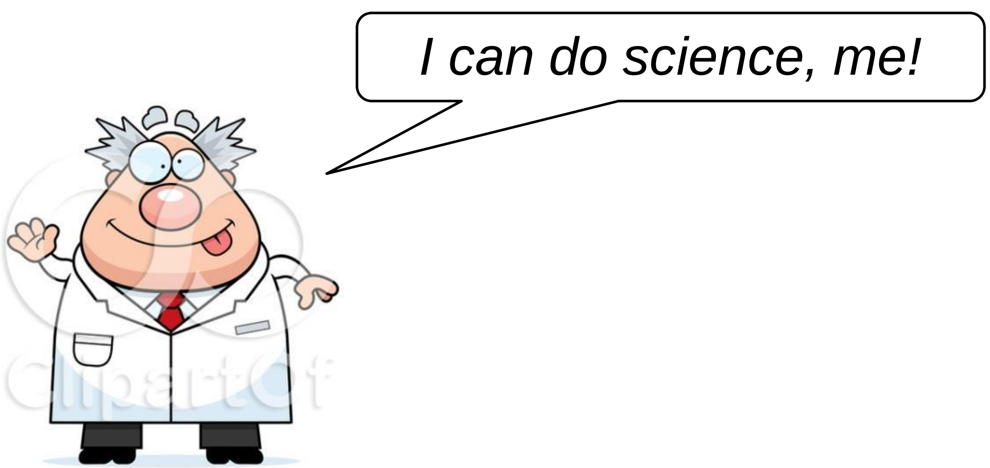


Image source: clipartof.com

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.

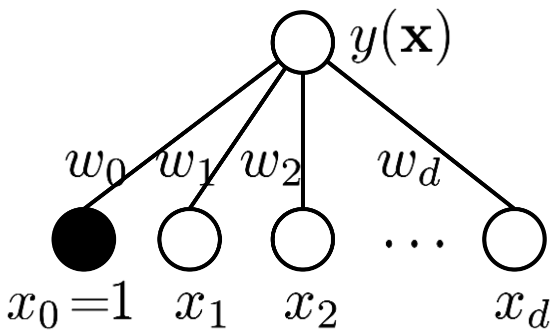
Image source: clipartpanda.com, clipartof.com

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

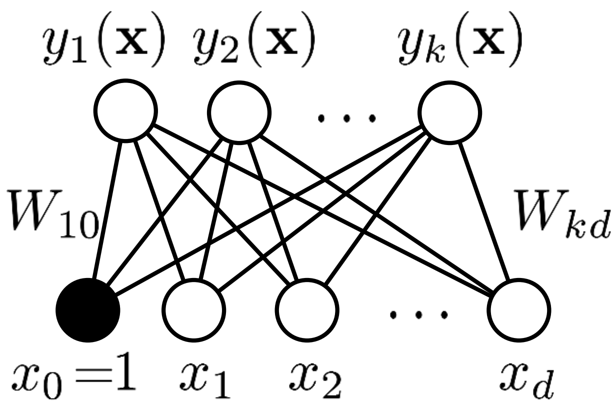
    $$y(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + w_0$$

    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

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

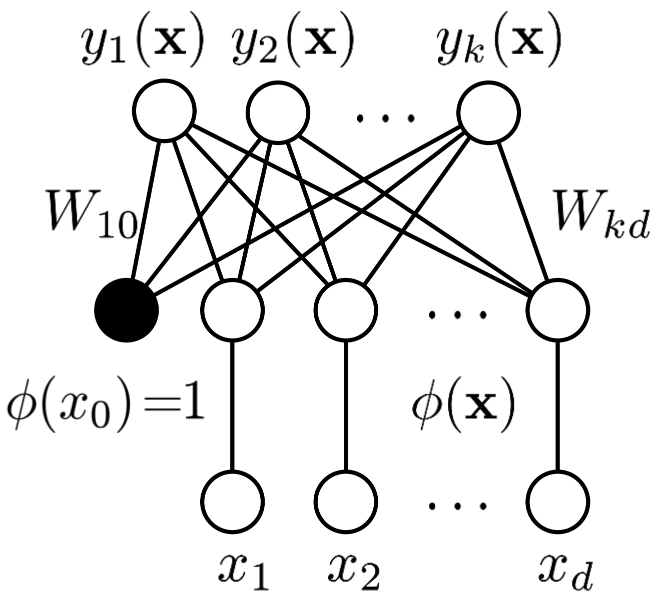
Logistic outputs

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

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

Extension: Non-Linear Basis Functions

- Straightforward generalization



Output layer

Weights

Feature layer

Mapping (fixed)

Input layer

- Outputs
  - Linear outputs

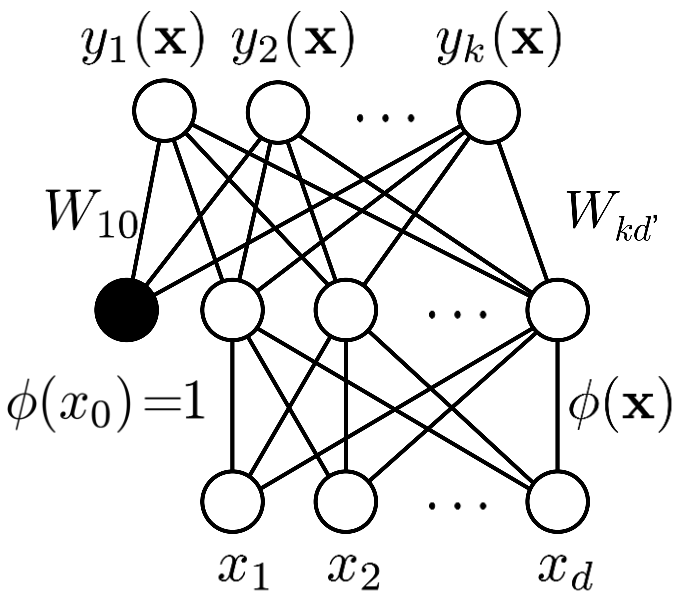
$$y_k(\mathbf{x}) = \sum_{i=0}^{d} W_{ki}\phi(x_i)$$

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!

$\Rightarrow$ Perceptron Learning corresponds to 1st-order (stochastic) Gradient Descent of a quadratic error function!

Loss Functions

- We can now also apply other loss functions
  - L2 loss $\Rightarrow$ Least-squares regression
    $$L(t, y(\mathbf{x})) = \sum_n \left(y(\mathbf{x}_n) - t_n\right)^2$$
  - L1 loss: $\Rightarrow$ Median regression
    $$L(t, y(\mathbf{x})) = \sum_n \left|y(\mathbf{x}_n) - t_n\right|$$
  - Cross-entropy loss $\Rightarrow$ Logistic regression
    $$L(t, y(\mathbf{x})) = -\sum_n \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$
  - Hinge loss $\Rightarrow$ SVM classification
    $$L(t, y(\mathbf{x})) = \sum_n [1 - t_n y(\mathbf{x}_n)]_+$$
  - Softmax loss $\Rightarrow$ Multi-class probabilistic classification
    $$L(t, y(\mathbf{x})) = -\sum_n \sum_k \left\{ \mathbb{I}\left(t_n = k\right) \ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))} \right\}$$

Regularization

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

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

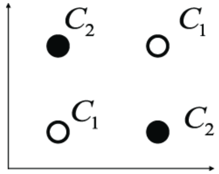
  - This is known as weight decay in Neural Networks.
  - We can also apply other regularizers, e.g. L1 $\Rightarrow$ sparsity
  - Since Neural Networks often have many parameters, regularization becomes very important in practice.
  - We will see more complex regularization techniques later on...

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!*

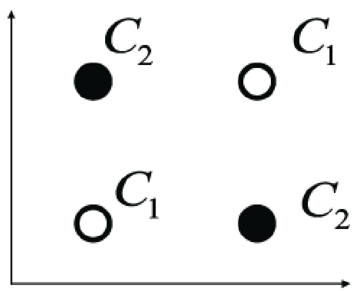
  - Once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.
    - Classic example: XOR function.



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 $\phi^\top\phi$.

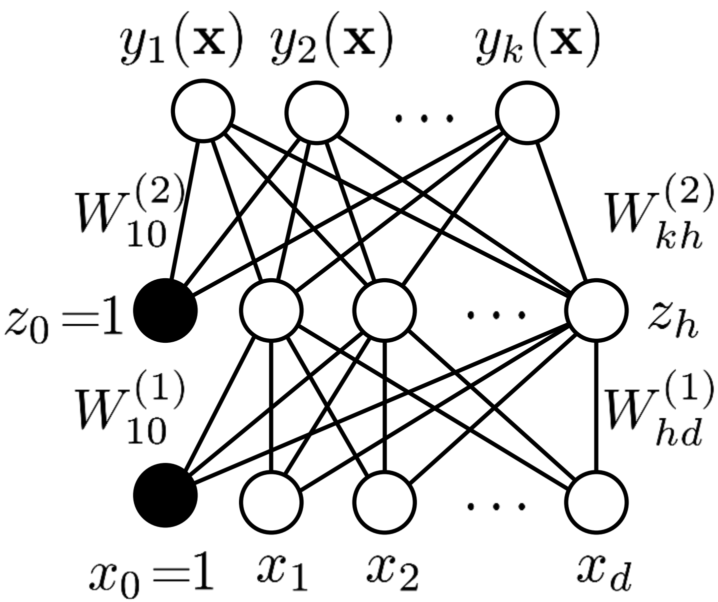
$\Rightarrow$ *Can we also learn such feature representations?*

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

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

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)

Learning with Hidden Units

- Networks without hidden units are very limited in what they can learn
  - More layers of linear units do not help $\Rightarrow$ 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.

  $\Rightarrow$ Main challenge in deep learning.

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_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

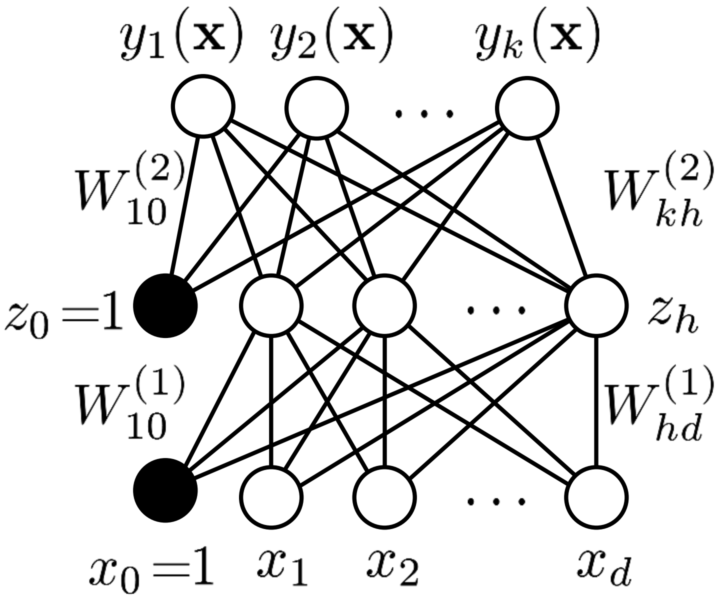
- Two main steps
  1. Computing the gradients for each weight — today
  2. Adjusting the weights in the direction of the gradient — next lecture

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



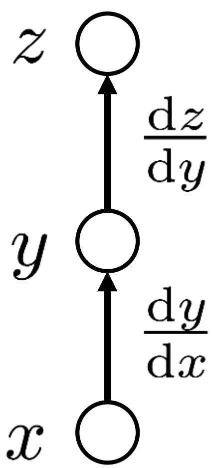
$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

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

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

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions



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

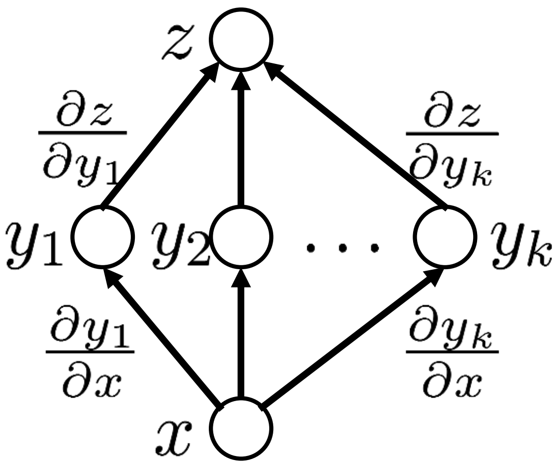
$$\Delta y = \frac{\mathrm{d}y}{\mathrm{d}x} \Delta x$$

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

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

Excursion: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative



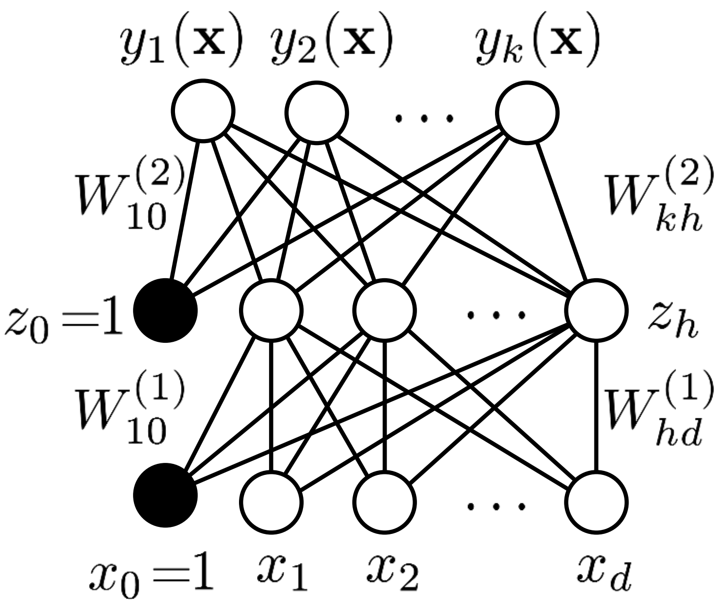
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \dots$$

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

$\Rightarrow$ 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!

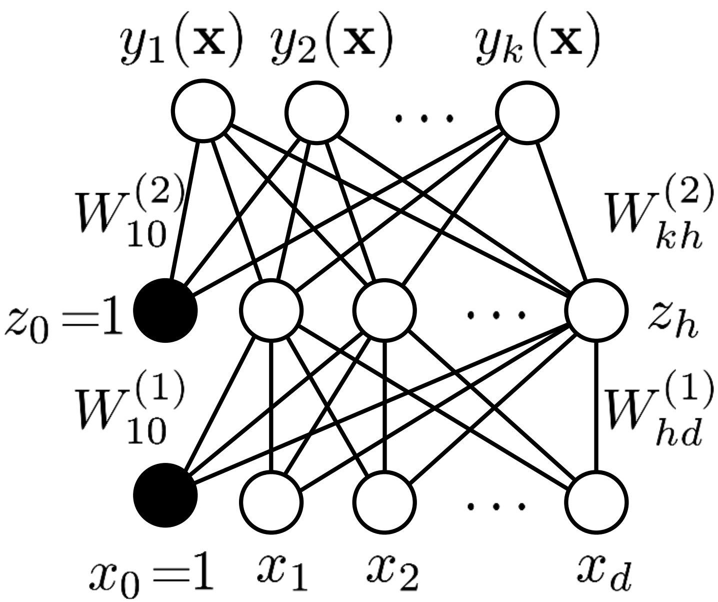
$\Rightarrow$ Infeasible to compute this way.

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 $E(\mathbf{W}^{(\tau)})$.

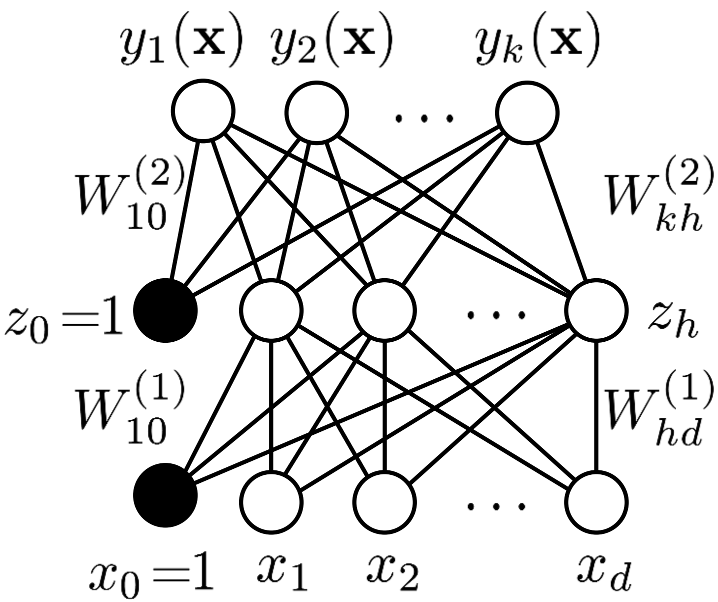
$\Rightarrow$ Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

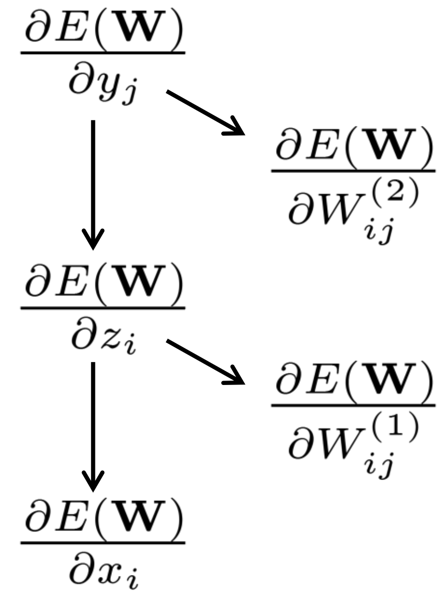
Topics of This Lecture

- 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





  - 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.

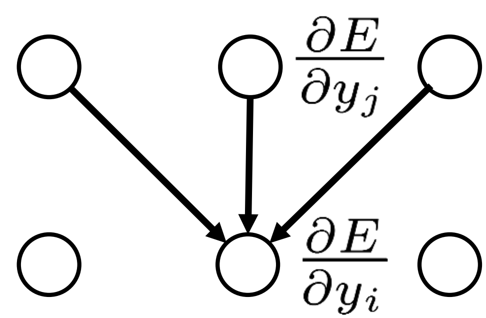
$\Rightarrow$ Backpropagation algorithm

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

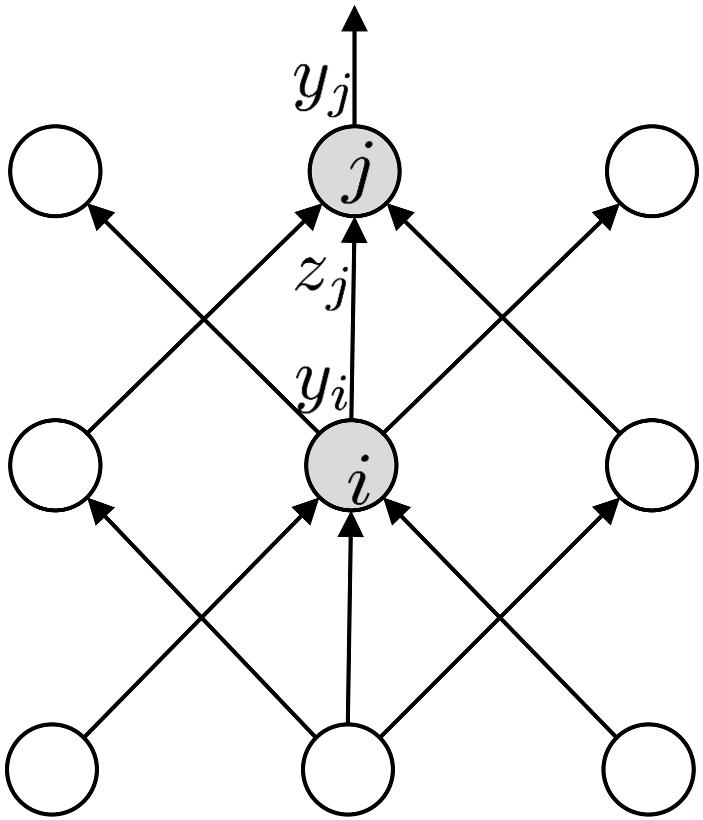
$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

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



$$\frac{\partial E}{\partial y_j} \longrightarrow \frac{\partial E}{\partial w_{ik}}$$

Backpropagation Algorithm



E.g. with sigmoid output nonlinearity

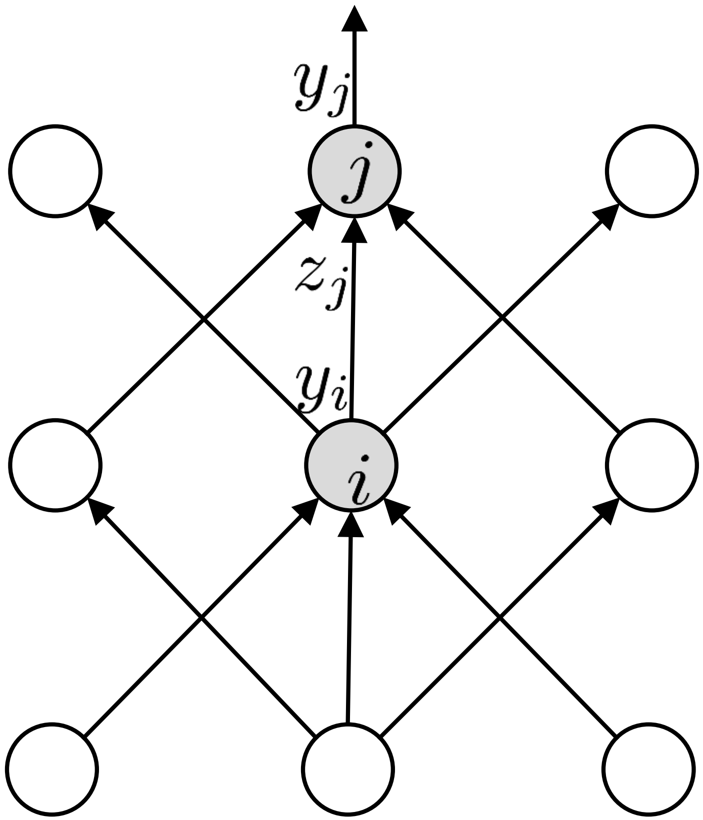
$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

- Notation
  - y_j Output of layer j
  - z_j Input of layer j

Connections: $z_j = \sum_i w_{ij} y_i$

$$y_j = g(z_j)$$

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

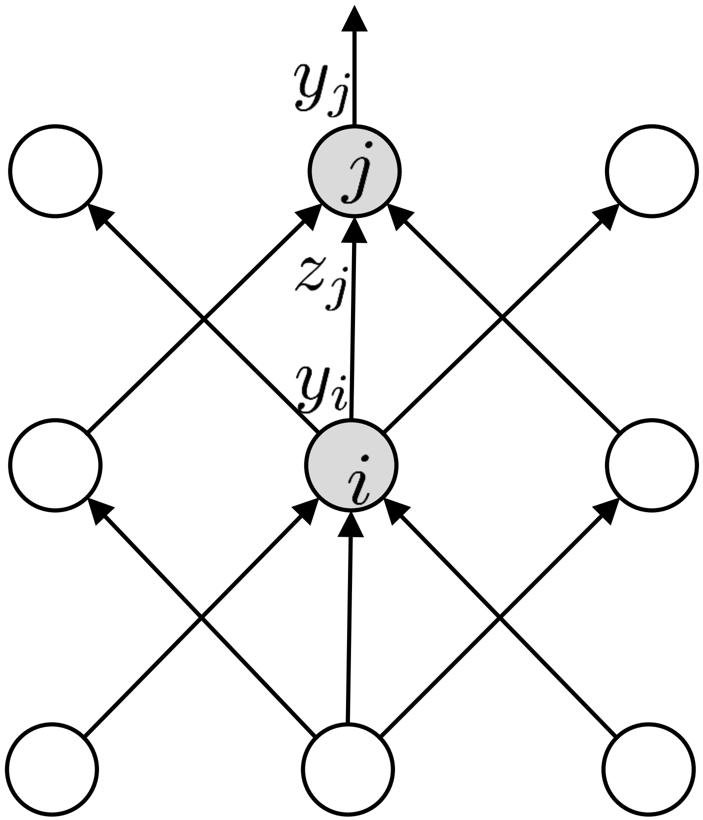
$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

- Notation
  - y_j Output of layer j
  - z_j Input of layer j

Connections: $z_j = \sum_i w_{ij} y_i$

$$\frac{\partial z_j}{\partial y_i} = w_{ij}$$

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

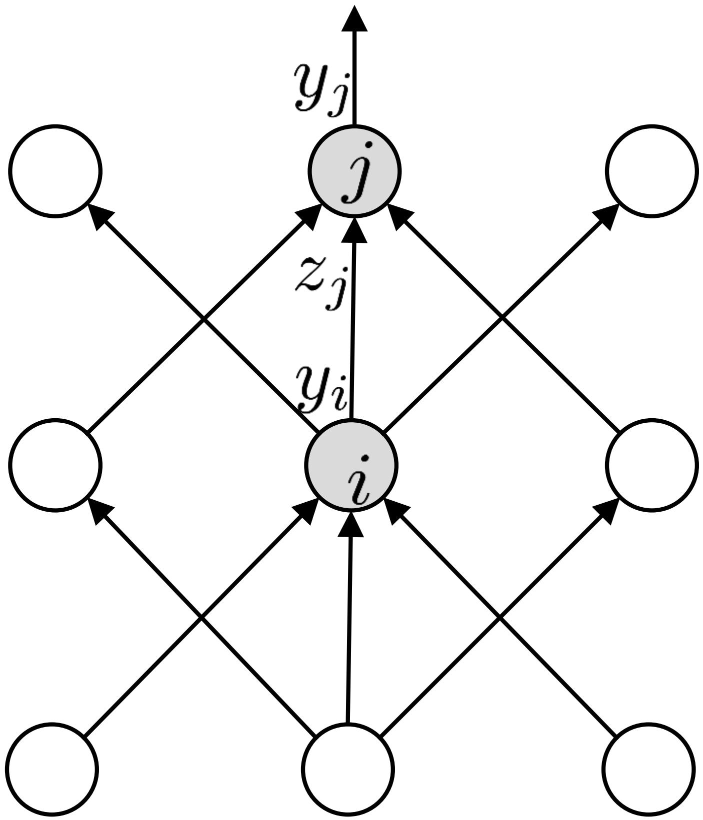
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

- Notation
  - y_j Output of layer j
  - z_j Input of layer j

Connections: $z_j = \sum_i w_{ij} y_i$

$$\frac{\partial z_j}{\partial w_{ij}} = y_i$$

Backpropagation Algorithm



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

- Efficient propagation scheme
  - y_i is already known from forward pass! (Dynamic Programming)

  $\Rightarrow$ Propagate back the gradient from layer j and multiply with y_i.

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

$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{z}^{(k)}} = \mathbf{h} \odot g'(\mathbf{y}^{(k)})$$

$$\frac{\partial E}{\partial \mathbf{W}^{(k)}} = \mathbf{h}\mathbf{y}^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial \mathbf{W}^{(k)}}$$

$$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}^{(k-1)}} = \mathbf{W}^{(k)\top}\mathbf{h}$$

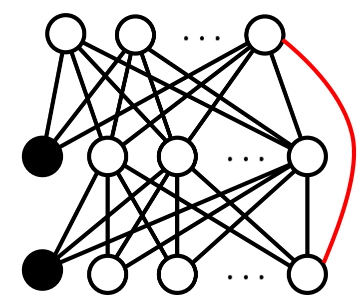
endfor

- 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.

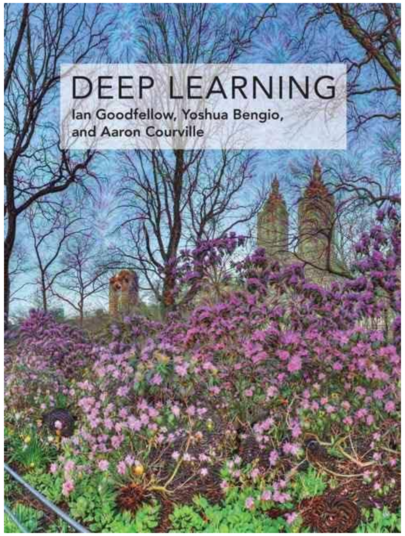


$\Rightarrow$ Tedious...

- 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/