Advanced Machine Learning
Lecture 13

Backpropagation

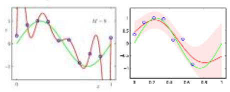
14.12.2015

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

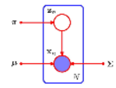
leibe@vision.rwth-aachen.de

This Lecture: *Advanced Machine Learning*

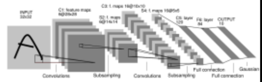
- Regression Approaches $f : \mathcal{X} \rightarrow \mathbb{R}$
  - Linear Regression
  - Regularization (Ridge, Lasso)
  - Gaussian Processes



- Learning with Latent Variables
  - Prob. Distributions & Approx. Inference
  - Mixture Models
  - EM and Generalizations

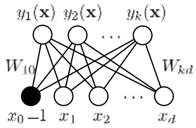


- Deep Learning
  - Linear Discriminants
  - Neural Networks
  - Backpropagation
  - CNNs, RNNs, RBMs, etc.



Recap: Perceptrons

- One output node per class



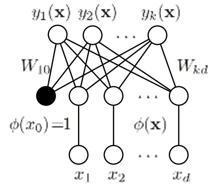
Output layer
Weights
Input layer

- Outputs
  - Linear outputs $y_k(\mathbf{x}) = \sum_{i=0}^{d} W_{ki} x_i$
  - With output nonlinearity $y_k(\mathbf{x}) = g\left(\sum_{i=0}^{d} W_{ki} x_i\right)$

⇒ Can be used to do multidimensional linear regression or multiclass classification.

Recap: Non-Linear Basis Functions

- Straightforward generalization

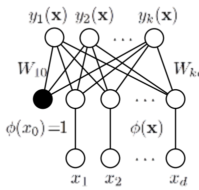


Output layer
Weights
Feature layer
Mapping (fixed)
Input layer

- Outputs
  - Linear outputs $y_k(\mathbf{x}) = \sum_{i=0}^{d} W_{ki} \phi(x_i)$
  - with output nonlinearity $y_k(\mathbf{x}) = g\left(\sum_{i=0}^{d} W_{ki} \phi(x_i)\right)$

Recap: 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!

Recap: Perceptron Learning

- 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!

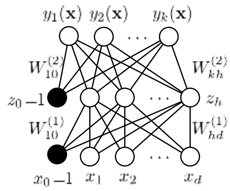
⇒ Perceptron Learning corresponds to 1st-order (stochastic) Gradient Descent of a quadratic error function!

Recap: Loss Functions

- We can now also apply other loss functions
  - L_2 loss ⇒ Least-squares regression
    $L(t, y(\mathbf{x})) = \sum_n (y(\mathbf{x}_n) - t_n)^2$
  - L_1 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
    $L(t, y(\mathbf{x})) = -\sum_n \sum_k \left\{ \mathbb{I}(t_n = k) \ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))} \right\}$

Recap: Multi-Layer Perceptrons

- Adding more layers



Output layer

Hidden layer

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

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numeric differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- Practical Issues
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

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., $L(t, y(\mathbf{x}; \mathbf{W})) = \sum_n (y(\mathbf{x}_n; \mathbf{W}) - t_n)^2$ — L_2 loss
    $\Omega(\mathbf{W}) = ||\mathbf{W}||_F^2$ — L_2 regularizer ("weight decay")

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

Gradient Descent

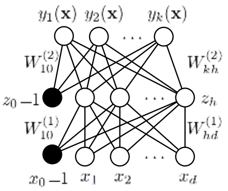
- Two main steps
  1. Computing the gradients for each weight — today
  2. Adjusting the weights in the direction of the gradient — Thursday

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numeric differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- Practical Issues
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



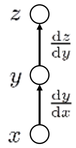
$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \quad \cdots \quad \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

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

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

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions



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

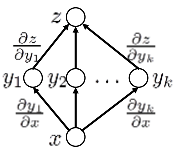
$$\Delta y = \frac{\mathrm{d}y}{\mathrm{d}x}\Delta x$$

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

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

Excursion: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative



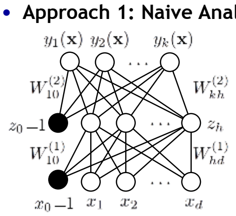
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \ldots$$

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

⇒ 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?*

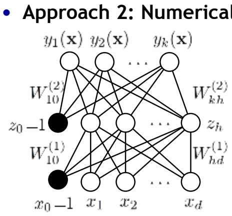
⇒ With increasing depth, there will be exponentially many paths!
⇒ Infeasible to compute this way.

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- Practical Issues
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

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

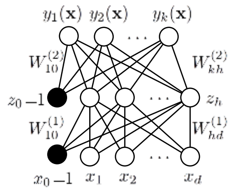
⇒ Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

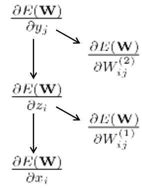
Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- Practical Issues
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

Obtaining the Gradients

- Approach 3: Incremental Analytical Differentiation



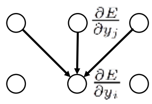


- Idea: Compute the gradients layer by layer.
- Each layer below builds upon the results of the layer above.

⇒ The gradient is propagated backwards through the layers.
⇒ Backpropagation algorithm

Backpropagation Algorithm

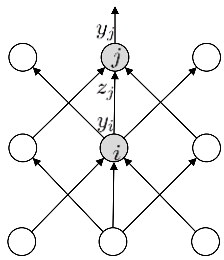
- Core steps
  1. Convert the discrepancy between each output and its target value into an error derivate.
     $$E = \frac{1}{2}\sum_{j \in output}(t_j - y_j)^2$$
     $$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$
  2. Compute error derivatives in each hidden layer from error derivatives in the layer above.



  3. Use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights
     $$\frac{\partial E}{\partial y_j} \longrightarrow \frac{\partial E}{\partial w_{ik}}$$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



E.g. with sigmoid output nonlinearity

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

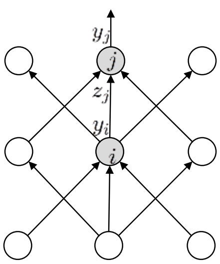
- Notation
  - y_j Output of layer j
  - z_j Input of layer j

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

$y_j = g(z_j)$

Slide adapted from Geoff Hinton

Backpropagation Algorithm



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

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

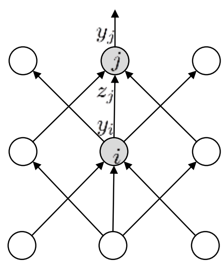
- Notation
  - y_j Output of layer j
  - z_j Input of layer j

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

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

Slide adapted from Geoff Hinton

Backpropagation Algorithm



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

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

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

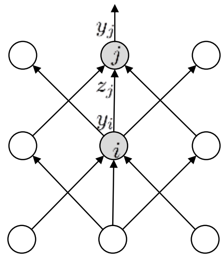
- Notation
  - y_j Output of layer j
  - z_j Input of layer j

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

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

Slide adapted from Geoff Hinton

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)
  - ⇒ 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
$\mathbf{y} = \mathbf{y}^{(l)}$
$E = L(\mathbf{t}, \mathbf{y}) + \lambda\Omega(\mathbf{W})$

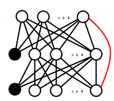
- **Backward Pass**

$\mathbf{h} \leftarrow \frac{\partial E}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}}L(\mathbf{t},\mathbf{y}) + \lambda\frac{\partial}{\partial \mathbf{y}}\Omega$
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.
  - ⇒ Tedious...



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

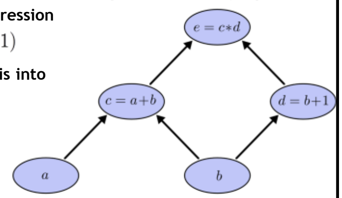
Computational Graphs

- **We can think of mathematical expressions as graphs**
  - E.g., consider the expression

$$e = (a+b)*(b+1)$$

  - We can decompose this into the operations

$$c = a+b$$
$$d = b+1$$
$$e = c*d$$

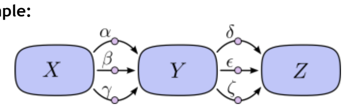


and visualize this as a computational graph.

- **Evaluating partial derivatives $\frac{\partial Y}{\partial X}$ in such a graph**
  - General rule: sum over all possible paths from Y to X and multiply the derivatives on each edge of the path together.

Factoring Paths

- **Problem: Combinatorial explosion**
  - Example:



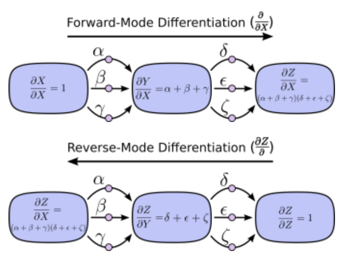
  - There are 3 paths from X to Y and 3 more from Y to Z.
  - If we want to compute $\frac{\partial Z}{\partial X}$, we need to sum over 3×3 paths:

$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

  - Instead of naively summing over paths, it's better to factor them

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma) * (\delta + \epsilon + \zeta)$$

Efficient Factored Algorithms



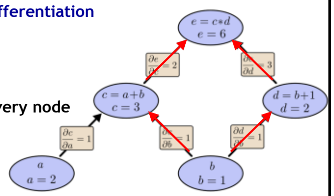
Apply operator $\frac{\partial}{\partial X}$ to every node.

Apply operator $\frac{\partial Z}{\partial}$ to every node.

- **Efficient algorithms for computing the sum**
  - Instead of summing over all of the paths explicitly, compute the sum more efficiently by merging paths back together at every node.

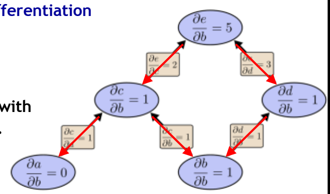
Why Do We Care?

- Let's consider the example again
  - Using forward-mode differentiation from b up...
  - Runtime: $\mathcal{O}$(#edges)
  - Result: derivative of every node with respect to b.



Why Do We Care?

- Let's consider the example again
  - Using reverse-mode differentiation from e down...
  - Runtime: $\mathcal{O}$(#edges)
  - Result: derivative of e with respect to every node.



⇒ *This is what we want to compute in Backpropagation!*
- Forward differentiation needs one pass per node. With backward differentiation can compute all derivatives in one single pass.

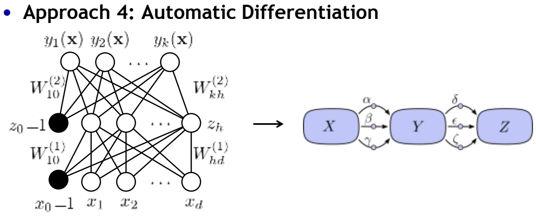
⇒ Speed-up in $\mathcal{O}$(#inputs) compared to forward differentiation!

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- Practical Issues
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

Obtaining the Gradients

- Approach 4: Automatic Differentiation



  - Convert the network into a computational graph.
  - Each new layer/module just needs to specify how it affects the forward and backward passes.
  - Apply reverse-mode differentiation.

⇒ Very general algorithm, used in today's Deep Learning packages

Modular Implementation (e.g., Torch)

- Solution in many current Deep Learning libraries
  - Provide a limited form of automatic differentiation
  - Restricted to "programs" composed of "modules" with a predefined set of operations.
- Each module is defined by two main functions
  1. Computing the outputs $\mathbf{y}$ of the module given its inputs $\mathbf{x}$

     $$\mathbf{y} = \text{module}.\mathbf{fprop}(\mathbf{x})$$

     where $\mathbf{x}$, $\mathbf{y}$, and intermediate results are stored in the module.
  2. Computing the gradient $\partial E / \partial \mathbf{x}$ of a scalar cost w.r.t. the inputs $\mathbf{x}$ given the gradient $\partial E / \partial \mathbf{y}$ w.r.t. the outputs $\mathbf{y}$

     $$\frac{\partial E}{\partial \mathbf{x}} = \text{module}.\mathbf{bprop}\left(\frac{\partial E}{\partial \mathbf{y}}\right)$$

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- Practical Issues
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly
  - Efficient batch processing

Commonly Used Nonlinearities

- **Sigmoid**

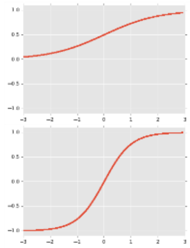
$$g(a) = \sigma(a) = \frac{1}{1+\exp\{-a\}}$$

- **Hyperbolic tangent**

$$g(a) = tanh(a) = 2\sigma(2a) - 1$$

- **Softmax**

$$g(\mathbf{a}) = \frac{\exp\{-a_i\}}{\sum_j \exp\{-a_j\}}$$



Commonly Used Nonlinearities (2)

- **Hard tanh**

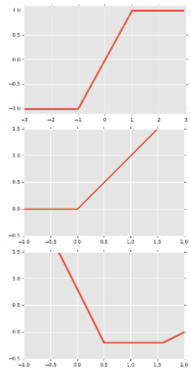
$$g(a) = \max\{-1, \min\{1, a\}\}$$

- **Rectified linear unit (ReLU)**

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

- **Maxout**

$$g(\mathbf{a}) = \max_i \{\mathbf{w}_i^\top \mathbf{a} + b_i\}$$



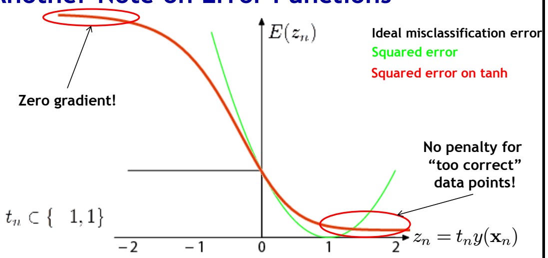
Usage

- **Output nodes**
  - Typically, a sigmoid or tanh function is used here.
    - Sigmoid for nice probabilistic interpretation (range [0,1]).
    - tanh for regression tasks
- **Internal nodes**
  - Historically, tanh was most often used.
  - tanh is better than sigmoid for internal nodes, since it is already centered.
  - Internally, tanh is often implemented as piecewise linear function (similar to hard tanh and maxout).
  - More recently: ReLU often used for classification tasks.

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numeric differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- **Practical Issues**
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

Another Note on Error Functions



- **Squared error on sigmoid/tanh output function**
  - Avoids penalizing "too correct" data points.
  - But: zero gradient for confidently incorrect classifications!
  - ⇒ Do not use L_2 loss with sigmoid outputs (instead: cross-entropy)!

Image source: Bishop, 2006

Topics of This Lecture

- Learning with Hidden Units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation
- **Practical Issues**
  - Nonlinearities
  - Sigmoid outputs and the L_2 loss
  - Implementing Softmax correctly

Implementing Softmax Correctly

- Softmax output
  - De-facto standard for multi-class outputs

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\left\{\mathbb{I}\left(t_n = k\right)\ln\frac{\exp(\mathbf{w}_k^\top \mathbf{x})}{\sum_{j=1}^{K}\exp(\mathbf{w}_j^\top \mathbf{x})}\right\}$$

- Practical issue
  - Exponentials get very big and can have vastly different magnitudes.
  - Trick 1: Do not compute first softmax, then log, but instead directly evaluate log-exp in the denominator.
  - Trick 2: Softmax has the property that for a fixed vector $\mathbf{b}$

$$\mathrm{softmax}(\mathbf{a}+\mathbf{b}) = \mathrm{softmax}(\mathbf{a})$$

⇒ Subtract the largest weight vector $\mathbf{w}_j$ from the others.

References and Further Reading

- More information on Backpropagation can be found in Chapter 6 of the Goodfellow & Bengio book

Ian Goodfellow, Aaron Courville, Yoshua Bengio
Deep Learning
MIT Press, in preparation

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