Recap: Perceptrons
- One output node per class
- Outputs
  - Linear outputs
  - With output nonlinearity
  - Can be used to do multidimensional linear regression or multiclass classification.

Recap: Non-Linear Basis Functions
- Straightforward generalization
- Outputs
  - Linear outputs
  - With output nonlinearity

Remarks
- Perceptrons are generalized linear discriminants!
- Everything we know about the latter can also be applied here.
- Note: feature functions \( \phi(x) \) are kept fixed, not learned!
Recap: Loss Functions
- We can now also apply other loss functions
  - L₂ loss: \( L(t, y(x)) = \sum_n (y(x_n) - t_n)^2 \) ⇒ Least-squares regression
  - L₁ loss: \( L(t, y(x)) = \sum_n |y(x_n) - t_n| \) ⇒ Median regression
  - Cross-entropy loss:
  - Hinge loss
  - Softmax loss
  \[ L(t, y(x)) = -\sum_n \{ t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \} \]

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 L₂ 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
    \[ E(W) = \sum_n L(t_n, y(x_n; W)) + \Omega(W) \]
    with a loss \( L(\cdot) \) and a regularizer \( \Omega(\cdot) \).
  - E.g., \( L(t, y(x; W)) = \sum_n (y(x_n; W) - t_n)^2 \) L₂ loss
    \[ \Omega(W) = \|W\|^2 \] L₂ regularizer ("weight decay")
  ⇒ Update each weight \( W_{ij}^{(k)} \) in the direction of the gradient
    \[ \frac{\partial E(W)}{\partial W_{ij}^{(k)}} \]

Gradient Descent
- Two main steps
  1. Computing the gradients for each weight
  2. Adjusting the weights in the direction of the gradient

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 L₂ loss
  - Implementing Softmax correctly
Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation
  - Compute the gradients for each variable analytically.
  - What is the problem when doing this?

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions
  \[ z = \frac{dz}{dy} \quad \Delta z = \frac{dz}{dy} \Delta y \]
  \[ y = \frac{dy}{dx} \quad \Delta y = \frac{dy}{dx} \Delta x \]
  \[ \Delta z = \frac{dz}{dy} \frac{dy}{dx} \Delta x \]
  \[ \frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx} \frac{dz}{dx} \]

Excursion: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative
  \[ \frac{dz}{dx} = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x} + \ldots \]
  \[ = \sum_{i=1}^{k} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x} \]

⇒ Need to sum over all paths that lead to the target variable \( x \).

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 \( W^{(0)} \), we can evaluate \( E(W^{(0)}) \).
  - Idea: Make small changes to \( W^{(0)} \) and accept those that improve \( E(W^{(0)}) \).
  ⇒ 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 derivative.
  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.g. with sigmoid output nonlinearity

Notation

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

Connections:

$y_j = g(z_j)$

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

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

$\frac{\partial E}{\partial y_j} = \sum_j \frac{\partial z_j}{\partial y_j} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$
Backpropagation Algorithm

- 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
  \[ y^{(i)} = x \]
  \[ \text{for } k = 1, \ldots, l \text{ do} \]
  \[ z^{(k)} = W^{(k)} y^{(k-1)} \]
  \[ y^{(k)} = g_k(z^{(k)}) \]
  \[ \text{endfor} \]
  \[ y = y^{(l)} \]
  \[ E = L(t, y) + \lambda \Omega(W) \]

- Backward Pass
  \[ h \leftarrow \frac{\partial E}{\partial y} = \frac{\partial E}{\partial y} = \frac{\partial E}{\partial y} \]
  \[ \text{for } k = l, l-1, \ldots, 1 \text{ do} \]
  \[ h \leftarrow \frac{\partial E}{\partial y^{(k)}} = h \odot g'(y^{(k)}) \]
  \[ \frac{\partial E}{\partial W^{(k)}} = h y^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial W^{(k)}} \]
  \[ h \leftarrow \frac{\partial E}{\partial y^{(k-1)}} = W^{(k)\top} h \]
  \[ \text{endfor} \]

- Notes
  - For efficiency, an entire batch of data $X$ 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 $e = (a + b) \times (b + 1)$

- We can decompose this into the operations
  - $c = a + b$
  - $d = b + 1$
  - $e = c \times d$

and visualize this as a computational graph.

- Evaluating partial derivatives $\frac{\partial E}{\partial X}$ 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 $\times$ 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

  \[ \frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma) \ast (\delta + \epsilon + \zeta) \]

Efficient Factored Algorithms

- Forward-Mode Differentiation ($\frac{\partial E}{\partial X}$)
  - $\frac{\partial E}{\partial X} = 1$
  - $\frac{\partial E}{\partial X} = \frac{\partial E}{\partial X} + \frac{\partial E}{\partial X} \odot \frac{\partial E}{\partial X}$

- Reverse-Mode Differentiation ($\frac{\partial E}{\partial \theta}$)
  - $\frac{\partial E}{\partial \theta} = 1$
  - $\frac{\partial E}{\partial \theta} = \frac{\partial E}{\partial \theta} \odot \frac{\partial E}{\partial \theta}$

- 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: $O(\#\text{edges})$
  - Result: derivative of every node with respect to b.

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

⇒ This is what we want to compute in Backpropagation!
- Forward differentiation needs one pass per node. With backward differentiation we can compute all derivatives in one single pass.
⇒ Speed-up in $O(\#\text{inputs})$ 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

1. Computing the outputs $y$ of the module given its inputs $x$
   $$ y = \text{module.funprop}(x) $$
   where $x$, $y$, and intermediate results are stored in the module.

2. Computing the gradient $\frac{\partial E}{\partial x}$ of a scalar cost w.r.t. the inputs $x$ given the gradient $\frac{\partial E}{\partial y}$ w.r.t. the outputs $y$
   $$ \frac{\partial E}{\partial x} = \text{module.bprop}(\frac{\partial E}{\partial y}) $$

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 $y$ of the module given its inputs $x$
     $$ y = \text{module.funprop}(x) $$
     where $x$, $y$, and intermediate results are stored in the module.
  2. Computing the gradient $\frac{\partial E}{\partial x}$ of a scalar cost w.r.t. the inputs $x$ given the gradient $\frac{\partial E}{\partial y}$ w.r.t. the outputs $y$
     $$ \frac{\partial E}{\partial x} = \text{module.bprop}(\frac{\partial E}{\partial y}) $$

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
  \[ g(a) = \frac{\exp[-a_i]}{\sum_j \exp[-a_j]} \]

Commonly Used Nonlinearities (2)

• Hard tanh
  \[ g(a) = \max\{-1, \min\{1, a\}\} \]

• Rectified linear unit (ReLU)
  \[ g(a) = \max\{0, a\} \]

• Maxout
  \[ g(a) = \max\{w_i^T a + b_i\} \]

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
Implementing Softmax Correctly

- Softmax output
  - De-facto standard for multi-class outputs
  \[
  E(w) = - \sum_{n=1}^{N} \sum_{k=1}^{K} \left\{ \xi \left( t_n = k \right) \ln \frac{\exp(w_k x)}{\sum_{j=1}^{K} \exp(w_j x)} \right\}
  \]

- 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 \( b \)
    \[
    \text{softmax}(a + b) = \text{softmax}(a)
    \]
  - Subtract the largest weight vector \( w \) 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/