Machine Learning – Lecture 12

Deep Learning

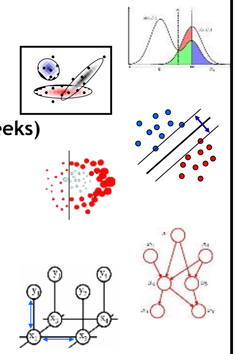
14.06.2016

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

leibe@vision.rwth-aachen.de

Course Outline

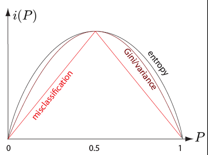
- Fundamentals (2 weeks)
  - Bayes Decision Theory
  - Probability Density Estimation
- Discriminative Approaches (5 weeks)
  - Linear Discriminant Functions
  - Statistical Learning Theory & SVMs
  - Ensemble Methods & Boosting
  - Randomized Trees, Forests & Ferns
  - Deep Learning
- Generative Models (4 weeks)
  - Bayesian Networks
  - Markov Random Fields



Recap: Decision Tree Training

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

$$\triangle i(N) = i(N) - P_L i(N_L) - (1 - P_L) i(N_R)$$



- Impurity measures
  - Entropy impurity (information gain):

$$i(N) = -\sum_j p(\mathcal{C}_j|N)\log_2 p(\mathcal{C}_j|N)$$

  - Gini impurity:

$$i(N) = \sum_{i \neq j} p(\mathcal{C}_i|N)p(\mathcal{C}_j|N) = \frac{1}{2}\left[1 - \sum_j p^2(\mathcal{C}_j|N)\right]$$

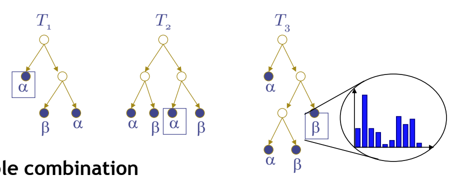
Image source: R.O. Duda, P.E. Hart, D.G. Stork, 2001

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)

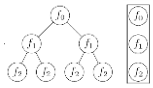
$$p(\mathcal{C}|\mathbf{x}) = \frac{1}{L}\sum_{l=1}^{L} p_{l,\eta}(\mathcal{C}|\mathbf{x})$$

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)!

Recap: Ferns

- Ferns
  - Ferns are semi-naïve Bayes classifiers.
  - They assume independence between sets of features (between the ferns)…
  - …and enumerate all possible outcomes inside each set.



- Interpretation
  - Combine the tests $f_l,\ldots,f_{l+S}$ into a binary number.
  - Update the "fern leaf" corresponding to that number.

0
0
1

Update leaf 100_2 = 4

B. Leibe

8

Machine Learning, Summer '16

Recap: Ferns (Semi-Naïve Bayes Classifiers)

- Ferns
  - A fern F is defined as a set of S binary features $\{f_l,\ldots,f_{l+S}\}$.
  - M: number of ferns, $N_f = S \cdot M$.
  - This represents a compromise:

$$p(f_1,\ldots,f_{N_f}|\mathcal{C}_k) \approx \prod_{j=1}^{M} p(F_j|\mathcal{C}_k)$$

$$= \underbrace{p(f_1,\ldots,f_S|\mathcal{C}_k)}_{\text{Full joint inside fern}} \cdot p(f_{S+1},\ldots,f_{2S}|\mathcal{C}_k) \cdot \ldots$$

Naïve Bayes between ferns

⇒ Model with $M \cdot 2^S$ parameters ("Semi-Naïve").

⇒ Flexible solution that allows complexity/performance tuning.

B. Leibe

9

Machine Learning, Summer '16

Today's Topic

B. Leibe

10

Machine Learning, Summer '16

Topics of This Lecture

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

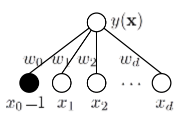
B. Leibe

11

Machine Learning, Summer '16

Perceptrons (Rosenblatt 1957)

- Standard Perceptron



- 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 $\mathbf{w}$

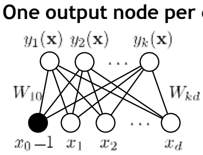
B. Leibe

12

Machine Learning, Summer '16

Extension: Multi-Class Networks

- One output node per class



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

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

Slide adapted from Stefan Roth

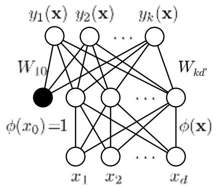
B. Leibe

13

Machine Learning, Summer '16

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

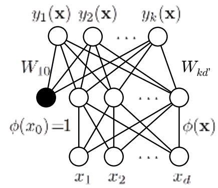
Machine Learning, Summer '16

B. Leibe

14

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!

Machine Learning, Summer '16

B. Leibe

15

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.

Machine Learning, Summer '16

Slide adapted from Geoff Hinton

B. Leibe

16

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

Machine Learning, Summer '16

Slide adapted from Geoff Hinton

B. Leibe

17

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 of a quadratic error function!

Machine Learning, Summer '16

Slide adapted from Geoff Hinton

B. Leibe

18

Loss Functions

- We can now also apply other loss functions
  - L2 loss ⇒ Least-squares regression
    $$L(t, y(\mathbf{x})) = \sum_n \left(y(\mathbf{x}_n) - t_n\right)^2$$
  - 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
    $$L(t, y(\mathbf{x})) = -\sum_n \sum_k \left\{\mathbb{I}(t_n = k)\ln \frac{\exp(y_k(\mathbf{x}))}{\sum_j \exp(y_j(\mathbf{x}))}\right\}$$

Machine Learning, Summer '16

B. Leibe

19

Regularization

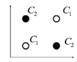
- In addition, we can apply regularizers
  - E.g., an L2 regularizer

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

  - 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.
  - More complex regularization techniques exist (and are an active field of research)

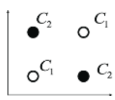
Limitations of Perceptrons

- 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!
  - ⇒ 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...

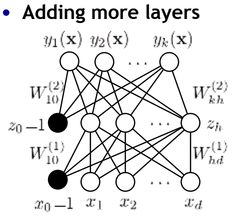
- Didn't we just say that...
  - Perceptrons correspond to generalized linear discriminants
  - And Perceptrons are very limited...
  - *Doesn't this mean that what we have been doing so far in this lecture has the same problems???*
- Yes, this is the case.
  - A linear classifier cannot solve certain problems (e.g., XOR).
  - However, with a non-linear classifier based on the right kind of features, the problem becomes solvable.
  - ⇒ So far, we have solved such problems by hand-designing good features ϕ and kernels $\phi^\top \phi$.
  - ⇒ *Can we also learn such feature representations?*



Topics of This Lecture

Multi-Layer Perceptrons

- Adding more layers



Output layer

Hidden layer

Input layer

- Output

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

Slide adapted from Stefan Roth

Multi-Layer Perceptrons

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

- Activation functions $g^{(k)}$:
  - For example: $g^{(2)}(a) = \sigma(a)$, $g^{(1)}(a) = \tanh(a)$
- 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.

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")

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

Gradient Descent

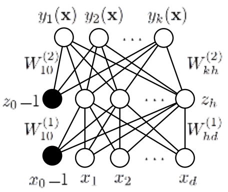
- Two main steps
  1. Computing the gradients for each weight — today
  2. Adjusting the weights in the direction of the gradient — next lecture

Topics of This Lecture

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



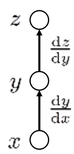
$$\frac{\partial E(\mathbf{W})}{\partial W_{10}^{(2)}} \cdots \frac{\partial E(\mathbf{W})}{\partial W_{kh}^{(2)}}$$

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

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

Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions



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

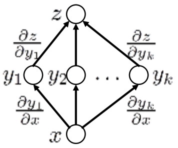
$$\Delta y = \frac{\mathrm{d}y}{\mathrm{d}x}\Delta x$$

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

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

Excursion: Chain Rule of Differentiation

- Multi-dimensional case: Total derivative



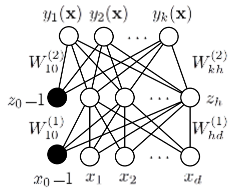
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x} + \ldots$$

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

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

Obtaining the Gradients

- Approach 1: Naive Analytical Differentiation



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

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

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

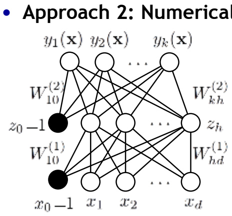
$\Rightarrow$ With increasing depth, there will be exponentially many paths!
$\Rightarrow$ Infeasible to compute this way.

Topics of This Lecture

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

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

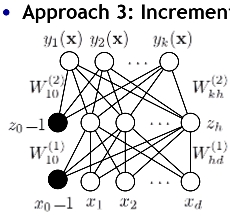
$\Rightarrow$ Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!

Topics of This Lecture

- Perceptrons
  - Definition
  - Loss functions
  - Regularization
  - Limits
- Multi-Layer Perceptrons
  - Definition
  - Learning with hidden units
- Obtaining the Gradients
  - Naive analytical differentiation
  - Numerical differentiation
  - Backpropagation
  - Computational graphs
  - Automatic differentiation

Obtaining the Gradients

- Approach 3: Incremental Analytical Differentiation



$$\begin{array}{ccc} \frac{\partial E(\mathbf{W})}{\partial y_j} & \searrow & \\ \downarrow & & \frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(2)}} \\ \frac{\partial E(\mathbf{W})}{\partial z_j} & \searrow & \\ \downarrow & & \frac{\partial E(\mathbf{W})}{\partial W_{ij}^{(1)}} \\ \frac{\partial E(\mathbf{W})}{\partial x_i} & & \end{array}$$

- 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

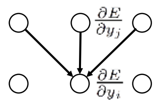
Backpropagation Algorithm

- **Core steps**
  1. Convert the discrepancy between each output and its target value into an error derivate.

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

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

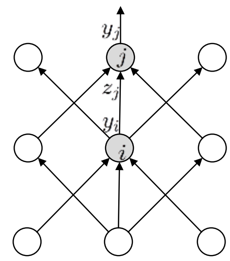
  2. Compute error derivatives in each hidden layer from error derivatives in the layer above.



  3. Use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights

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

Backpropagation Algorithm



E.g. with sigmoid output nonlinearity

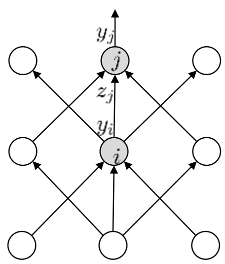
$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

- **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



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

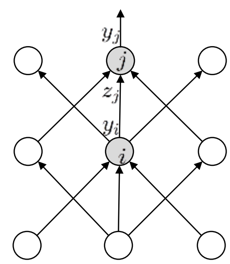
$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

- **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



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

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

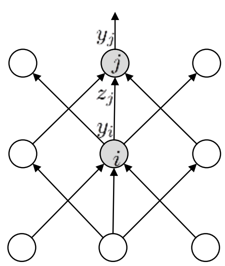
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

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

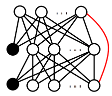
- **Backward Pass**

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

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

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