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
  \[ \Delta i(N) = i(N) - P_L i(N_L) - (1 - P_L) i(N_R) \]

- **Impurity measures**
  - Entropy impurity (information gain):
    \[ i(N) = - \sum_j p(C_j|N) \log_2 p(C_j|N) \]
  - Gini impurity:
    \[ i(N) = \sum_{i \neq j} p(C_i|N)p(C_j|N) = \frac{1}{2} \left[ 1 - \sum_j p^2(C_j|N) \right] \]
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):
    \[
    \Delta E = \sum_{k=1}^{K} \frac{|S_k|}{|S|} \sum_{j=1}^{N} p_j \log_2(p_j)
    \]
Recap: Ensemble Combination

- **Ensemble combination**
  - Tree leaves \((l, \eta)\) store posterior probabilities of the target classes.
  
    \[ p_{l, \eta}(C|x) \]
  
  - Combine the output of several trees by averaging their posteriors (Bayesian model combination)
  
    \[ p(C|x) = \frac{1}{L} \sum_{l=1}^{L} p_{l, \eta}(C|x) \]

B. Leibe
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)!

B. Leibe
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 $f_1, \ldots, f_{l+S}$ into a binary number.
  - Update the “fern leaf” corresponding to that number.

\[
\begin{array}{c}
0 \\
0 \\
1 \\
\end{array}
\rightarrow \quad \text{Update leaf } 100_2 = 4
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:

\[
p(f_1, \ldots, f_{N_f} | C_k) \approx \prod_{j=1}^{M} p(F_j | C_k)
\]

\[
= p(f_1, \ldots, f_S | C_k) \cdot p(f_{S+1}, \ldots, f_{2S} | C_k) \cdot \ldots
\]

⇒ Full joint inside fern

⇒ Naïve Bayes between ferns

⇒ Model with $M \cdot 2^S$ parameters (“Semi-Naïve”).
⇒ Flexible solution that allows complexity/performance tuning.
Today’s Topic

Deep Learning
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
Perceptrons (Rosenblatt 1957)

• Standard Perceptron

\[ y(x) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_d x_d \]

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

• Outputs
  - Linear outputs
    \[ y(x) = w^T x + w_0 \]
  - Logistic outputs
    \[ y(x) = \sigma(w^T x + w_0) \]

• Learning = Determining the weights \( w \)
Extension: Multi-Class Networks

- One output node per class

\[ y_1(x) \quad y_2(x) \quad \ldots \quad y_k(x) \]

\[ W_{10} \quad \ldots \quad W_{kd} \]

\[ x_0 = 1 \quad x_1 \quad x_2 \quad \ldots \quad x_d \]

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

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

Slide adapted from Stefan Roth
Extension: Non-Linear Basis Functions

- Straightforward generalization

  \[ y_1(x) \quad y_2(x) \quad \cdots \quad y_k(x) \]

  \[
  \begin{align*}
  y_1(x) &= \sum_{i=0}^{d} W_{ki} \phi(x_i) \\
  y_2(x) &= \sum_{i=0}^{d} W_{ki} \phi(x_i) \\
  \vdots & \vdots \\
  y_k(x) &= \sum_{i=0}^{d} W_{ki} \phi(x_i)
  \end{align*}
  \]

  \[
  \phi(x_0) = 1 \\
  \phi(x) = \cdots
  \]

  \[
  x_1 \quad x_2 \quad \cdots \quad x_d
  \]

  \[
  W_{10} \quad \cdots \quad W_{kd'}
  \]

- Outputs

  - Linear outputs

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

  - Logistic outputs

    \[ y_k(x) = \sigma \left( \sum_{i=0}^{d} W_{ki} \phi(x_i) \right) \]
Extension: Non-Linear Basis Functions

• Straightforward generalization

\[ y_1(x) \ y_2(x) \ y_k(x) \]

\[ \phi(x_0) = 1 \]

\[ x_1 \ x_2 \ \ldots \ x_d \]

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 \( \phi(x) \) 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.
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

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

\[ w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta (y_k(x_n; w) - t_{kn}) \phi_j(x_n) \]

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

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

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

- In addition, we can apply regularizers
  - E.g., an L2 regularizer
    \[ E(w) = \sum_n L(t_n, y(x_n; w)) + \lambda ||w||^2 \]
  - 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.
  - 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.
  
  \[ \Rightarrow \text{So far, we have solved such problems by hand-designing good features } \phi \text{ and kernels } \phi^T \phi. \]

  \[ \Rightarrow \text{Can we also learn such feature representations?} \]
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
Multi-Layer Perceptrons

• Adding more layers

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

Output layer
Hidden layer
Input layer

Slide adapted from Stefan Roth
Multi-Layer Perceptrons

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

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

- 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
    \[ E(W) = \sum_n L(t_n, y(x_n; W)) + \lambda \Omega(W) \]
    with a loss \( L(\cdot) \) and a regularizer \( \Omega(\cdot) \).
  - E.g., \( L(t, y(x; W)) = \sum_n (y(x_n; W) - t_n)^2 \) \( \text{L}_2 \) loss
    \[ \Omega(W) = \| W \|_F^2 \] \( \text{L}_2 \) regularizer (“weight decay”)

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

  - Compute the gradients for each variable analytically.
  - **What is the problem when doing this?**
Excursion: Chain Rule of Differentiation

- One-dimensional case: Scalar functions

\[
\Delta z = \frac{dz}{dy} \Delta y
\]

\[
\Delta y = \frac{dy}{dx} \Delta x
\]

\[
\Delta z = \frac{dz}{dy} \frac{dy}{dx} \Delta x
\]

\[
\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}
\]
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**

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

- **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**

  \[ y_1(x) \quad y_2(x) \quad y_k(x) \]

  - Idea: Make small changes to \( W(x) \) and accept those that improve \( E(W(x)) \).

  \[ z_0 = 1 \quad x_1 \quad x_2 \quad ... \quad x_d \]

  \[ W^{(1)}_{10} \quad W^{(2)}_{10} \quad W^{(2)}_{kh} \quad W^{(1)}_{hd} \]

  - Given the current state \( W^{(\tau)} \), we can evaluate \( E(W^{(\tau)}) \).
  - Horribly inefficient! Need several forward passes for each weight. Each forward pass is one run over the entire dataset!
Topics of This Lecture

- **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**

  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.

  \[ E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2 \]
  \[ \frac{\partial E}{\partial y_j} = -(t_j - y_j) \]

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

  \[ \frac{\partial E}{\partial y_i} \]

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

\[ \frac{\partial E}{\partial y_j} \rightarrow \frac{\partial E}{\partial w_{ik}} \]

Slide adapted from Geoff Hinton
Backpropagation Algorithm

- **Notation**
  - $y_j$: Output of layer $j$
  - $z_j$: Input of layer $j$

- **Connections:**
  - $z_j = \sum_i w_{ij} y_i$
  - $y_j = g(z_j)$

E.g. with sigmoid output nonlinearity

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

Slide adapted from Geoff Hinton
### Backpropagation Algorithm

#### Notation

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

#### Connections:

$z_j = \sum_i w_{ij} y_i$

#### Derivatives:

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

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

Slide adapted from Geoff Hinton
Summary: MLP Backpropagation

- **Forward Pass**
  
  \[ y^{(0)} = x \]
  
  for \( k = 1, \ldots, l \) do
  
  \[ z^{(k)} = W^{(k)} y^{(k-1)} \]
  
  \[ y^{(k)} = g_k(z^{(k)}) \]
  
  endfor
  
  \[ y = y^{(l)} \]
  
  \[ E = L(t, y) + \lambda \Omega(W) \]

- **Backward Pass**
  
  \[ h \leftarrow \frac{\partial E}{\partial y} = \frac{\partial}{\partial y} L(t, y) + \lambda \frac{\partial}{\partial y} \Omega \]
  
  for \( k = l, l-1, \ldots, 1 \) do
  
  \[ h \leftarrow \frac{\partial E}{\partial z^{(k)}} = h \odot g'(y^{(k)}) \]
  
  \[ \frac{\partial E}{\partial W^{(k)}} = hy^{(k-1)\top} + \lambda \frac{\partial \Omega}{\partial W^{(k)}} \]
  
  \[ h \leftarrow \frac{\partial E}{\partial y^{(k-1)}} = W^{(k)\top} h \]
  
  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
  - 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/