MVA - Discrete Inference and Learning
Lecture 6
- Classical Learning

Yuliya Tarabalka
Inria Sophia Antipolis-Méditerranée - TITANE team
Université Côte d'Azur - France
Overview

1. Classification based on Features
   - Decision Boundary
   - Linear Decision Boundary
   - Non-linear Decision Boundary
2. Feature extraction
3. Machine Learning Methods
   - Support Vector Machine (SVM)
   - Multi-Layer Perceptron (MLP)
   - Random Forest (RF)
Overview

1. Classification based on Features
   - Decision Boundary
   - Linear Decision Boundary
   - Non-linear Decision Boundary

2. Feature extraction

3. Machine Learning Methods
   - Support Vector Machine (SVM)
   - Multi-Layer Perceptron (MLP)
   - Random Forest (RF)
Classical Learning

- Features extract very basic, low level information
- We want very high level information (e.g. class of objects)
- Classical Learning: Learn the mapping between low level features and high level information
Machine Learning is a huge (growing) field

Many different approaches for modeling/parametrizing this mapping!
Methods

- Choice of method not always rational
- Different pros/cons
- Speed, memory, scalability of training data, ease of implementation, ease of hyper parameter tuning, ...
- First intuitive understanding of the problems, then identifying methods
Decision based on features

Toy example

Task: Classify fruits into either bananas or apples

Extracted Feature Vector

- Hue (yellow to red)
- Elongation (max extend over min extend)
Some training data

- Feature space is just 2D
- Datapoints can be plotted as a scatter plot
Some training data

- Feature space is just 2D
- Datapoints can be plotted as a scatter plot
- Can we “learn”, which part of the feature space is bananas/apples?
(Very) simple idea: Split the feature space into two half spaces
(Very) simple idea: Split the feature space into two half spaces.

During application, classify data based on this decision boundary.
(Very) simple idea: Split the feature space into two half spaces

During application, classify data based on this decision boundary
Perceptron

\[ y = \text{sign}(w^T x + b) \]  

- \( y \in \{-1, 1\} \): Predicted class
- \( x \in \mathbb{R}^2 \): Feature vector
- \( w \in \mathbb{R}^2 \): “Weight vector” (needs to be learned)
- \( b \in \mathbb{R} \): “Bias” (needs to be learned)
What if no such line exists?
Quite often, problem not linearly separable
Needs non-linear decision boundary
Non-linear Decision Boundary

- Decision boundaries of more complex ML techniques usually non-linear
- Regions need not be connected
kNN

- Very simple idea: k-Nearest-Neighbors for classification
kNN

- Very simple idea:
  - k-Nearest-Neighbors for classification
- For a sample find the k (e.g. 5) closest data points in the training dataset
kNN

- Very simple idea: k-Nearest-Neighbors for classification
- For a sample find the k (e.g. 5) closest data points in the training dataset
- Look at the labels of those neighbors
kNN

- Very simple idea: k-Nearest-Neighbors for classification
- For a sample find the k (e.g. 5) closest data points in the training dataset
- Look at the labels of those neighbors
- Fast lookup through trees/approximate methods
- Needs to keep all training data around
kNN Example - Simple
kNN Example - Simple - kNN K=1
kNN Example - Simple - kNN $K=5$
kNN Example - Simple - kNN K=25
kNN Example - Hard
kNN Example - Hard - kNN $K=1$
kNN Example - Hard - kNN K=5
kNN Example - Hard - kNN K=25
kNN Example
Model Complexity vs Overfitting

- With sufficient model complexity, it is often easy to get ZERO training error.
- Generalization is what matters!
- **Test on data not used during training**
  - Disjoint train and test set
  - Non-overlapping samples if spatial features are used
  - Semi-manual parameter tuning (grid-search, etc.) needs third independent data set
Why do we need feature extraction?
Motivation

**Main motivation:** get out most of the data

For classification task: find a space where samples from different classes are well separable

Objectives:

- Reduce computational load of the classifier
- Increase data consistency
- Incorporate different sources of information into a feature vector: spectral, spatial, multisource, ...
Motivation - Curse of dimensionality

- Too few features do not allow to discriminate between classes
  - In the color image, both trees and a truck are green

- As the dimensionality of the feature space increases, the classifier’s performance increases until the optimal number of features is reached

- Further increasing the dimensionality without increasing the number of training samples yields a performance decrease
Motivation - Curse of dimensionality

- As the dimensionality increases:
  - The volume of the hypersphere tends to zero
  - A larger percentage of the training data resides in the corners of the feature space
  - Distance measures start losing their effectiveness
  - Gaussian likelihoods become flat and heavy tailed distributions
Feature extraction

How to reduce data dimensions?

**Principal component analysis**
Convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, called *principal components*.

**Discriminant analysis**
Find the best set of vectors which best separates the patterns.
Principal component analysis

- **Goal**: represent data is a space that best describes the variation in a sum-squared error sense

- Projection onto eigenvectors that correspond to the first few largest eigenvalues of the covariance matrix
  - $d$-dimensional data are represented in a lower-dimensional space
  - Reduces the space and time complexities

- Intuitive introduction: [http://www.youtube.com/watch?v=BfTMmoDFXyE&feature=related](http://www.youtube.com/watch?v=BfTMmoDFXyE&feature=related)
Principal component analysis

- **Step 1**: Get some data

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>2.2</td>
<td>2.9</td>
</tr>
<tr>
<td>1.9</td>
<td>2.2</td>
</tr>
<tr>
<td>3.1</td>
<td>3.0</td>
</tr>
<tr>
<td>2.3</td>
<td>2.7</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>1.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Feature extraction

Principal component analysis

- **Step 2**: Subtract the mean
  - From each of the data dimensions (from $x$- and $y$-dimension)

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>2.2</td>
<td>2.9</td>
</tr>
<tr>
<td>1.9</td>
<td>2.2</td>
</tr>
<tr>
<td>3.1</td>
<td>3.0</td>
</tr>
<tr>
<td>2.3</td>
<td>2.7</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>1.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Data =

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.69</td>
<td>.49</td>
</tr>
<tr>
<td>-1.31</td>
<td>-1.21</td>
</tr>
<tr>
<td>.39</td>
<td>.99</td>
</tr>
<tr>
<td>.09</td>
<td>.29</td>
</tr>
<tr>
<td>1.29</td>
<td>1.09</td>
</tr>
<tr>
<td>.49</td>
<td>.79</td>
</tr>
<tr>
<td>.19</td>
<td>-.31</td>
</tr>
<tr>
<td>-.81</td>
<td>-.81</td>
</tr>
<tr>
<td>-.31</td>
<td>-.31</td>
</tr>
<tr>
<td>-.71</td>
<td>-1.01</td>
</tr>
</tbody>
</table>

DataAdjust =
**Step 3**: Calculate the covariance matrix

Data:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>2.2</td>
<td>2.9</td>
</tr>
<tr>
<td>1.9</td>
<td>2.2</td>
</tr>
<tr>
<td>3.1</td>
<td>3.0</td>
</tr>
<tr>
<td>2.3</td>
<td>2.7</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>1.1</td>
<td>0.9</td>
</tr>
</tbody>
</table>

\[ \text{cov} = \begin{pmatrix} \frac{616555556}{615444444} & \frac{615444444}{716555556} \\ \frac{615444444}{616555556} & \frac{716555556}{716555556} \end{pmatrix} \]
**Principal component analysis**

- **Step 4**: Calculate the unit eigenvectors and eigenvalues of the covariance matrix

\[
cov = \begin{pmatrix}
0.616555556 & 0.615444444 \\
0.615444444 & 0.716555556
\end{pmatrix}
\]

\[
eigenvalues = \begin{pmatrix}
0.0490833989 \\
1.28402771
\end{pmatrix}
\]

\[
eigenvectors = \begin{pmatrix}
-0.735178656 & -0.677873399 \\
0.677873399 & -0.735178656
\end{pmatrix}
\]
Principal component analysis

- The 1st eigenvector (principle component) shows how data in two dimensions are related along the eigenvector line.

- The 2nd eigenvector shows that all the points are off to the side of the main line by some amount.

- Eigenvectors are lines that characterize the data.

- The next steps: transforming the data so that it is expressed in terms of these lines.
Principal component analysis

- **Step 5:** Choose components and form a feature vector

  - Order eigenvectors by eigenvalues
    - This gives the components in order of significance
    - You can decide to ignore the components of lesser significance ⇒ final data will have less dimensions ($p < d$)

  - Form a feature vector (matrix of vectors):
    \[ \text{FeatureVector} = (\text{eig}_1 \quad \text{eig}_2 \quad \text{eig}_3) \]

  - For our example, two feature vectors are possible:

\[
\begin{align*}
eigenvalues &= \begin{pmatrix} .0490833989 \\ 1.28402771 \end{pmatrix} \\
eigenvectors &= \begin{pmatrix} -.735178656 & -.677873399 \\ .677873399 & -.735178656 \end{pmatrix}
\end{align*}
\]

\[
\begin{pmatrix} -.677873399 & -.735178656 \\ -.735178656 & .677873399 \end{pmatrix}
\text{ or }
\begin{pmatrix} -.677873399 \\ -.735178656 \end{pmatrix}
\]
Principal component analysis

- **Step 6**: Derive the new dataset:

\[
\text{FinalData} = \text{FeatureVector}^T \times \text{RowDataAdjust}
\]

where \( \text{RowDataAdjust} \) is the mean-adjusted data transposed.

- It will give us the original data solely in terms of the vectors we chose.
# Principal component analysis

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
</tr>
<tr>
<td>2.2</td>
<td>2.9</td>
</tr>
<tr>
<td>1.9</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Data = 3.1 3.0 2.3 2.7 2 1.6 1 1.1 1.5 1.6 1.1 0.9

Transformed Data =

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.827970186</td>
<td>-0.175115307</td>
</tr>
<tr>
<td>1.77758033</td>
<td>0.142857227</td>
</tr>
<tr>
<td>-0.992197494</td>
<td>0.384374989</td>
</tr>
<tr>
<td>-0.274210416</td>
<td>0.130417207</td>
</tr>
</tbody>
</table>

Data transformed with 2 eigenvectors
Principal component analysis (PCA)

- If only one eigenvector was kept, the transformed data will have only one dimension
Example of PCA for hyperspectral image analysis

- Principal component analysis in the spectral space
  - Principal components (PCs) 1-3 contain 97% of information from original 103 channels
Principal component analysis

- Projection onto eigenvectors that correspond to the first few largest eigenvalues of the covariance matrix

![Ideal situation](image1)

![Adversary situation](image2)
Discriminant analysis

- PCA seeks directions that are efficient for representation
  - *Unsupervised technique*

- Discriminant analysis seeks directions that are efficient for discrimination
  - *Supervised technique*
Discriminant analysis

- Projection onto directions that can best separate data of different classes
Discriminant analysis

- Project on line in the direction \( \mathbf{v} \) which maximizes:

\[
J(\mathbf{v}) = \frac{(\bar{\mu}_1 - \bar{\mu}_2)^2}{\bar{s}_1^2 + \bar{s}_2^2}
\]

\( \bar{\mu}_1 \): mean of class 1
\( \bar{\mu}_2 \): mean of class 2
\( \bar{s}_1^2 \): scatter of class 1
\( \bar{s}_2^2 \): scatter of class 2

- Main drawback: in most real-life cases, projection to even the best line results in unseparable projected samples
1. Classification based on Features
   - Decision Boundary
   - Linear Decision Boundary
   - Non-linear Decision Boundary

2. Feature extraction

3. Machine Learning Methods
   - Support Vector Machine (SVM)
   - Multi-Layer Perceptron (MLP)
   - Random Forest (RF)
Models

1. Classification based on Features
   - Decision Boundary
   - Linear Decision Boundary
   - Non-linear Decision Boundary

2. Feature extraction

3. Machine Learning Methods
   - Support Vector Machine (SVM)
   - Multi-Layer Perceptron (MLP)
   - Random Forest (RF)
Reconsider the perceptron:

\[
y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \quad (2)
\]
Reconsider the perceptron:

**Perceptron**

\[ y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \]  \hspace{1cm} (2)
SVM

Reconsider the perceptron:

\[ y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \] (2)

Perceptron

\[ y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \] (2)
Reconsider the perceptron:

\[ y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \]  \hspace{1cm} (2)
Reconsider the perceptron:

**Perceptron**

\[ y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \]  (2)
Reconsider the perceptron:

$$y = \text{sign}(\mathbf{w}^T \mathbf{x} + b) \quad (2)$$
Machine Learning Methods

Support Vector Machine (SVM)

SVM

Reconsider the perceptron:

Perceptron

\[ y = \text{sign} (w^T x + b) \]  (2)

- Don’t just pick any decision boundary
- Pick the one with the \textit{maximal margin}
- Perceptron of maximal stability

\[ x_1 \]
\[ x_2 \]
\[ H_1 \]
\[ H_2 \]
\[ H_3 \]
\[ H_4 \]

Y. Tarabalka

Lecture 6: Classical Learning

27 Nov 2017

54 / 131
Reconsider the perceptron:

**Perceptron**

$$y = \text{sign}(\mathbf{w}^T \mathbf{x} + b)$$ (2)

- Don’t just pick any decision boundary
- Pick the one with the *maximal margin*
- Perceptron of maximal stability
SVM

- Maximal margin equivalent to:
  Minimize $||w||^2$
  subject to $\hat{y}_i(w^T x_i - b) \geq 1$
SVM

- Maximal margin equivalent to:
  Minimize $||w||^2$
  subject to $\hat{y}_i(w^T x_i - b) \geq 1$

- Allow small errors (soft margin):
  Minimize $\lambda ||w||^2 + \frac{1}{n} \sum_{i=1}^{n} \xi_i$
  subject to $\hat{y}_i(w^T x_i - b) \geq 1 - \xi_i$
  ($\xi_i \geq 0$)
The Lagrangian dual gives:
Maximize
\[ \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{y}_i \alpha_i (x_i \cdot x_j) \hat{y}_j \alpha_j \]
subject to \( \sum_{i=1}^{n} \alpha_i \hat{y}_i = 0 \)

Support vectors: \( x_i \) if \( \alpha_i \neq 0 \)

Classification: \( \text{sign}(w^T x + b) \)
with \( w = \sum_{i=1}^{n} \alpha_i \hat{y}_i x_i \)
The Lagrangian dual gives:
Maximize
\[
\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{y}_i \alpha_i (x_i \cdot x_j) \hat{y}_j \alpha_j
\]
subject to \(\sum_{i=1}^{n} \alpha_i \hat{y}_i = 0\)

Support vectors: \(x_i\) if \(\alpha_i \neq 0\)

Classification: \(\text{sign}(w^T x + b)\)
with \(w = \sum_{i=1}^{n} \alpha_i \hat{y}_i x_i\)

What if \(x_i\) not linear separable at all?
The Lagrangian dual gives:
Maximize
\[
\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{y}_i \alpha_i (x_i \cdot x_j) \hat{y}_j \alpha_j
\]
subject to \( \sum_{i=1}^{n} \alpha_i \hat{y}_i = 0 \)

Support vectors: \( x_i \) if \( \alpha_i \neq 0 \)

Classification: \( \text{sign}(w^T x + b) \)
with \( w = \sum_{i=1}^{n} \alpha_i \hat{y}_i x_i \)

What if \( x_i \) not linear separable at all?
→ Compute new features \( x \mapsto \phi(x) \)
The Lagrangian dual gives:

Maximize

$$\sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{y}_i \alpha_i (\phi(x_i) \cdot \phi(x_j)) \hat{y}_j \alpha_j$$

subject to $$\sum_{i=1}^{n} \alpha_i \hat{y}_i = 0$$

Support vectors: $$x_i$$ if $$\alpha_i \neq 0$$

Classification: $$\text{sign}(w^T \phi(x) + b)$$
with $$w = \sum_{i=1}^{n} \alpha_i \hat{y}_i \phi(x_i)$$

What if $$x_i$$ not linear separable at all?
→ Compute new features $$x \mapsto \phi(x)$$
The Lagrangian dual gives:

Maximize

\[ \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{y}_i \alpha_i k(x_i, x_j) \hat{y}_j \alpha_j \]

subject to \( \sum_{i=1}^{n} \alpha_i \hat{y}_i = 0 \)

Support vectors: \( x_i \) if \( \alpha_i \neq 0 \)

Classification:

\[ \text{sign}(\sum_{i=1}^{n} \alpha_i \hat{y}_i k(x_i, x) + b) \]

What if \( x_i \) not linear separable at all?

→ Compute new features \( x \mapsto \phi(x) \)

Use \( k(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \)
SVM Kernels

- Multiple kernels exist
  - Linear $k(x_i, x_j) = x_i \cdot x_j$
  - Polynomial $k(x_i, x_j) = (x_i \cdot x_j)^d$
  - RBF $k(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$
  - Hyperbolic tangent $k(x_i, x_j) = \tanh(\kappa \cdot x_i \cdot x_j + c)$

- Linear kernel very fast and easy to train, but very simple
- RBF kernel very powerful and most often used
- Kernel can (should) be adapted to task and data
  - e.g. complex-valued kernels are possible [Moser and Serpico, 2014]
    $$k(z, s) = \mathbb{R} \left[ \exp \left( -\frac{1}{2\sigma^2} \sum_{r=1}^{d} (z_r - s_r^*)^2 \right) \right]$$
- Kernels for different features can be fused into one common kernel
SVM Conclusion

- Kernels can be designed to different purposes
- Hyperparameter tuning not easy
  → Usually grid search with cross validation
- Slow for large amounts of data
  → Potentially results in many support vectors and thus scalar products during prediction
- (Usually) all data needs to be considered at once
  → No “streaming” of data
- Designed for binary tasks
  → Extension to multi-class problems usually decreases performance and increases computational load
Models

1. Classification based on Features
   - Decision Boundary
   - Linear Decision Boundary
   - Non-linear Decision Boundary

2. Feature extraction

3. Machine Learning Methods
   - Support Vector Machine (SVM)
   - Multi-Layer Perceptron (MLP)
   - Random Forest (RF)
Multi-Layer Perceptron

- Feed forward neural network
- Neural networks “inspired by biology”
  - But work quite differently
- Core idea: concatenate multiple simple mappings to get one powerful mapping
- Multiple simple steps more powerful than one complex step
- Keep everything (mostly) differentiable
- Train by doing gradient descend on classification error
Building Blocks

Standard Layers:

- Fully connected layer with...
- ... activation function
Building Blocks

Standard Layers:

- Fully connected layer with...
- ... activation function

Special Layers (selection):

- Dropout (for regularization)
- Normalization (Improves training)
- Softmax (Produces nice classification output)
\[ x^{n+1} = y^n = f(A^n \cdot x^n + b^n) \] (3)

- \( x^n \): Layer input
- \( y^n = x^{n+1} \): Layer output
- \( A^n \): Weights
- \( b^n \): Bias
- \( f(\cdot) \): Activation function
Activation Functions

\[ y^n = f(A^n \cdot x^n + b^n) \] (4)

- Assume \( f(x) = x \)
- Layer can assume any linear function (plus offset)
Activation Functions

\[ y^n = f(A^n \cdot x^n + b^n) \] (4)

- Assume \( f(x) = x \)
- Layer can assume any linear function (plus offset)
- Stacked layers can't improve that
- Activation function must be non-linear
Activation Functions

Typical choices:

ReLU

\[ f(x_i)_i = \text{max}(x_i, 0) \]  \hspace{1cm} (5)

Sigmoid / Logistic

\[ f(x_i)_i = \frac{1}{1 + e^{-x_i}} \] \hspace{1cm} (6)

TanH

\[ f(x_i)_i = \tanh(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \] \hspace{1cm} (7)
Activation Functions

Typical choices:

ReLU

\[ f(x_i)_i = \max(x_i, 0) \]  

- ReLU (and variations of it) today the most common choice
- Better for deep networks
  - Derivative of activation function = 1 (in positive direction)
  - No saturation (in positive direction)
  - Gradients propagate better
Training

- How to find correct model parameters $\theta$?
  - weight values
  - bias values
  - sometimes aux parameters
Training

- How to find correct model parameters $\theta$?
  - weight values
  - bias values
  - sometimes aux parameters

- Setup/define energy function objective $E(\theta)$

- Derive analytic gradients $\frac{\partial E(\theta)}{\partial \theta}$

- Perform gradient descent $\Delta \theta = -\lambda \cdot \frac{\partial E(\theta)}{\partial \theta}$
  - Usually slightly more sophisticated, more later
Training Objective

Empirical Risk Minimization (over N training samples)

\[ E(\theta) = \sum_{\alpha}^{N} e(y^L(x_\alpha, \theta), \hat{y}_\alpha) \]

Known/Desired value/label of \( x_\alpha \)

Training sample

with, e.g.,:

\[ e(y^a, y^b) = \left| y^a - y^b \right|^2 \]

Though bad for classification, see softmax layer later.

- Energy function defines training loss
- Gradient descent will try to minimize this
- Usually not convex (as network not convex)
Backpropagation

- How to compute $\frac{\partial E(\theta)}{\partial \theta}$?
- MLP is concatenation of “simple” functions
  $y^L(...y^2(y^1(x^1, \theta^1), \theta^2),...\theta^L)$
Backpropagation

- How to compute $\frac{\partial E(\theta)}{\partial \theta}$?
- MLP is concatenation of “simple” functions $y^L(y^2(y^1(x^1, \theta^1), \theta^2), \ldots \theta^L)$
- Exploit chain rule

\[
\frac{\partial E(\theta)}{\partial \theta^1} = \frac{\partial E(\theta)}{\partial y^L} \cdots \frac{\partial y^3}{\partial y^2} \cdot \frac{\partial y^2}{\partial y^1} \cdot \frac{\partial y^1}{\partial \theta^1}
\]

(11)

Gradient computation happens in two passes:

- **Forward pass:** Feeds training data through network, computes all $y_n$ and training loss.
- **Backward pass:** Feeds error gradient backward through network, computes all $\frac{\partial E(\theta)}{\partial y_n}$ and $\frac{\partial E(\theta)}{\partial \theta_n}$. 

Y. Tarabalka
Lecture 6: Classical Learning
Backpropagation

- How to compute \( \frac{\partial E(\theta)}{\partial \theta} \)?
- MLP is concatenation of “simple” functions \( y^L(\ldots y^2(y^1(x^1, \theta^1), \theta^2), \ldots \theta^L) \)
- Exploit chain rule

\[
\frac{\partial E(\theta)}{\partial \theta^1} = \frac{\partial E(\theta)}{\partial y^L} \cdot \frac{\partial y^3}{\partial y^2} \cdot \frac{\partial y^2}{\partial y^1} \cdot \frac{\partial y^1}{\partial \theta^1}
\]

(11)

- Gradient computation happens in two passes:
  - Forward pass:
    - Feeds training data through network
    - Computes all \( y^n \) and training loss
  - Backward pass:
    - Feeds error gradient backward through network
    - Computes all \( \frac{\partial E(\theta)}{\partial y^n} \) and \( \frac{\partial E(\theta)}{\partial \theta^n} \)
Stochastic Gradient Descent

- Exact gradient usually not needed or wanted
- Just empirical average over $N$ samples anyways
- Stochastic Gradient Descent: Split into batches of $M < N$ samples and update weights after every batch

$$\Delta \theta = -\lambda \cdot \frac{\partial \hat{E}(\theta)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{\alpha}^{M} e(y^L(x_\alpha, \theta), \hat{y}_\alpha)$$  \hspace{1cm} (12)

- Usually small batch sizes (eg. around 128) sufficient
  - Stepsize limited by curvature of energy function, not by precision of gradient
  - Computation time increases with $O(M)$, precision of gradient only with $O(\sqrt{M})$
  - Large batch sizes lead to sharp minimizers that don’t generalize
- Further reading: [Keskar et al., 2016]
Parameter Update Rule

- $\Delta \theta = -\lambda \cdot \frac{\partial \hat{E}(\theta)}{\partial \theta}$ most simple update rule

- Momentum
  - Accumulate “momentum” over time
  - Pick up speed in the valley direction, average out noise

- Adam [Kingma and Ba, 2014]/Adagrad/Adadelta [Zeiler, 2012]
  - Normalize based on average gradient variance in the past
Parameter Initialization

- How to initialize $\theta$?
- Random Gaussian
- Xavier (and some variants) [Glorot and Bengio, 2010]
  - Draw weights randomly
  - Choose variance per layer depending on input/output size
  - Balance variance to keep signal/gradient variance constant
Special Layers

- Softmax
- Normalization
- Dropout
Softmax

\[ f(x_i)_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \]  \hspace{1cm} (13)

- Special (last) layer/activation for classification
- Creates vector that sums to one (read probabilities), one element per class
- Usually together with a specific optimization objective: Cross-entropy loss
  - Comparing the predicted probability mass distribution to the ground truth one
Dropout

- [Srivastava et al., 2014]
- During training, randomly disable neurons with probability $p$
- During application, scale output with $1 - p$
- Prevents co-adaptation
- Fosters redundancy throughout the network
- Reduces overfitting and improves generalization
Normalization

- Normalization can be important for learning
- Neither signal (forward) nor gradients (backward) must explode/shrink in magnitude
Normalization

- Normalization can be important for learning
- Neither signal (forward) nor gradients (backward) must explode/shrink in magnitude
- Input Normalization
  - Normalize input to have zero mean and unit stddev
Normalization

- Normalization can be important for learning
- Neither signal (forward) nor gradients (backward) must explode/shrink in magnitude
- Input Normalization
  - Normalize input to have zero mean and unit stddev
- Local Response Normalization (LRN) [Krizhevsky et al., 2012]
  - Special layer placed at strategic locations
  - Let strong activations inhibit activations of other neurons in same layer
  - Normalizes the otherwise unbounded output of ReLU
Normalization

- Normalization can be important for learning
- Neither signal (forward) nor gradients (backward) must explode/shrink in magnitude
- Input Normalization
  - Normalize input to have zero mean and unit stddev
- Local Response Normalization (LRN) [Krizhevsky et al., 2012]
  - Special layer placed at strategic locations
  - Let strong activations inhibit activations of other neurons in same layer
  - Normalizes the otherwise unbounded output of ReLU
- Batch Normalization [Ioffe and Szegedy, 2015]
  - Special layer placed at strategic locations
  - Normalize mean and variance of activations across training batch (or accumulate running averages)
  - After learning, becomes fixed scale & offset
Handling Overfitting

- Dropout

Weight regularization:
- Penalize large weight values
  - e.g., add $\lambda \cdot |\theta|^2$ to optimization objective

Data Augmentation:
- Randomly modify training data
  - Based on what kind of invariances you want to have
- Resistance to noise: add noise
- Resistance to brightness/contrast/hue changes: Change those
- Translation/Rotation (ex. for images)
  - Can also be applied to data before extracting features!
Handling Overfitting

- Dropout
- Weight regularization
  - Penalize large weight values
  - e.g., add $\lambda \cdot |\theta|^2$ to optimization objective
  - Soft limit on model complexity
Handling Overfitting

- **Dropout**
- **Weight regularization**
  - Penalize large weight values
  - e.g., add $\lambda \cdot |\theta|^2$ to optimization objective
  - Soft limit on model complexity

**Data Augmentation**

- Randomly modify training data
- Based on what kind of invariances you want to have
  - Resistance to noise: add noise
  - Resistance to brightness/contrast/hue changes: Change those
  - Translation/Rotation (ex. for images)
  - Can also be applied to data before extracting features!
Increasing Depth

- Recent trend goes towards deeper networks
- Networks more powerful, but ...
- ... more difficult to train
  - Gradients collapse/explode/diffuse through the layers
- This is the book to read: Deep Learning [Goodfellow et al., 2016]
MLPs provide a functional mapping \( f : \mathcal{X} \rightarrow \mathcal{Y} \)

- \( f \) needs to be differentiable (due to backprop)
- Usually \( \mathcal{X} \equiv \mathbb{R}^d \) (or \( \mathbb{R}^{N \times M} \))
- But: (e.g.) PolSAR images are \( \mathbb{C}^{N \times M} \)
  - One solution: Compute real-valued features, then use standard MLP
    → Advantage: Usage of common MLPs and their extensions
    → Disadvantage: Dependency on feature extraction
  - Second solution: Use complex-valued MLP
    → Advantage: No dependency on feature extraction
    → Disadvantage: Math slightly more complicated
Complex-valued MLPs

Gradient in $\mathbb{R}$

$$f : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto f(x)$$

$$\frac{\partial f}{\partial x} = f'(x)$$

Gradient in $\mathbb{C}$

$$f : \mathbb{C} \rightarrow \mathbb{C}, z \mapsto f(z)$$

$$\frac{\partial f}{\partial z} = \frac{1}{2} \left( \frac{\partial f}{\partial \Re z} - i \frac{\partial f}{\partial \Im z} \right)$$

$$\frac{\partial f}{\partial z^*} = \frac{1}{2} \left( \frac{\partial f}{\partial \Re z} + i \frac{\partial f}{\partial \Im z} \right)$$
Complex-valued MLPs

Activation in $\mathbb{R}$

Analytical and bounded
$\rightarrow$ e.g. tanh, logistic function
(ReLU as exception)

Activation in $\mathbb{C}$

Analytical and bounded?
$\rightarrow$ only constant functions
Complex-valued MLPs

**Activation in \( \mathbb{R} \)**

- \text{relu}
- \text{sigmoid}
- \text{tanh}

**Activation in \( \mathbb{C} \)**

- Analytical or bounded
  - e.g. tanh

Analytical and bounded
→ e.g. tanh, logistic function

Analytical or bounded
→ e.g. tanh
Complex-valued MLPs

Activation in $\mathbb{R}$

Analytical and bounded
$\rightarrow$ e.g. tanh, logistic function

Activation in $\mathbb{C}$

Analytical or bounded
$\rightarrow$ e.g. split-tanh

$f(z) = \tanh(\Re(z)) + i \tanh(\Im(z))$
Complex-valued MLPs

- **PolSAR Data:** \( \mathbb{C}^{N \times M \times 3 \times 3} \)
- **Input:** Local patches, each pixel a Hermitian matrix (local covariance matrix of complex-valued scattering vector)
- **Activation of few neurons in first layer is shown.**
MLP Conclusion

- Architecture design a bit of an art
  - Though some tips/tricks exist
- Can ingest a lot of training data
- Training/Application not fast
- With modern tricks (ReLU, normalization, ...) scale surprisingly well
  - Up to very complex networks
  - Trained on lots of data
Models

1. Classification based on Features
   - Decision Boundary
   - Linear Decision Boundary
   - Non-linear Decision Boundary

2. Feature extraction

3. Machine Learning Methods
   - Support Vector Machine (SVM)
   - Multi-Layer Perceptron (MLP)
   - Random Forest (RF)
From kNN to Search Trees

- Data samples $\mathbf{x}$
  - Pixel information, image patch, feature vector, etc.
  - Often $\mathbf{x} \in \mathbb{R}^n$

- Classification:
  $\Rightarrow$ Estimate class label

- Training data: Values of target variable given e.g. class label
Task: Given training data, estimate label of query sample
Task: Given training data, estimate label of query sample

kNN/Parzen Window:
- Compute distance to all samples
Task: Given training data, estimate label of query sample

**kNN/Parzen Window:**
- Compute distance to all samples
- Select samples within window of given size (Parzen)
From kNN to Search Trees

- **Task**: Given training data, estimate label of query sample
- **kNN/Parzen Window**:
  - Compute distance to **all** samples
  - Select samples within window of given size (Parzen)
  - Use these samples to estimate target variable, e.g. class label

- **Problem**: Computationally expensive (exhaustive search)
From kNN to Search Trees

- Search trees
  → Quad/Octree, KD tree, etc.
Search trees
→ Quad/Octree, KD tree, etc.
- Divide space recursively into cells
From kNN to Search Trees

- Search trees
  → Quad/Octree, KD tree, etc.
  - Divide space recursively into cells
Search trees
→ Quad/Octree, KD tree, etc.
- Divide space recursively into cells
- Given a query, find relevant cells
From kNN to Search Trees

- Search trees
  → Quad/Octree, KD tree, etc.
  - Divide space recursively into cells
  - Given a query, find relevant cells
From kNN to Search Trees

- **Search trees**
  - Quad/Octree, KD tree, etc.
  - Divide space recursively into cells
  - Given a query, find relevant cells
  - Perform exhaustive search in these cells ONLY
From kNN to Search Trees

- **Search trees**
  - Quad/Octree, KD tree, etc.
  - Divide space recursively into cells
  - Given a query, find relevant cells
  - Perform exhaustive search in these cells ONLY

- **Exact search**: Leads to equivalent results
From kNN to Search Trees

- Search trees
  → Quad/Octree, KD tree, etc.
    - Divide space recursively into cells
    - Given a query, find relevant cells
    - Perform exhaustive search in these cells ONLY

- Exact search: Leads to equivalent results
- Approximation: Use samples within query cell directly
Cell construction

From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees

- Cell construction
Cell construction
→ Simple threshold operation
→ Different threshold definitions (e.g. equi-sized cells, threshold as data median) lead to different search tree variants (e.g. quad-tree, k-D tree).
Cell construction
→ Simple threshold operation
Decision stump:

\[ t(x) = \begin{cases} 
0 & \text{if } x_1 < \theta_1 \\
1 & \text{otherwise.} 
\end{cases} \]
From Search Trees to (Random) Decision Trees

- Cell construction → Simple threshold operation
- Decision stump:
  \[ t(x) = \begin{cases} 
  0 & \text{if } x_1 < \theta_1 \\
  1 & \text{otherwise.} 
\end{cases} \]
Cell construction
→ Simple threshold operation
Decision stump:
\[ t(x) = \begin{cases} 
0 & \text{if } x_1 < \theta_1 \\
1 & \text{otherwise.} 
\end{cases} \]
Cell construction
  → Simple threshold operation

Decision stump:
\[ t(x) = \begin{cases} 
0 & \text{if } x_1 < \theta_1 \\
1 & \text{otherwise.} 
\end{cases} \]

When to stop? Minimal resolution reached, purity, ...

How to select split points?
Randomly, optimized selection
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
From Search Trees to (Random) Decision Trees
Local estimate of the target variable (e.g. class posterior) is assigned to cells
From Search Trees to (Random) Decision Trees

- Local estimate of the target variable (e.g. class posterior) is assigned to cells
- Results in highly non-linear, even non-connected (but piece-wise constant) decision boundaries
Other node tests are possible:

- **Axis-aligned:**

\[
t(x) = \begin{cases} 
0 & \text{if } x_1 < \theta_1 \\
1 & \text{otherwise.}
\end{cases}
\]

\[
t(x) = \begin{cases} 
0 & \text{if } \theta_1 < x_1 < \theta_2 \\
1 & \text{otherwise.}
\end{cases}
\]
Other node tests are possible:

- **Axis-aligned**

- **Linear:**
  \[
  \tilde{x} = [x, 1] \in \mathbb{R}^{d+1}, \psi \in \mathbb{R}^{d+1} \\
  t(x) = \begin{cases} 
  0 & \text{if } \psi^T \tilde{x} < \theta_1 \\
  1 & \text{otherwise.} 
  \end{cases} \\
  t(x) = \begin{cases} 
  0 & \text{if } \theta_1 < \psi^T \tilde{x} < \theta_2 \\
  1 & \text{otherwise.} 
  \end{cases}
  \]
Other node tests are possible:

- Axis-aligned
- Linear
- Conic section:

\[
\tilde{x} = [x, 1] \in \mathbb{R}^{d+1}, \psi \in \mathbb{R}^{(d+1) \times (d+1)}
\]

\[
t(x) = \begin{cases} 
0 & \text{if } \tilde{x}^T \psi \tilde{x} < \theta_1 \\
1 & \text{otherwise.}
\end{cases}
\]

\[
t(x) = \begin{cases} 
0 & \text{if } \theta_1 < \tilde{x}^T \psi \tilde{x} < \theta_2 \\
1 & \text{otherwise.}
\end{cases}
\]
Other node tests are possible:
- Axis-aligned
- Linear
- Conic section
- Other data spaces than $\mathbb{R}^d$
  - PolSAR: $\mathbb{C}^3$, $\mathbb{C}^3 \times 3$
  - Image patches: $\mathbb{R}^{n \times n}$
  - Non-scalar features, e.g. histograms, cardinal features such as pre-classification
- ...
From (Random) Decision Trees to Random Forests

Advantages

- Can deal with very heterogeneous data
  → Different, data-specific types of node tests
- Not prone to the curse of dimensionality
  → Each node only works on a very limited set of dimensions
- Very efficient
  → Each sample passes maximal H nodes (H = maximal height)
- Easy to implement
  → Binary trees are one of the most basic data structures
- Easy to interpret
  → Path through tree is a connected set of decision rules
- Well understood
  → Theoretical and practical implications of design decisions have been researched for more than 4 decades
From (Random) Decision Trees to Random Forests

Disadvantages

- Optimized by greedy algorithms
  → A chain of individually optimal decisions, might not lead to an overall optimum
- The optimal solution (i.e. decision boundary) might not be part of the model class (e.g. piece-wise linear and axis-aligned functions)
- Prone to overfitting
- Model capacity depends on amount of data
  → Few samples lead to small trees: Only few questions can be asked.
  → Many samples (might) lead to very high trees: Long processing times, large memory footprint.
From (Random) Decision Trees to Random Forests

### Disadvantages

- Optimized by greedy algorithms
  - A chain of individually optimal decisions, might not lead to an overall optimum

- The optimal solution (i.e. decision boundary) might not be part of the model class (e.g. piece-wise linear and axis-aligned functions)

- Prone to overfitting

- Model capacity depends on amount of data
  - Few samples lead to small trees: Only few questions can be asked.
  - Many samples (might) lead to very high trees: Long processing times, large memory footprint.

### How to

- keep (most) of the advantages
- getting rid of (most) disadvantages?
From (Random) Decision Trees to Random Forests
From (Random) Decision Trees to Random Forests
From (Random) Decision Trees to Random Forests
From (Random) Decision Trees to Random Forests
Random Forests

- Many (suboptimal) baselearners, i.e. decision trees
- Fusion of the individual output
- Minimization of the risk to use wrong model
- Extension of the model space
- Decreased dependence on initialization
- One name to rule them all
  - Bagged Decision Trees
  - Randomized Trees
  - Decision Forests
  - ERT, PERT, Rotation Forests, Hough Forests, Semantic Texton Forests, ...
Random Forests - Randomization through node tests

Before: \( t(x) = \begin{cases} 0 & \text{if } x_1 < \theta_1 \\ 1 & \text{otherwise.} \end{cases} \)

Now: More general

→ Concatenation of several functions with different tasks

\[
t_{\tau} : D \to \{0, 1\} \quad \tau \in T \equiv \text{Parameter set}
\]

\[
t_{\tau} = \xi \circ \psi \circ \phi
\]

\[
\phi : D \to \mathbb{R}^n \equiv \text{Implicit feature extraction}
\]

e.g. \( x \in \mathbb{R}^n : \phi : \mathbb{R}^n \to \mathbb{R}^2, x \mapsto (x_i, x_j)^T \)

\[
\psi : \mathbb{R}^n \to \mathbb{R} \equiv \text{Feature fusion}
\]

e.g. \( \phi(x) \in \mathbb{R}^2 : \psi : \mathbb{R}^2 \to \mathbb{R}, \phi(x) \mapsto [\psi_i, \psi_j] \cdot \phi(x) \)

\[
\xi : \mathbb{R} \to \{0, 1\} \equiv \text{Child node assignment}
\]

e.g. thresholding

Decision trees perform exhaustive search for optimal parameters \( \tau \) in \( T \)

Random Forests use random subset \( \tilde{T} \) (Note: \( |\tilde{T}| = 1 \) possible)
Random Forests - Randomization through Bagging

Given: Training set $D \subset \mathbb{D}$ with $|D| = N$ samples.

Bagging (Bootstrap aggregating):
1. Randomly sample $M$ data sets $D_m$ with replacement ($|D_m| = N$).
2. Train $M$ models where $m$-th model has only access to $m$-th dataset.
3. Average all models.

- Meta learning technique
- Works if small change in input data leads to large model variation
- Reduces variance (of final model), avoids overfitting.
- Leads to diverse decision trees, even if all other parameters are fixed
Random Forests - Key questions

- What kind of node tests?
  → For images, for other data spaces than $\mathbb{R}^n$

- How to select node tests?
  → How to measure good tests?

- What kind of target variables?
  → More than a single class label?

- How to limit model capacity (tree height, tree number)?
  → The more the better? What about overfitting?

- How to fuse tree decisions?
  → Whom to trust?

- How to interpret results?
  → Tree properties and visualization.


