Pattern Recognition - Summary

Friedrich-Alexander-Universität Erlangen-Nürnberg

Winter Term 2025/26

December 16, 2025

Contents

1 Norms and Norm Dependent Linear Regression

Norms and similarity measures are fundamental concepts in pattern recognition and machine learning. They are essential for defining distances between feature vectors, formulating optimization problems, and regularizing models.

1.1 Inner Products and Norms

1.1.1 Inner Product

The inner product provides a way to measure the angle and length of vectors.

Definition: Inner Product

The inner product of two vectors $\vec{x}, \vec{y} \in \mathbb{R}^d$ is defined as:

$$\langle \vec{x}, \vec{y} \rangle = \vec{x}^T \vec{y} = \sum_{i=1}^{d} x_i y_i \tag{1.1}$$

This concept extends to matrices. For two matrices $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m \times n}$, the inner product is defined using the trace operator:

$$\langle \mathbf{X}, \mathbf{Y} \rangle = \mathrm{tr}(\mathbf{X}^T \mathbf{Y}) = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} y_{ij} \tag{1.2}$$

1.1.2 Norms

A norm is a function $\|\cdot\| : \mathcal{X} \to \mathbb{R}$ that assigns a strictly positive length or size to a vector. To be a valid norm, the function must satisfy specific axioms.

- **Non-negativity:** $\forall \vec{x} : \|\vec{x}\| \geq 0$.
- **Definiteness:** $\|\vec{x}\| = 0$ if and only if $\vec{x} = \vec{0}$.
- **Homogeneity:** $\|a\vec{x}\| = |a| \cdot \|\vec{x}\|$ for any scalar $a \in \mathbb{R}$.
- **Triangle Inequality:** $\|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|$.

Common Vector Norms: The general L_p-norm ($p \geq 1$) is defined as:

$$\|\vec{x}\|_p = \left(\sum_{i=1}^{d} |x_i|^p \right)^{\frac{1}{p}} \tag{1.3}$$

Specific important instances include:

- **L_1-norm (Manhattan):** $\|\vec{x}\|_1 = \sum_{i=1}^{d} |x_i|$. This norm sums the absolute values of the components.
- **L_2-norm (Euclidean):** $\|\vec{x}\|_2 = \sqrt{\sum_{i=1}^{d} x_i^2}$. This corresponds to the standard straight-line distance.
- **L_∞-norm (Maximum):** $\|\vec{x}\|_\infty = \max_i |x_i|$. It takes the value of the largest component magnitude.

Note on the L_0-"norm": The L_0-norm counts the number of non-zero entries in a vector. Although often referred to as a norm, it is **not** a valid norm because it does not satisfy the homogeneity property.

Quadratic Norms and Mahalanobis Distance: Given a symmetric positive definite matrix $\mathbf{P}$, the quadratic L_P-norm is defined as:

$$\|\vec{x}\|_{\mathbf{P}} = \sqrt{\vec{x}^T\mathbf{P}\vec{x}} = \|\mathbf{P}^{\frac{1}{2}}\vec{x}\|_2 \tag{1.4}$$

If we choose $\mathbf{P} = \mathbf{\Sigma}^{-1}$ (the inverse covariance matrix), the distance induced by this norm corresponds to the **Mahalanobis distance**. This measure accounts for the correlation between variables in a dataset.

1.1.3 Unit Balls

The unit ball is the set of all vectors $\vec{x}$ such that $\|\vec{x}\| \leq 1$. The shape of the unit ball visualizes the geometric properties of the norm:

- **L_1-norm:** A diamond (rotated square) in 2D. This sharp geometry promotes sparsity in optimization.
- **L_2-norm:** A circle in 2D (or sphere in higher dimensions).
- **L_∞-norm:** A square (or hypercube).
- **L_p-norm ($p < 1$):** Concave shapes (star-like). Since the set is not convex, these do not form valid norms.

1.2 Norm Dependent Linear Regression

Linear regression aims to approximate a target vector $\vec{b}$ using a linear model $\mathbf{A}\vec{x}$. The goal is to minimize the residual vector $\vec{r} = \mathbf{A}\vec{x} - \vec{b}$. The general optimization problem depends on the chosen norm:

$$\hat{\vec{x}} = \text{argmin}_{\vec{x}} \|\mathbf{A}\vec{x} - \vec{b}\| \tag{1.5}$$

Different norms lead to different solutions and robustness properties.

1.2.1 Least-Squares Regression (L_2-Norm)

Minimizing the squared L_2-norm of the residuals is the most common approach:

$$\hat{\vec{x}} = \text{argmin}_{\vec{x}} \|\mathbf{A}\vec{x} - \vec{b}\|_2^2 = \text{argmin}_{\vec{x}}(\mathbf{A}\vec{x} - \vec{b})^T(\mathbf{A}\vec{x} - \vec{b}) \tag{1.6}$$

To solve this, we expand the term and calculate the derivative with respect to $\vec{x}$:

$$f(\vec{x}) = \vec{x}^T\mathbf{A}^T\mathbf{A}\vec{x} - 2\vec{b}^T\mathbf{A}\vec{x} + \vec{b}^T\vec{b} \tag{1.7}$$

$$\nabla_{\vec{x}} f(\vec{x}) = 2\mathbf{A}^T\mathbf{A}\vec{x} - 2\mathbf{A}^T\vec{b} \tag{1.8}$$

Setting the gradient to zero yields the closed-form solution (Normal Equations):

$$\hat{\vec{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\vec{b} \tag{1.9}$$

This solution assumes that the columns of $\mathbf{A}$ are linearly independent.

1.2.2 Chebyshev and Sum of Absolute Residuals

- **Chebyshev Regression (L_∞):** Minimizes the maximum individual residual ($\max_i |r_i|$). This optimization problem can be reformulated as a Linear Programming (LP) problem.
- **Sum of Absolute Residuals (L_1):** Minimizes $\sum |r_i|$. This can also be rewritten as an LP problem. The L_1-norm is significantly more **robust to outliers** than the L_2-norm, as large errors are not squared.

1.3 Regularization

In many cases, the problem is ill-posed or we want to enforce specific properties on the solution vector $\vec{x}$. This is achieved by adding a regularization term to the objective function.

1.3.1 Ridge Regression (L_2 Regularization)

We minimize the residual error plus the squared Euclidean length of the parameter vector:

$$\min_{\vec{x}} \|\mathbf{A}\vec{x} - \vec{b}\|_2^2 + \lambda \|\vec{x}\|_2^2 \tag{1.10}$$

Geometrically, this constrains the solution to lie within or near an L_2 unit ball. It prevents overfitting by keeping weights small.

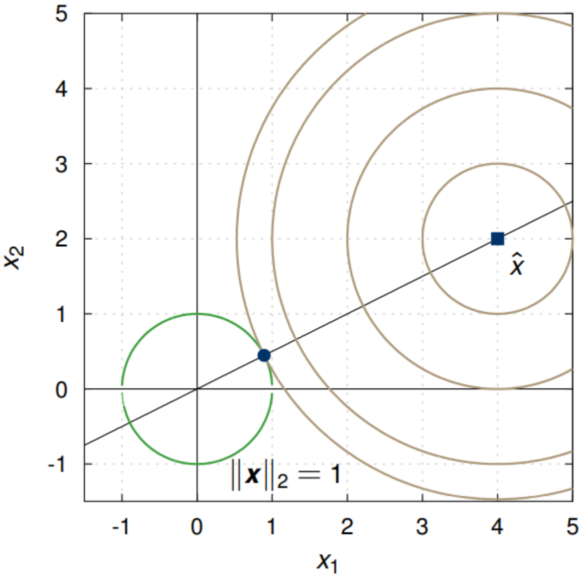

Figure 1: Geometric illustration of the L2-constrained optimization problem

1.3.2 Lasso (L_1 Regularization)

The Lasso (Least Absolute Shrinkage and Selection Operator) uses the L_1-norm for regularization:

$$\min_{\vec{x}} \|\mathbf{A}\vec{x} - \vec{b}\|_2^2 + \lambda \|\vec{x}\|_1 \tag{1.11}$$

Because the L_1 unit ball has "corners" (spikes) aligned with the axes, the solution often hits these corners. This forces some coefficients to become exactly zero (solution lies on one of the corrdinate axes), effectively performing **feature selection** and leading to **sparse solutions**. This property is exploited in **Compressed Sensing** to reconstruct signals from few measurements.

1.4 Penalty Functions

Instead of strictly using norms, we can define a general penalty function $\phi(r_i)$ for the residuals. The problem becomes:

$$\min \sum_{i=1}^{m} \phi(r_i) \quad \text{subject to} \quad \vec{r} = \mathbf{A}\vec{x} - \vec{b} \tag{1.12}$$

The choice of ϕ determines how the algorithm treats errors of different magnitudes.

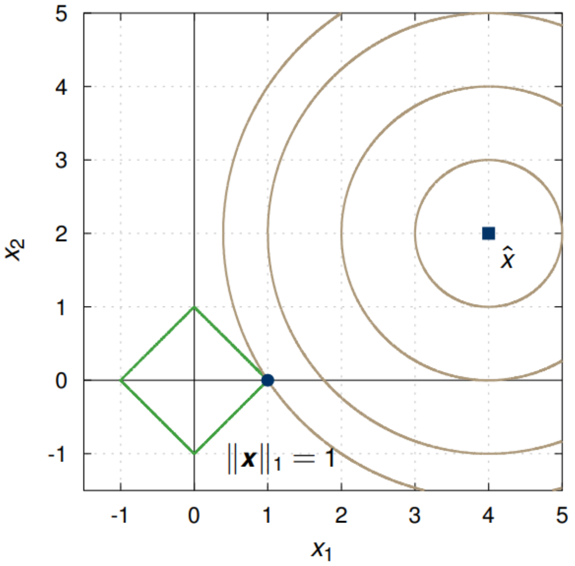

Figure 2: Geometric illustration of the L1-constrained optimization problem

- L_2**-Norm:** Parabolic shape. Heavy penalty for large errors (outliers).

$$\phi(r) = r^2 \tag{1.13}$$

- L_1**-Norm:** V-shape. Linear penalty for large errors, making it robust.

$$\phi(r) = |r| \tag{1.14}$$

- **Log Barrier:** The cost approaches infinity as residuals approach the limit a. This acts as a strict constraint (hard limit).

$$\phi_{Barrier}(r) = \begin{cases} -a^2 \log\left(1 - \left(\frac{r}{a}\right)^2\right) & \text{if } |r| \le a \\ \text{inf} & \text{otherwise} \end{cases} \tag{1.15}$$

- **Dead Zone:** Costs are zero for small residuals ($|r| \le a$) and linear afterwards. This ignores small noise ("insensitivity").

$$_{dz}(r) = \begin{cases} 0 & \text{if } |r| \le a \\ |r| - a & \text{otherwise} \end{cases} \tag{1.16}$$

- **Large Error:** Quadratic for small errors, but constant (a^2) for large errors. This completely caps the influence of outliers, but the function is non-convex, which makes optimization difficult.

- **Huber Loss:** A hybrid approach. Quadratic for small errors (differentiable at 0) and linear for large errors (robustness).

$$\phi_{\text{Huber}}(r) = \begin{cases} r^2 & \text{if } |r| \le a \\ a(2|r| - a) & \text{otherwise} \end{cases} \tag{1.17}$$

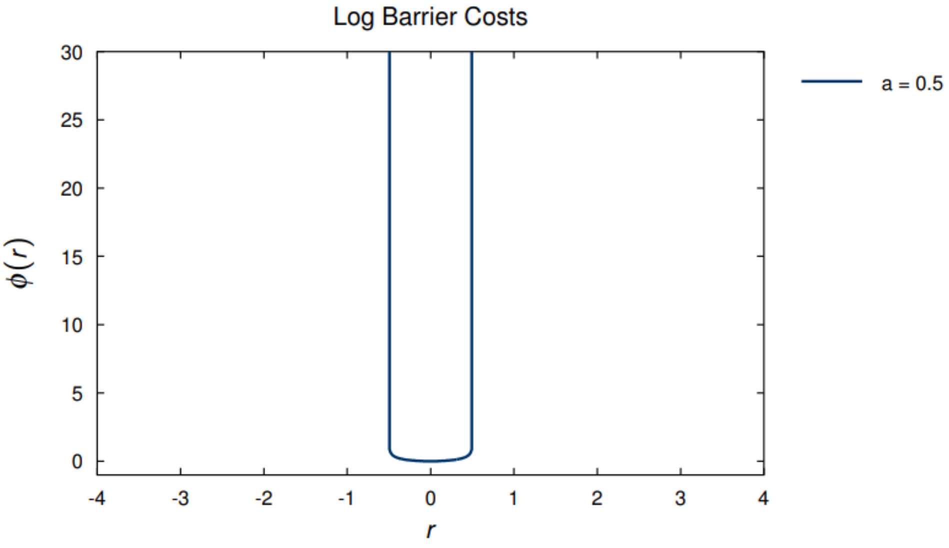

Figure 3: Geometric illustration of the Log-Barrier penalty function

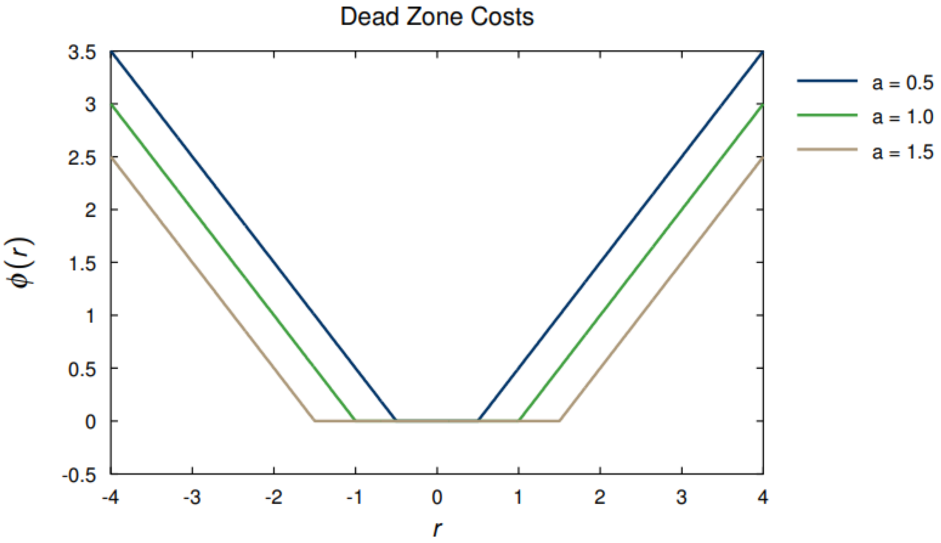

Figure 4: Geometric illustration of the Dead Zones penalty function

2 Neural Networks: From Perceptron to MLP

2.1 Rosenblatt's Perceptron

The Perceptron, introduced by Rosenblatt in 1957, is the fundamental building block of neural networks. It is a linear classifier designed to separate two classes.

2.1.1 The Linear Decision Boundary

We assume that the data is linearly separable and the class labels are $y \in \{+1, -1\}$. The decision rule is given by a linear function:

$$y^* = \text{sgn}(\vec{\alpha}^T \vec{x} + \alpha_0) \tag{2.1}$$

where $\vec{\alpha}$ represents the normal vector of the separating hyperplane and α_0 is the bias.

2.1.2 Objective Function

To find the optimal parameters $\vec{\alpha}$ and α_0, we do not simply count the classification errors (which would lead to a discrete and non-differentiable step function). Instead, we minimize the distance of **misclassified** feature vectors to the decision boundary.

Let $\mathcal{M}$ be the set of misclassified feature vectors. The perceptron criterion is defined as:

$$D(\alpha_0, \vec{\alpha}) = -\sum_{\vec{x}_i \in \mathcal{M}} y_i(\vec{\alpha}^T \vec{x}_i + \alpha_0) \tag{2.2}$$

If a point $\vec{x}_i$ is misclassified, the sign of y_i differs from $(\vec{\alpha}^T \vec{x}_i + \alpha_0)$, making the term inside the sum negative. The minus sign ensures that the total sum D is positive. We want to minimize this value.

$$\min_{\alpha_0, \vec{\alpha}} D(\alpha_0, \vec{\alpha}) = -\sum_{\vec{x}_i \in \mathcal{M}} y_i(\vec{\alpha}^T \vec{x}_i + \alpha_0) \tag{2.3}$$

Optimization Challenges:

- The set $\mathcal{M}$ changes in every iteration step.
- The cardinality $|\mathcal{M}|$ is a discrete variable, competing with the continuous parameters $\vec{\alpha}$.

2.1.3 The Perceptron Learning Algorithm

To minimize the objective function, we use **Stochastic Gradient Descent**. The gradients are:

$$\nabla_{\vec{\alpha}} D = -\sum_{\vec{x}_i \in \mathcal{M}} y_i \vec{x}_i, \quad \frac{\partial D}{\partial \alpha_0} = -\sum_{\vec{x}_i \in \mathcal{M}} y_i \tag{2.4}$$

Since we update after every single misclassification (online learning), the update rule for the $(k+1)$-th step, given a misclassified sample $(\vec{x}_i, y_i)$, is:

$$\begin{pmatrix} \alpha_0^{(k+1)} \\ \vec{\alpha}^{(k+1)} \end{pmatrix} = \begin{pmatrix} \alpha_0^{(k)} \\ \vec{\alpha}^{(k)} \end{pmatrix} + \lambda \begin{pmatrix} y_i \\ y_i \vec{x}_i \end{pmatrix} \tag{2.5}$$

where λ is the learning rate (usually set to 1).

Algorithm 1 Perceptron Learning Algorithm

```
Input: Training data $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$
Initialize: $k = 0$, $\alpha_0^{(0)} = 0$, $\vec{\alpha}^{(0)} = \vec{0}$
repeat
    Select a pair $(\vec{x}_i, y_i)$ from the training set $S$
    if $y_i(\vec{\alpha}^{(k)T} \vec{x}_i + \alpha_0^{(k)}) \leq 0$ then          ▷ Misclassified
        $\vec{\alpha}^{(k+1)} \leftarrow \vec{\alpha}^{(k)} + y_i \vec{x}_i$
        $\alpha_0^{(k+1)} \leftarrow \alpha_0^{(k)} + y_i$
        $k \leftarrow k + 1$
    end if
until all samples are correctly classified
Output: $\vec{\alpha}^{(k)}$, $\alpha_0^{(k)}$
```

2.1.4 Convergence (Novikoff's Theorem)

If the data is linearly separable, the algorithm is guaranteed to converge.

Theorem: Convergence of Rosenblatt's Perceptron

Assume there exists an optimal solution $\vec{\alpha}^*$ (with $\|\vec{\alpha}^*\| = 1$) and a margin $\rho > 0$ such that $y_i(\vec{\alpha}^{*T}\vec{x}_i + \alpha_0^*) \geq \rho$ for all samples. Let $M = \max_i \|\vec{x}_i\|_2$. The number of updates k is bounded by:

$$k \leq \frac{(\alpha_0^{*2} + 1)(1 + M^2)}{\rho^2} \tag{2.6}$$

Proof Sketch: The proof relies on two observations:

1. The inner product between the current weight vector and the optimal solution grows linearly with each update ($(\vec{\alpha}^{(k)})^T\vec{\alpha}^* \geq k\rho$). This implies alignment.
2. The squared norm of the weight vector grows at most linearly ($\|\vec{\alpha}^{(k)}\|^2 \leq k(1 + M^2)$).
3. Combining these using the Cauchy-Schwarz inequality yields the upper bound.

Result: The number of iterations does *not* depend on the dimension of the feature space, but on the geometric margin ρ.

2.2 Multi-Layer Perceptrons (MLP)

Since the single Perceptron can only solve linearly separable problems, we extend the concept to Multi-Layer Perceptrons (MLPs).

2.2.1 Physiological Motivation & Topology

The MLP is inspired by biological neural networks:

- **Dendrites:** Input channels (feature vectors).
- **Synapses:** Weights (strength of connection).
- **Axon Hillock:** Summation and activation threshold.

An MLP consists of an **Input Layer**, one or more **Hidden Layers**, and an **Output Layer**.

- $w_{ij}^{(l)}$: Weight from neuron i in layer $(l-1)$ to neuron j in layer (l).
- $net_j^{(l)}$: The weighted sum input for neuron j in layer l.
- $y_j^{(l)}$: The output of neuron j after applying the activation function.

2.2.2 Activation Functions

To introduce non-linearity, we apply an activation function $f(\cdot)$ to the net input:

$$y_j^{(l)} = f(net_j^{(l)}) = f\left(\sum_i w_{ij}^{(l)} y_i^{(l-1)} - w_{0j}^{(l)}\right) \tag{2.7}$$

Common functions are:

- **Sigmoid:** $f(\sigma) = \frac{1}{1+e^{-\sigma}}$
- **Tanh:** $f(\sigma) = \tanh(\sigma)$
- **ReLU:** $f(\sigma) = \max(0, \sigma)$

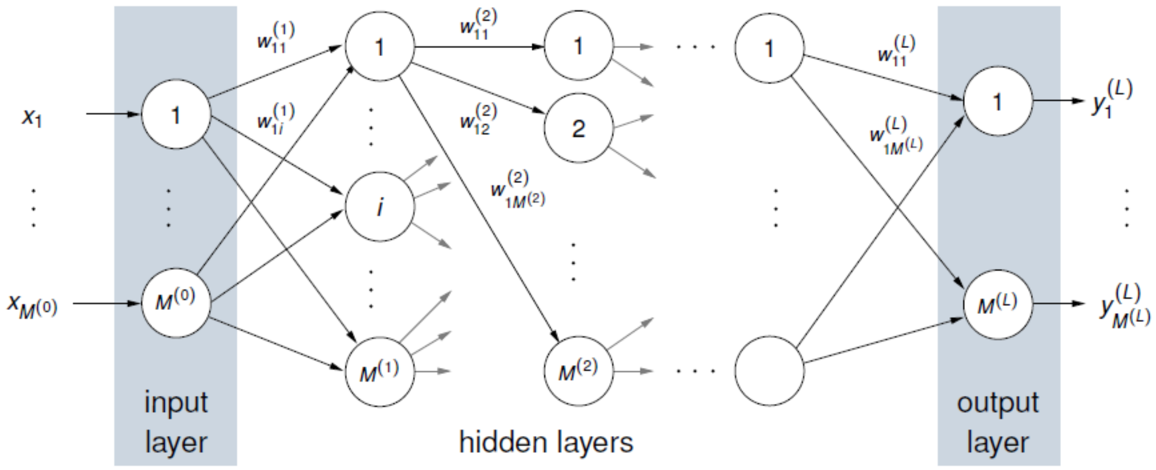

Figure 5: Structure of a Multi-Layer Perceptron (MLP)

2.3 Backpropagation Algorithm

We train the MLP using **Gradient Descent** to minimize a loss function, typically the Mean Squared Error (MSE):

$$\epsilon_{MSE}(w) = \frac{1}{2} \sum_{k=1}^{M^{(L)}} (t_k - y_k^{(L)})^2 \tag{2.8}$$

where t_k is the target value and $y_k^{(L)}$ is the actual output of the last layer L. The update rule is:

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial \epsilon}{\partial w_{ij}^{(l)}} \tag{2.9}$$

2.3.1 Derivation of the Gradients

We calculate the gradient using the chain rule, starting from the output layer and propagating the error backwards ("Backpropagation").

1. Output Layer ($l = L$): We define the **sensitivity** $\delta_k^{(L)}$ as the derivative of the error with respect to the net input. For the output layer, this depends directly on the target values t_k.

The gradient for the weights $w_{jk}^{(L)}$ connecting the last hidden layer to the output layer is given by:

$$\frac{\partial \epsilon_{MSE}}{\partial w_{jk}^{(L)}} = \frac{\partial \epsilon_{MSE}}{\partial net_k^{(L)}} \cdot \frac{\partial net_k^{(L)}}{\partial w_{jk}^{(L)}} = -\delta_k^{(L)} \cdot y_j^{(L-1)} \tag{2.10}$$

with the sensitivity:

$$\delta_k^{(L)} = -\frac{\partial \epsilon_{MSE}}{\partial net_k^{(L)}} = \frac{\partial \epsilon_{MSE}}{\partial y_k^{(L)}} \cdot \frac{\partial y_k^{(L)}}{\partial net_k^{(L)}} = (t_k - y_k^{(L)}) \cdot f'(net_k^{(L)}) \tag{2.11}$$

2. Hidden Layers ($l < L$): For a hidden layer l, we do not have target values. The error ϵ depends on the output $y_j^{(l)}$ of neuron j only via the neurons k in the **next layer** $(l + 1)$ that it is connected to. We apply the chain rule to derive the sensitivity $\delta_j^{(l)}$:

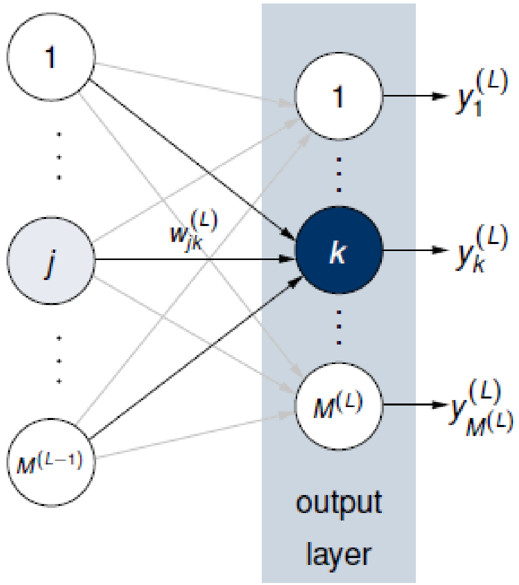

Figure 6: Backpropagation at the Output Layer: The error signal is computed directly from the output $y_k^{(L)}$ and the target t_k.

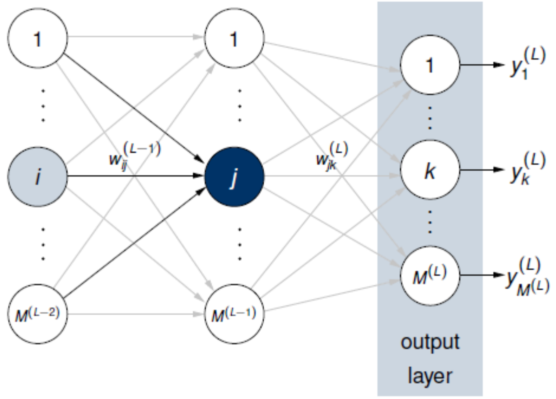

Figure 7: Backpropagation at a Hidden Layer: The error signal δ_j is derived from the weighted sum of errors δ_k from the subsequent layer.

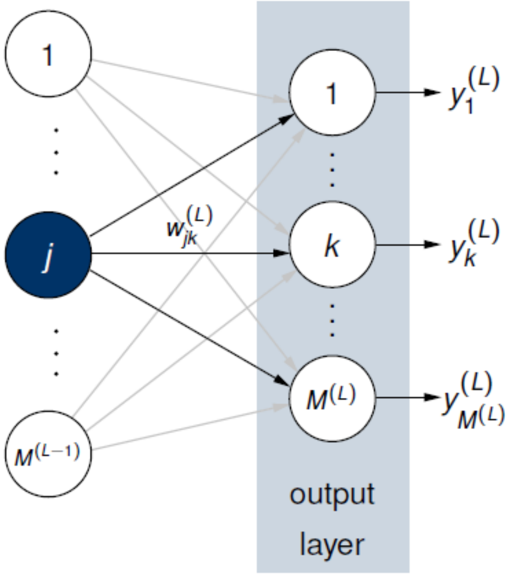

Figure 8: Backpropagation at a Hidden Layer: The sensitivities from the next layer are combined to compute the sensitivity for the current layer.

$$\delta_j^{(l)} = -\frac{\partial \epsilon_{MSE}}{\partial net_j^{(l)}} = -\frac{\partial \epsilon_{MSE}}{\partial y_j^{(l)}} \cdot \frac{\partial y_j^{(l)}}{\partial net_j^{(l)}} \tag{2.12}$$

The second term is simply the derivative of the activation function, $f'(net_j^{(l)})$. The first term, $\frac{\partial \epsilon}{\partial y_j^{(l)}}$, requires summing the contributions from all neurons k in the subsequent layer $(l+1)$ (since $y_j^{(l)}$ feeds into all of them):

$$\frac{\partial \epsilon_{MSE}}{\partial y_j^{(l)}} = \sum_{k=1}^{M^{(l+1)}} \left(\frac{\partial \epsilon_{MSE}}{\partial net_k^{(l+1)}} \cdot \frac{\partial net_k^{(l+1)}}{\partial y_j^{(l)}} \right) \tag{2.13}$$

Here, we identify the terms:

- $\frac{\partial \epsilon_{MSE}}{\partial net_k^{(l+1)}} = -\delta_k^{(l+1)}$ (Sensitivity of the next layer)
- $\frac{\partial net_k^{(l+1)}}{\partial y_j^{(l)}} = w_{jk}^{(l+1)}$ (The weight connecting j to k)

Substituting these back yields the recursive formula for the sensitivity:

$$\delta_j^{(l)} = f'(net_j^{(l)}) \cdot \sum_{k=1}^{M^{(l+1)}} \delta_k^{(l+1)} w_{jk}^{(l+1)} \tag{2.14}$$

Interpretation: The error $\delta_j^{(l)}$ is the weighted sum of the errors $\delta_k^{(l+1)}$ from the layer above, scaled by the local derivative f'. This is the essence of "propagating" the error back.

Finally, the weight update rule remains structurally consistent:

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial \epsilon_{MSE}}{\partial w_{ij}^{(l)}} = \eta \cdot \delta_j^{(l)} \cdot y_i^{(l-1)} \tag{2.15}$$

2.3.2 Matrix Notation

The entire forward pass of a fully connected network can be expressed compactly using matrix multiplication. For a 3-layer network:

$$\hat{y} = f_3(W_3 \cdot f_2(W_2 \cdot f_1(W_1 \vec{x}))) \tag{2.16}$$

The gradient computation (Backpropagation) then corresponds to a series of vector-matrix multiplications involving the Jacobian matrices of the layers.

3 Optimization

Optimization is crucial for many solutions in pattern recognition (e.g., training classifiers). We distinguish between unconstrained and constrained optimization problems.

3.1 Convexity

A function $f : \mathbb{R}^n \to \mathbb{R}$ is called **convex** if for all $\vec{x}, \vec{y} \in \mathbb{R}^n$ and for all $\theta \in [0, 1]$, the following condition holds:

Definition: Convexity & Concavity

A function $f : \mathbb{R}^d \to \mathbb{R}$ is **convex** if the domain $\mathrm{dom}(f)$ is a convex set and if $\forall \vec{x}, \vec{y} \in \mathrm{dom}(f)$, and θ with $0 \leq \theta \leq 1$, we have:

$$f(\theta\vec{x} + (1-\theta)\vec{y}) \leq \theta f(\vec{x}) + (1-\theta) f(\vec{y}) \tag{3.1}$$

A function is **concave** if $-f$ is convex.

Implication: For convex functions, any local minimum is also a global minimum. This property is particularly useful because gradient-based methods won't get stuck in suboptimal local minima.

3.2 Unconstrained Optimization

Here, we aim to find the minimum of a function $f(\vec{x})$ without any restrictions on $\vec{x}$. Typically, we assume f is twice differentiable and convex.

$$\vec{x}^* = \mathrm{argmin}_{\vec{x}} f(\vec{x}) \tag{3.2}$$

A necessary and sufficient condition for the minimum is the zero-crossing of the gradient:

$$\nabla f(\vec{x}^*) = 0 \tag{3.3}$$

Since a closed-form solution is often impossible, we use iterative approaches:

$$\begin{aligned} \text{initialization:} \quad & \vec{x}^{(0)} \\ \text{iteration step:} \quad & \vec{x}^{(k+1)} = \vec{x}^{(k)} + t^{(k)} \Delta\vec{x}^{(k)} \end{aligned}$$

where $\Delta\vec{x}^{(k)}$ is the **search direction** and $t^{(k)}$ is the **step size**.

3.2.1 Finding a Suitable Step Size (Line Search)

Choosing the correct step size $t^{(k)}$ is crucial:

- **Too small:** Convergence is extremely slow.
- **Too large:** The algorithm might overshoot the minimum or diverge.

Instead of finding the exact optimal t (which is computationally expensive), we use **inexact line search** methods like **Backtracking Line Search** (Armijo-Goldstein).

The goal is to find a step size t that, that puts us below the red line, ensuring that the function value decreases sufficiently (not just barely) relative to the step size.

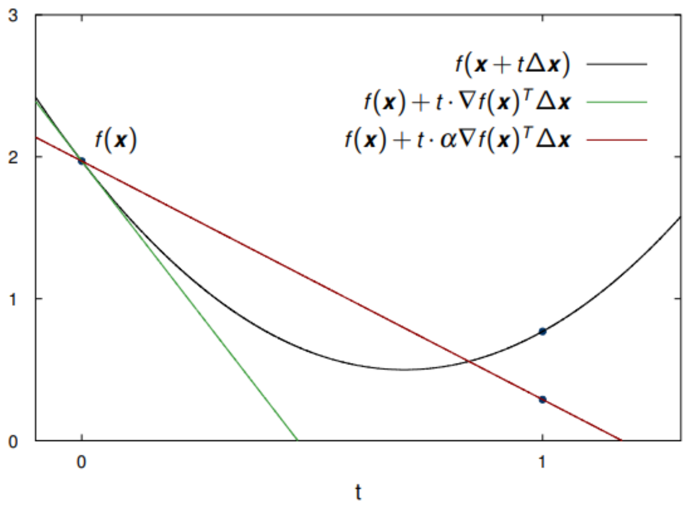

Figure 9: Backtracking line search

Armijo-Goldstein Condition (Backtracking)

We start with a large step size ($t = 1$) and iteratively reduce it ($t := \beta t$) until the function value decreases sufficiently. The condition is:

$$f(\vec{x} + t\Delta\vec{x}) \leq f(\vec{x}) + \alpha t \nabla f(\vec{x})^T \Delta\vec{x} \tag{3.4}$$

Where $\alpha \in (0, 0.5)$ defines the required "steepness" of the descent.

3.2.2 Gradient Descent

The most natural choice for the search direction is the direction of steepest descent, which is the negative gradient.

$$\Delta\vec{x}^{(k)} = -\nabla f(\vec{x}^{(k)}) \tag{3.5}$$

Algorithm: Gradient Descent

1. Set direction: $\Delta\vec{x}^{(k)} = -\nabla f(\vec{x}^{(k)})$
2. Line search (find optimal step size t):

   $$t^{(k)} = \text{argmin}_{t\geq 0} f(\vec{x}^{(k)} + t\Delta\vec{x}^{(k)})$$

   (Usually approximated via **Backtracking Line Search** / **Armijo-Goldstein**).
3. Update: $\vec{x}^{(k+1)} = \vec{x}^{(k)} + t^{(k)}\Delta\vec{x}^{(k)}$
4. Repeat until convergence ($\|\vec{x}^{(k)} - \vec{x}^{(k-1)}\| < \epsilon$).

3.2.3 Normalized Steepest Descent (General Norms)

Ideally, we want the direction that gives the largest decrease in the linear approximation of f. This depends on the chosen norm $\|\cdot\|$.

$$\Delta\vec{x} = \text{argmin}_{\vec{u}}\{\nabla f(\vec{x})^T \vec{u} \mid \|\vec{u}\| = 1\} \tag{3.6}$$

- **L_2-Norm:** Direction is $-\nabla f(\vec{x})$ (Standard Gradient Descent). Unit ball is a sphere.

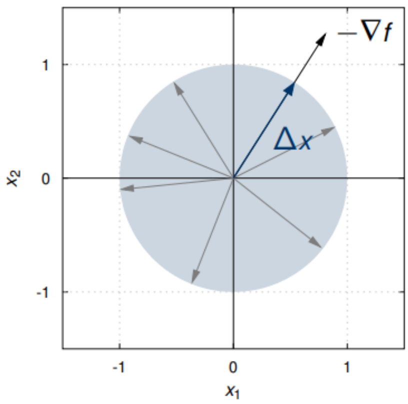

Figure 10: Unit ball in L_2 norm

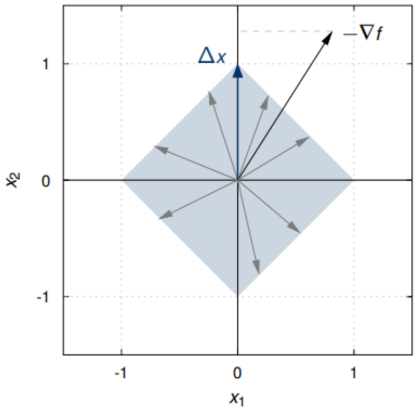

Figure 11: Unit ball in L_1 norm

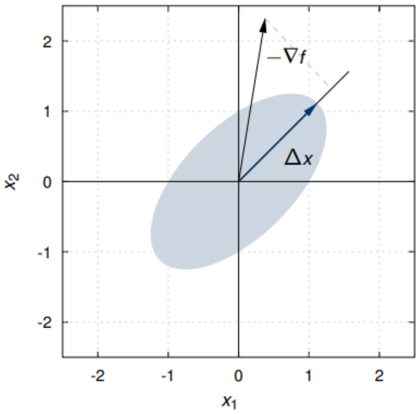

Figure 12: Unit ball in L_P norm

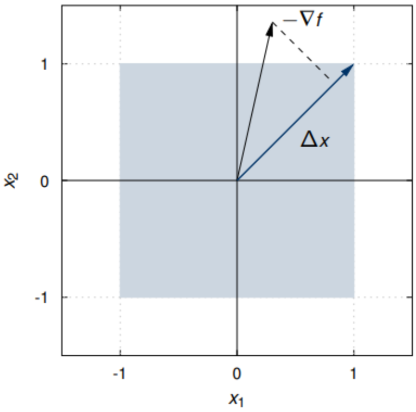

Figure 13: Unit ball in L_∞ norm

- L_1**-Norm:** Direction is along the coordinate axis with the largest absolute gradient value (Coordinate Descent). Unit ball is a diamond.
- L_∞**-Norm:** Direction points towards the corners of the unit hypercube (e.g., vector with entries ±1). Unit ball is a square/cube.
- L_P**-Norm (Quadratic Norm):** Defined by a positive definite matrix $\mathbf{P}$ as $\|\vec{u}\|_{\mathbf{P}} = \sqrt{\vec{u}^T\mathbf{P}\vec{u}}$.

  $$\Delta\vec{x} = -\mathbf{P}^{-1}\nabla f(\vec{x}) \tag{3.7}$$

  This transforms the space to make the contours spherical before taking the gradient step (similar to whitening in LDA). The update direction aligns with the largest principal component in the transformed space.

  Crucial Connection: If we set $\mathbf{P} = \nabla^2 f(\vec{x})$ (the Hessian), we effectively perform steepest descent in the local curvature norm, which yields **Newton's Method**.

3.2.4 Newton's Method

Newton's method finds the search direction by approximating the function $f(\vec{x})$ locally using a **second-order Taylor polynomial**:

$$f(\vec{x} + \Delta\vec{x}) \approx f(\vec{x}) + \nabla f(\vec{x})^T \Delta\vec{x} + \frac{1}{2}\Delta\vec{x}^T \nabla^2 f(\vec{x}) \Delta\vec{x} \tag{3.8}$$

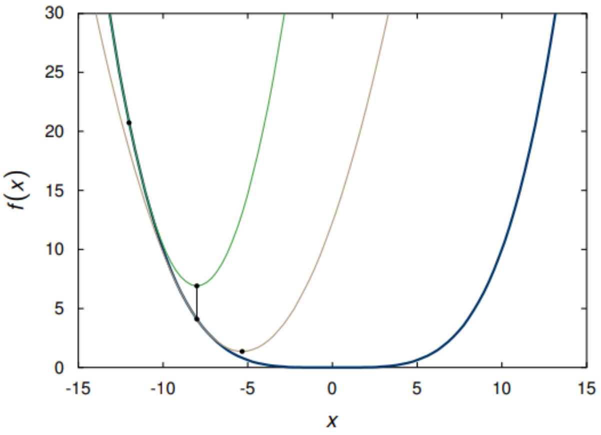

Figure 14: Newtons Method using 2nd order Taylor approximation

We find the optimal step $\Delta\vec{x}$ by setting the derivative of this approximation to zero:

$$\nabla_{\Delta\vec{x}}(\dots) = \nabla f(\vec{x}) + \nabla^2 f(\vec{x})\Delta\vec{x} = 0$$
$$\Rightarrow \quad \Delta\vec{x} = -(\nabla^2 f(\vec{x}))^{-1}\nabla f(\vec{x})$$

Key Difference: While Gradient Descent assumes the local geometry is a sphere (linear approximation), Newton's Method accounts for the local **curvature** (ellipsoidal geometry) given by the Hessian $\nabla^2 f(\vec{x})$.

3.3 Constrained Optimization

We consider the primal optimization problem with constraints:

$$\begin{aligned} \text{minimize} \quad & f_0(\vec{x}) \\ \text{subject to} \quad & f_i(\vec{x}) \leq 0, \quad i = 1, \dots, m \\ & h_i(\vec{x}) = 0, \quad i = 1, \dots, p \end{aligned}$$

Note: f_0 is not required to be convex yet.

3.3.1 The Lagrangian & Dual Function

To handle constraints, we define the **Lagrangian** L:

$$L(\vec{x}, \vec{\lambda}, \vec{\nu}) = f_0(\vec{x}) + \sum_{i=1}^{m} \lambda_i f_i(\vec{x}) + \sum_{i=1}^{p} \nu_i h_i(\vec{x}) \tag{3.9}$$

We formulate the **Lagrange dual function** $g(\vec{\lambda}, \vec{\nu})$ by minimizing L over $\vec{x}$:

$$g(\vec{\lambda}, \vec{\nu}) = \inf_{\vec{x}} L(\vec{x}, \vec{\lambda}, \vec{\nu}) \tag{3.10}$$

Properties of the Dual Function

- $g(\vec{\lambda}, \vec{\nu})$ is **always concave**, even if the primal problem is not convex.
- It provides a **lower bound** on the optimal primal value p^*: $g(\vec{\lambda}, \vec{\nu}) \leq p^*$.

This leads to the **Dual Problem**: Find the best lower bound by maximizing g.

$$\text{maximize } g(\vec{\lambda}, \vec{\nu}) \quad \text{subject to } \vec{\lambda} \succeq 0$$

3.3.2 Strong Duality & Slater's Condition

- **Weak Duality:** $d^* \leq p^*$ (Always true). The difference $p^* - d^*$ is the *duality gap*.
- **Strong Duality:** $d^* = p^*$ (Gap is zero). This allows solving the dual problem to find the primal solution.

Strong duality is guaranteed if **Slater's Condition** holds:

Theorem: Slater's Condition

For a **convex** optimization problem, strong duality holds if there exists a strictly feasible point $\vec{x}$ such that:

$$f_i(\vec{x}) < 0 \quad \forall i = 1, \ldots, m \quad \text{and} \quad A\vec{x} = \vec{b} \tag{3.11}$$

Refinement: If constraints are affine (linear), touching the boundary ($f_i(\vec{x}) \leq 0$) is allowed. Only non-linear constraints require strict inequality.

3.3.3 Karush-Kuhn-Tucker (KKT) Conditions

If strong duality holds, any optimal pair $(\vec{x}^*, \vec{\lambda}^*, \vec{\nu}^*)$ **must** satisfy the KKT conditions:

1. **Primal Feasibility:**
$$f_i(\vec{x}^*) \leq 0, \quad h_i(\vec{x}^*) = 0$$
2. **Dual Feasibility:**
$$\vec{\lambda}^* \succeq 0$$
3. **Complementary Slackness:** (Crucial for SVMs!)
$$\lambda_i^* \cdot f_i(\vec{x}^*) = 0$$
Meaning: Either a constraint is active ($f_i(\vec{x}) = 0$) or its multiplier is zero ($\lambda_i = 0$).
4. **Stationarity:** Gradient of Lagrangian is zero.
$$\nabla f_0(\vec{x}^*) + \sum \lambda_i^* \nabla f_i(\vec{x}^*) + \sum \nu_i^* \nabla h_i(\vec{x}^*) = 0$$

Conclusion: For convex problems with strong duality, KKT conditions are necessary and sufficient for optimality. Finding a point that satisfies them means we found the global optimum.

4 Support Vector Machines (SVM)

Just like other classifiers (Neural Nets, Nearest Neighbor, etc.), the goal of SVMs is to draw a linear line (decision boundary) to separate classes. But instead of drawing any sufficient line to separate the classes, SVMs aim to find a unique decision boundary that **maximizes the margin (distance)** between each class. The solution to this problem is unique and depends only on the features that are close to the decision boundary.

4.1 Hard Margin Problems

The hard margin SVM needs linearly seperable classes. Lets assume there is an affine function defined as:

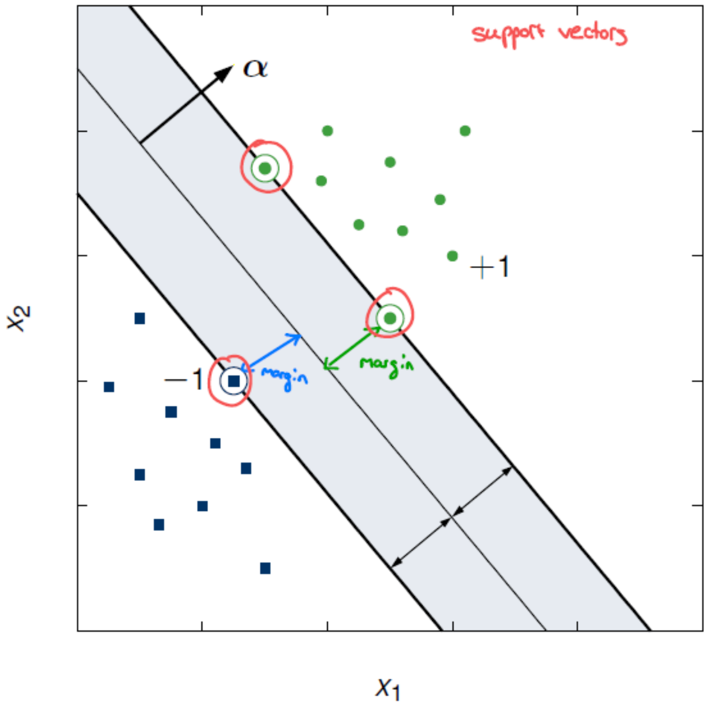

Figure 15: Hard margin SVM

$$f(x) = \vec{\alpha}^T x + \alpha_0 \tag{4.1}$$

Where $\vec{\alpha}$ is the normal vector to the decision boundary and α_0 is some sort of bias. For any point x on the decision boundary, it holds that $f(x) = 0$.
There are three major points we need to think about to end up with a nice optimization problem:

1. Introduce margin constraints:
We need to ensure that all points are classified correctly and therefore lie outside the margin. Therefore, we introduce the following constraints:

- For points of class +1: $f(x) \geq 1$
- For points of class −1: $f(x) \leq -1$

This means that for a sample point x_i with the label $y_i = +1$, it holds that $f(x_i) \geq 1$. Similarly, for a sample point x_j with the label $y_j = -1$, it holds that $f(x_j) \leq -1$. These two constraints can be combined into one single constraint ($y \in \{+1, -1\}$):

$$y_i f(x_i) - 1 = y_i(\vec{\alpha}^T x_i + \alpha_0) - 1 \geq 0 \quad \forall i \tag{4.2}$$

2. Define the margin:
The margin is defined as the distance between the decision boundary and the closest points from either class. To compute the margin width, we take a sample from each class that lies exactly on the margin (i.e., satisfies $y_i f(x_i) - 1 = 0 \quad \forall i$) and subtract them from each other. When we project the resulting vector onto tho normalized normal vector of the hyperplane, we get the margin width:

$$\text{width} = \frac{\vec{\alpha}}{\|\vec{\alpha}\|_2} \cdot (\vec{x}_{y=+1} - \vec{x}_{y=-1}) \tag{4.3}$$

Now we multiply this out:

$$\text{width} = \frac{1}{\|\vec{\alpha}\|_2} (\vec{\alpha}^T \vec{x}_{y=+1} - \vec{\alpha}^T \vec{x}_{y=-1}) \tag{4.4}$$

We know from our margin constraints defined in step 1 that for support vectors (points on the margin), the inequality becomes an equality:

- For the positive support vector $\vec{x}_{y=+1}$:

$$\vec{\alpha}^T \vec{x}_{y=+1} + \alpha_0 = 1 \quad \Rightarrow \quad \vec{\alpha}^T \vec{x}_{y=+1} = 1 - \alpha_0$$

- For the negative support vector $\vec{x}_{y=-1}$:

$$\vec{\alpha}^T \vec{x}_{y=-1} + \alpha_0 = -1 \quad \Rightarrow \quad \vec{\alpha}^T \vec{x}_{y=-1} = -1 - \alpha_0$$

Substituting these expressions back into the width equation:

$$\begin{aligned} \text{width} &= \frac{1}{\|\vec{\alpha}\|_2}((1 - \alpha_0) - (-1 - \alpha_0)) && (4.5) \\ &= \frac{1}{\|\vec{\alpha}\|_2}(1 - \alpha_0 + 1 + \alpha_0) && (4.6) \\ &= \frac{2}{\|\vec{\alpha}\|_2} && (4.7) \end{aligned}$$

3. Minimize the norm:
Since we want to **maximize** the margin width $\frac{2}{\|\vec{\alpha}\|_2}$, this is mathematically equivalent to **minimizing** the length of the normal vector $\|\vec{\alpha}\|_2$. For mathematical convenience (to make derivatives easier later), we minimize the squared norm:

Primal Optimization Problem (Hard Margin)

$$\text{minimize} \quad \frac{1}{2}\|\vec{\alpha}\|_2^2 \quad \text{subject to} \quad y_i(\vec{\alpha}^T x_i + \alpha_0) \geq 1 \quad \forall i$$

4.2 Soft Margin Problems

In real world applications, data is often not perfectly linearly separable. The Soft Margin SVM relaxes the hard margin constraints by allowing some points to violate the margin or even be misclassified.

Therefore, we introduce slack variables $\xi_i \geq 0$ for each training sample x_i, which measure the degree of misclassification (**Hinge Loss**):

- $\xi_i = 0$: point is correctly classified and outside or on the margin
- $0 < \xi_i \leq 1$: point is inside the margin but still correctly classified

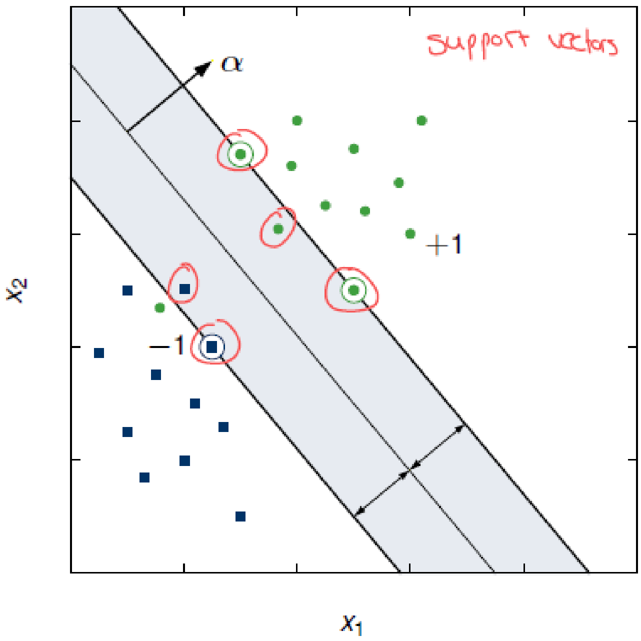

Figure 16: Soft margin SVM

- $\xi_i > 1$: point is misclassified

The margin constraints are now relaxed to:

$$y_i(\vec{\alpha}^T x_i + \alpha_0) \geq 1 - \xi_i \quad \forall i \tag{4.8}$$

The primal optimization problem becomes:

Primal Optimization Problem (Soft Margin)

$$\text{minimize} \quad \frac{1}{2}\|\vec{\alpha}\|_2^2 + \mu \sum_{i=1}^{n} \xi_i \quad \text{subject to} \quad -(y_i(\vec{\alpha}^T x_i + \alpha_0) - 1 + \xi_i) \leq 0, \quad -\xi_i \leq 0 \quad \forall i$$

where $\mu > 0$ is a hyperparameter that controls the trade-off between maximizing the margin and minimizing the classification error.

4.3 Dual Representation

The primal problem minimizes with respect to $\vec{\alpha}$ and α_0. However, solving the **Lagrange Dual Problem** is often more powerful. It reveals that the solution depends *only* on the inner products of the data points, which is the key to the Kernel Trick.

4.3.1 Derivation from the Lagrangian

For simplicity, we derive the dual only for the hard margin case here. In the soft margin case, the derivation is similar but includes additional terms for the slack variables. We start with the

Lagrangian of the hard margin problem (using multipliers $\lambda_i \geq 0$):

$$L(\vec{\alpha}, \alpha_0, \vec{\lambda}) = \underbrace{\frac{1}{2}\|\vec{\alpha}\|_2^2}_{\text{primal problem } f_0(x)} \underbrace{- \sum_{i=1}^{m} \lambda_i [y_i(\vec{\alpha}^T \vec{x}_i + \alpha_0) - 1]}_{\text{constraints } f_i(x)} \tag{4.9}$$

To find the dual, we minimize L with respect to the primal variables $\vec{\alpha}$ and α_0 (setting gradients to zero):

$$\nabla_{\vec{\alpha}} L = \vec{\alpha} - \sum_{i=1}^{m} \lambda_i y_i \vec{x}_i = 0 \quad \Rightarrow \quad \tilde{\alpha} = \sum_{\mathbf{i=1}}^{\mathbf{m}} \boldsymbol{\lambda_i y_i \tilde{x}_i} \tag{4.10}$$

$$\frac{\partial L}{\partial \alpha_0} = - \sum_{i=1}^{m} \lambda_i y_i = 0 \quad \Rightarrow \quad \sum_{i=1}^{m} \lambda_i y_i = 0 \tag{4.11}$$

Substituting (4.10) back into the Lagrangian eliminates $\vec{\alpha}$ and yields the **Dual Objective Function**, which depends only on $\vec{\lambda}$:

Dual Optimization Problem

$$\text{maximize} \quad \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j (\vec{x}_i^T \vec{x}_j)$$

Subject to:

$$\lambda_i \geq 0 \quad \forall i, \quad \text{and} \quad \sum_{i=1}^{m} \lambda_i y_i = 0$$

4.3.2 From Dual Variables to the Decision Boundary

Once we have solved the Dual Problem and found the optimal Lagrange multipliers λ_i^*, we need to reconstruct the decision boundary. But which λ_i are actually relevant?

This is answered by the **Complementary Slackness** condition (KKT condition 3):

$$\lambda_i \cdot \underbrace{(y_i \cdot (\vec{\alpha}^T \vec{x}_i + \alpha_0) - 1)}_{f_i(\vec{\alpha}, \alpha_0)} = 0 \tag{4.12}$$

Since the product must be zero, we have two cases for every data point i:

- **Case 1:** $\lambda_i = 0$. The point is correctly classified and lies somewhere "safe" (not on the margin). These points effectively vanish from our solution.
- **Case 2:** $\lambda_i > 0$. Then the bracket term *must* be zero:

  $$y_i(\vec{\alpha}^T \vec{x}_i + \alpha_0) = 1 \tag{4.13}$$

  These points lie exactly on the margin boundaries. We call them **Support Vectors**.

Conclusion 1 (Sparsity): The weight vector $\vec{\alpha}$ (see 4.3.1) depends *only* on the support vectors (where $\lambda_i > 0$):

$$\vec{\alpha} = \sum_{i=1}^{m} \lambda_i y_i \vec{x}_i = \sum_{i \in SV} \lambda_i y_i \vec{x}_i \tag{4.14}$$

Conclusion 2 (The Decision Rule): We can now write the classification function for a new point $\vec{u}$ purely in terms of training samples and their inner products:

$$f(\vec{u}) = \vec{\alpha}^T\vec{u} + \alpha_0 = \underbrace{\left(\sum_{i=1}^{m} \lambda_i y_i \vec{x}_i\right)^T}_{\vec{\alpha}^T} \vec{u} + \alpha_0 = \sum_{i=1}^{m} \lambda_i y_i (\vec{x}_i^T \vec{u}) + \alpha_0 \tag{4.15}$$

This form is crucial because it allows us to apply the **Kernel Trick** in the next section.

4.4 Feature Transform & The Kernel Trick

Both hard and soft margin SVMs can only generate a **linear decision boundary**. This has serious limitations:

- Non-linearly separable data cannot be classified.
- Noisy data can cause problems.
- The formulation deals with vectorial data only.

4.4.1 Mapping to a Higher Dimensional Space

To solve this, we map the data into a richer, higher-dimensional feature space using a non-linear transformation Φ:

$$\Phi : \mathbb{R}^d \to \mathbb{R}^D \quad (D \gg d) \tag{4.16}$$

such that the features $\Phi(\vec{x}_i)$ become linearly separable in that new space.

Example (Quadratic Decision Boundary): Assume a 2D decision boundary defined by a quadratic function:

$$f(\vec{x}) = \alpha_0 + \alpha_1 x_1^2 + \alpha_2 x_2^2 + \alpha_3 x_1 x_2 + \alpha_4 x_1 + \alpha_5 x_2$$

This is non-linear in the original space $\vec{x} = (x_1, x_2)^T$. However, we can define a transformation $\Phi(\vec{x})$:

$$\Phi(\vec{x}) = (1, x_1^2, x_2^2, x_1 x_2, x_1, x_2)^T \tag{4.17}$$

Now, the decision function becomes linear in the transformed space: $f(\vec{x}) = \vec{\alpha}^T \Phi(\vec{x})$.

4.4.2 Applying the Transform to the Dual Problem

Because we reformulated the SVM into its **Dual Representation**, we don't need to explicitly calculate the weight vector $\vec{\alpha}$ in the high-dimensional space. The decision boundary can be rewritten using inner products of the transformed features:

$$f(\vec{x}) = \sum_{i=1}^{m} \lambda_i y_i \langle \Phi(\vec{x}_i), \Phi(\vec{x}) \rangle + \alpha_0 \tag{4.18}$$

The optimization problem changes accordingly:

Dual Optimization with Feature Transform

$$\text{maximize} \quad -\frac{1}{2}\sum_i \sum_j \lambda_i \lambda_j y_i y_j \langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle + \sum_i \lambda_i$$

subject to $\vec{\lambda} \succeq 0, \quad \sum \lambda_i y_i = 0$

Now we have a linear decision boundary in a higher dimensional space!

4.4.3 Kernel Functions

Computing the explicit transformation $\Phi(\vec{x})$ can be computationally very expensive (or even impossible for infinite dimensions). However, since the algorithm **only depends on the inner product**, we can abstract this using a **Kernel Function**:

$$k(\vec{x}, \vec{x}') = \langle \Phi(\vec{x}), \Phi(\vec{x}') \rangle \tag{4.19}$$

Using kernel functions, we avoid modeling the transformation $\Phi(\vec{x})$ explicitly but still get the same result.

Typical Kernel Functions:

- **Linear:** $k(\vec{x}, \vec{x}') = \langle \vec{x}, \vec{x}' \rangle$
- **Polynomial:** $k(\vec{x}, \vec{x}') = (\langle \vec{x}, \vec{x}' \rangle + 1)^d$ (The example above corresponds to $d = 2$)
-
- **Radial Basis Function (RBF) / Gaussian:**

$$k(\vec{x}, \vec{x}') = \exp\left(-\frac{\|\vec{x} - \vec{x}'\|^2}{2\sigma^2}\right) \tag{4.20}$$

- **Sigmoid:** $k(\vec{x}, \vec{x}') = \tanh(a\vec{x}^T\vec{x}' + b)$

5 Kernels & Kernel Methods

5.1 Feature Transforms

Linear decision boundaries have limitations (cannot classify non-linearly separable data).
Solution: Map data into a higher-dimensional feature space using a non-linear transformation $\phi : \mathbb{R}^d \to \mathbb{R}^D$ $(D \geq d)$.

Example: Mapping 2D data to 2D (squaring):

$$\phi(\vec{x}) = (x_1^2, x_2^2)^T \tag{5.1}$$

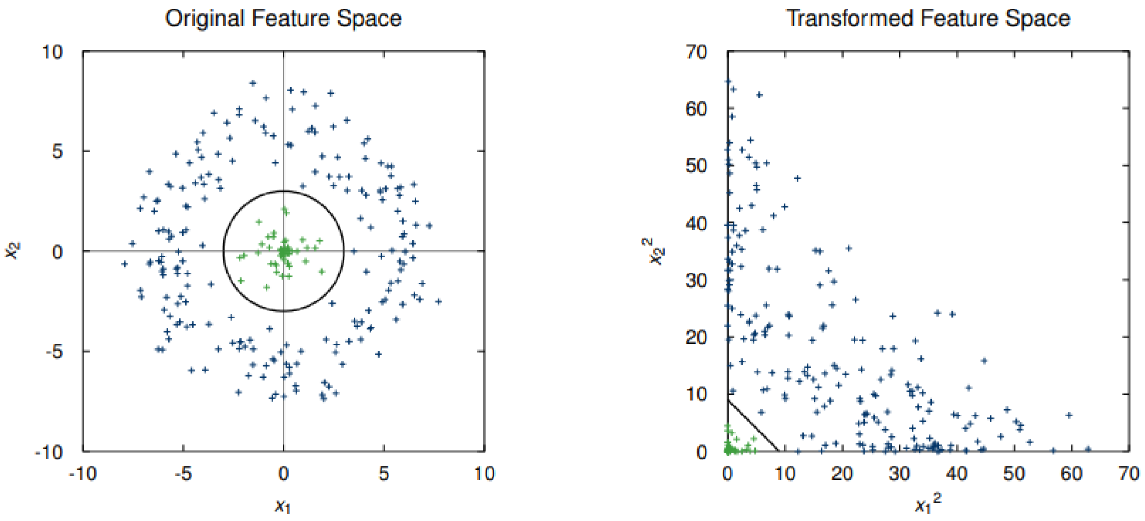

Figure 17: Feature transform: squaring each coordinate

Data that forms a circle in the original space becomes linearly separable in the transformed space.

Property: Distances in the transformed space can be computed using only inner products:

$$\|\phi(\vec{x}) - \phi(\vec{x}')\|_2^2 = \langle(\phi(\vec{x}) - \phi(\vec{x}')), (\phi(\vec{x}) - \phi(\vec{x}'))\rangle \tag{5.2}$$

$$= \langle\phi(\vec{x}), \phi(\vec{x})\rangle - 2\langle\phi(\vec{x}), \phi(\vec{x}')\rangle + \langle\phi(\vec{x}'), \phi(\vec{x}')\rangle \tag{5.3}$$

This observation leads us to Kernel Functions.

5.2 Kernel Functions & The Kernel Trick

Definition: Kernel Function

A kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a symmetric function that maps a pair of features to a real number. It corresponds to an inner product in some feature space:

$$k(\vec{x}, \vec{x}') = \langle\phi(\vec{x}), \phi(\vec{x}')\rangle \tag{5.4}$$

The Kernel Trick: In any algorithm formulated in terms of inner products (like SVM or PCA), we can replace the inner product with a kernel $k(\vec{x}, \vec{x}')$. This avoids the explicit (and expensive) computation of $\phi(\vec{x})$.

Mercer's Theorem: For any positve semidefinite kernel function k, there exists a feature mapping ϕ such that $k(\vec{x}, \vec{x}') = \langle \phi(\vec{x}), \phi(\vec{x}') \rangle$ (kernel function is fully defined by inner products).

Typical Kernel Functions

- **Linear:** $k(\vec{x}, \vec{x}') = \langle \vec{x}, \vec{x}' \rangle$
- **Polynomial:** $k(\vec{x}, \vec{x}') = (\langle \vec{x}, \vec{x}' \rangle + 1)^d$
- **RBF (Radial Basis Function):** $k(\vec{x}, \vec{x}') = \exp\left(-\frac{\|\vec{x}-\vec{x}'\|^2}{\sigma^2}\right)$
- **Sigmoid:** $k(\vec{x}, \vec{x}') = \tanh(\alpha \langle \vec{x}, \vec{x}' \rangle + \beta)$

Kernel Matrix

Definition: Kernel Matrix

For a given set of feature vectors $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m$, we define the **Kernel Matrix** $K \in \mathbb{R}^{m \times m}$ as:

$$K_{i,j} = k(\vec{x}_i, \vec{x}_j) = \langle \phi(\vec{x}_i), \phi(\vec{x}_j) \rangle \tag{5.5}$$

Important Properties:

- The entries of the matrix can be interpreted as **similarity measures** between the feature pairs (because the inner product (german: Skalarprodukt) measures similarity).
- The kernel matrix is always **positive semidefinite** ($K \succeq 0$). This is a necessary condition for a function to be a valid kernel (Mercer's Theorem).

5.3 Kernel SVM

In chapter 4, we saw, that standard SVMs can only learn linear decision boundaries. By applying the kernel trick, we can extend SVMs to learn non-linear decision boundaries. Since the decision rule 4.15 and the dual optimization 4.12 problem depend only on inner products, we can replace them with kernel functions. The decision rule becomes:

$$f(\vec{u}) = \sum_{i=1}^{m} \lambda_i y_i k(\vec{x}_i, \vec{u}) + \alpha_0 \tag{5.6}$$

The dual optimization problem becomes:

$$\max_{\vec{\lambda}} \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j k(\vec{x}_i, \vec{x}_j) \tag{5.7}$$

The choice of the kernel function k determines the type of non-linear decision boundary that can be learned. **Result:** By choosing different kernel functions, we can learn a variety of non-linear decision boundaries without explicitly computing the feature transformation ϕ.

5.4 Kernel PCA

Standard PCA finds principal components by diagonalizing the covariance matrix Σ. In Kernel PCA, we perform PCA in the feature space defined by ϕ. This allows us to capture non-linear structures in the data. Also the computatuional

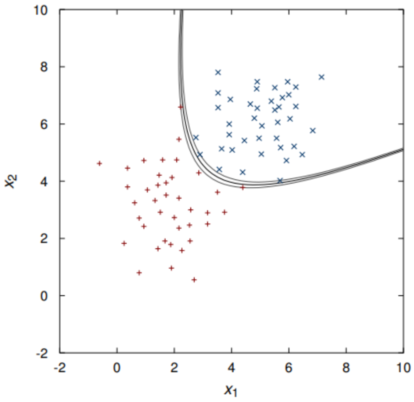

Figure 18: Polynomial Kernel Decision Boundary

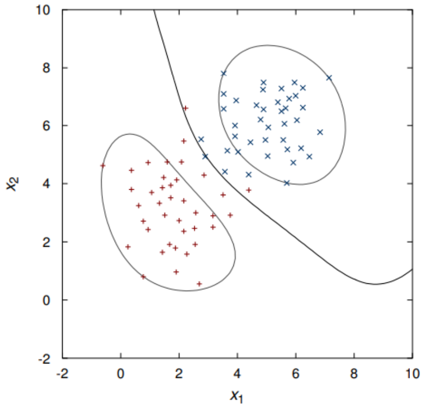

Figure 19: RBF (Gaussian) Kernel Decision Boundary

5.4.1 PCA Revisited

The covariance matrix in feature space is (assuming zero mean):

$$\Sigma = \frac{1}{m} \sum_{i=1}^{m} \vec{x}_i \vec{x}_i^T \tag{5.8}$$

We solve the eigenvalue problem: $\Sigma \vec{e}_i = \lambda_i \vec{e}_i$.

5.4.2 Rewriting the Eigenvalue Problem

The eigenvectors $\vec{e}_i$ lie in the span of the feature vectors and can therefore be expressed as a linear combination of feature vectors:

$$\vec{e}_i = \sum_{k=1}^{m} \alpha_{i,k} \vec{x}_k \tag{5.9}$$

Substituting this into the eigenvalue equation:

$$\left(\frac{1}{m} \sum_{j=1}^{m} \vec{x}_j \vec{x}_j^T \right) \cdot \sum_k \alpha_{i,k} \vec{x}_k = \lambda_i \sum_k \alpha_{i,k} \vec{x}_k \tag{5.10}$$

To introduce the kernel, we multiply from the left by $\vec{x}_l^T$ for all l:

$$\sum_{j,k} \alpha_{i,k} \underbrace{\vec{x}_l^T \vec{x}_j}_{k(\vec{x}_l,\vec{x}_j)} \underbrace{\vec{x}_j^T \vec{x}_k}_{k(\vec{x}_j,\vec{x}_k)} = m \lambda_i \sum_k \alpha_{i,k} \underbrace{\vec{x}_l^T \vec{x}_k}_{k(\vec{x}_l,\vec{x}_k)} \tag{5.11}$$

Since all feature vectors only appear in the form of inner products, we can replace them with the kernel function k:

Kernel PCA Equation

$$\sum_{j,k} \alpha_{i,k} k(\vec{x}_l, \vec{x}_j) k(\vec{x}_j, \vec{x}_k) = m\lambda_i \sum_k \alpha_{i,k} k(\vec{x}_l, \vec{x}_k) \tag{5.12}$$

In matrix notation using the Kernel Matrix K:

$$K^2 \vec{\alpha}_i = m\lambda_i K \vec{\alpha}_i \tag{5.13}$$

This implies:

$$K \vec{\alpha}_i = m\lambda_i \vec{\alpha}_i \tag{5.14}$$

So now we only need to solve the eigenvalue problem for the kernel matrix K!

5.4.3 Projection and Centering

- **Computation:** We solve the eigenvalue problem for the $(m \times m)$ kernel matrix K.
- **Projection:** The projection c of a transformed feature vector $\phi(\vec{x})$ onto the principal component $\vec{e}_i$ is easily computed by:

$$c = \phi(\vec{x})^T \vec{e}_i = \sum_k \alpha_{i,k} k(\vec{x}, \vec{x}_k) \tag{5.15}$$

- **Centering:** It is assumed that transformed features have zero mean. This is enforced by centering the kernel matrix:

$$\tilde{K}_{i,j} = \tilde{\phi}(\vec{x}_i)^T \tilde{\phi}(\vec{x}_j) = K_{i,j} - \frac{1}{m} \sum_k K_{i,k} - \frac{1}{m} \sum_k K_{k,j} - \frac{1}{m} \sum_k K_{k,l}$$

5.5 Kernels for Sequences

Kernels allow us to apply SVMs/PCA to non-vectorial data like text or speech.

- **String Kernels / DTW:** Based on Dynamic Time Warping distance D.

$$k(\vec{x}, \vec{y}) = \exp(-D(\vec{x}, \vec{y})) \tag{5.16}$$

- **Fisher Kernels:** Combine generative models (e.g., HMMs) with discriminative classifiers.

$$k(\vec{x}, \vec{x}') = \mathbf{J}_\theta(\vec{x})^T \mathbf{I}^{-1} \mathbf{J}_\theta(\vec{x}') \tag{5.17}$$

  where $\mathbf{J}_\theta$ is the Fisher Score (gradient of log-likelihood) and $\mathbf{I}$ is the Fisher Information Matrix.

5.6 Laplacian SVM

6 Independent Component Analysis (ICA)

While Principal Component Analysis (PCA) and Whitening rely on second-order statistics (covariance) to decorrelate data, Independent Component Analysis (ICA) goes a step further. It aims to recover latent variables that are not only uncorrelated but statistically **independent**.

6.1 The Cocktail-Party Problem

The classic motivating example for ICA is the "Cocktail-Party Problem". Imagine a room with two speakers $(s_1(t), s_2(t))$ and two microphones at different locations. The microphones record time signals $x_1(t)$ and $x_2(t)$. Since sound travels linearly, the recorded signals are weighted sums of the original sources:

$$\begin{aligned} x_1(t) &= a_{11}s_1(t) + a_{12}s_2(t) \\ x_2(t) &= a_{21}s_1(t) + a_{22}s_2(t) \end{aligned} \tag{6.1}$$

Here, the coefficients a_{ij} depend on the distances between microphones and speakers. The goal of ICA is to recover the original sources $s(t)$ using only the observations $x(t)$, without knowing the mixing parameters a_{ij} or the source distributions.

6.2 Latent Variable Model

We formalize this using a statistical latent variables model. We observe a random vector $\vec{x} \in \mathbb{R}^n$ which is a linear mixture of n independent latent components $\vec{s} \in \mathbb{R}^n$:

$$\vec{x} = A\vec{s} \tag{6.2}$$

where $A \in \mathbb{R}^{n \times n}$ is the constant, unknown **mixing matrix**. The optimization task is to estimate both the mixing matrix A and the realizations of the latent variables $\vec{s}$.

Assumptions of ICA

To ensure the model is solvable, we make the following assumptions:

1. The components s_i are statistically independent.
2. The components s_i must have **non-Gaussian** distributions (except for at most one component).
3. The mixing matrix A is square and invertible (for the basic ICA model).

6.2.1 Ambiguities

Since both A and $\vec{s}$ are unknown, the model $\vec{x} = A\vec{s}$ has two inherent ambiguities that cannot be resolved without prior knowledge:

1. **Scaling and Sign:** We cannot determine the variance of the independent components. A scalar factor α can be canceled out by dividing the corresponding column of A by α. Therefore, we usually restrict $\vec{s}$ to have unit variance ($E\{\vec{s}\vec{s}^T\} = I$). The sign is also undetermined.

2. **Ordering:** There is no natural order of the components. A permutation matrix P can swap components in $\vec{s}$ while permuting columns in A accordingly.

6.3 Why Decorrelation is Not Enough

In previous chapters, we used the Whitening Transform (based on PCA) to decorrelate data. For zero-mean data, the whitening transform is given by $\tilde{\vec{x}} = D^{-1/2}U^T\vec{x}$, where $E\{\vec{x}\vec{x}^T\} = UDU^T$.

While whitening ensures that the variables are uncorrelated ($E\{\tilde{\vec{x}}\tilde{\vec{x}}^T\} = I$), it does not guarantee independence.

The Rotation Problem: Whitening is only defined up to an orthogonal rotation. If $\tilde{\vec{x}}$ is white, then for any orthogonal matrix R, the vector $\hat{\vec{x}} = R\tilde{\vec{x}}$ is also white:

$$E\{\hat{\vec{x}}\hat{\vec{x}}^T\} = RE\{\tilde{\vec{x}}\tilde{\vec{x}}^T\}R^T = RIR^T = I \tag{6.3}$$

Since the Gaussian distribution is rotationally symmetric (determined fully by second-order moments), PCA cannot distinguish between the original axes and rotated axes for Gaussian data. This explains why ICA requires **non-Gaussian** data to identify the unique mixing matrix.

6.4 The Principle of Non-Gaussianity

To solve ICA, we use the Central Limit Theorem (CLT). The CLT states that the sum of independent random variables tends toward a Gaussian distribution.

Consider a linear combination of the observed mixture variables $y = \vec{w}^T\vec{x}$. Substituting $\vec{x} = A\vec{s}$ and letting $\vec{z} = A^T\vec{w}$:

$$y = \vec{w}^T A\vec{s} = \vec{z}^T\vec{s} = \sum_i z_i s_i \tag{6.4}$$

This y is a sum of independent components. According to the CLT, this sum is usually "more Gaussian" than the individual source components s_i. Conversely, y is **least Gaussian** when it corresponds to exactly one of the independent components (i.e., when only one z_i is non-zero).

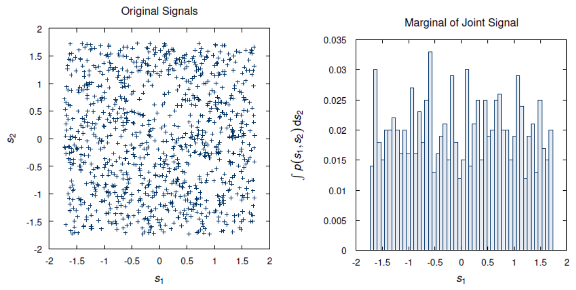

Figure 20: Original Signals: The scatter plot shows a square (independence). The histogram shows a uniform distribution, which is clearly non-Gaussian.

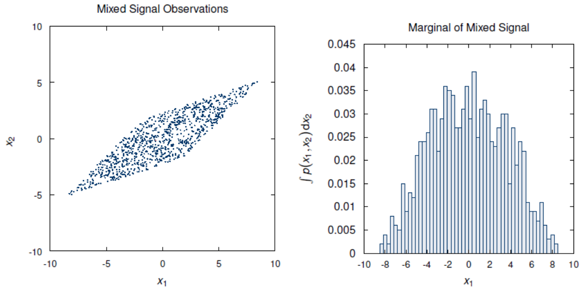

Figure 21: Mixed Signals: The scatter plot becomes a parallelogram (correlation). Due to the CLT, the histogram approaches a Gaussian bell curve.

Conclusion: To find the independent components, we must find the weight vector $\vec{w}$ that **maximizes the non-Gaussianity** of $\vec{w}^T\vec{x}$.

6.5 Measures of Non-Gaussianity

To perform the maximization described above, we need quantitative measures of how non-Gaussian a distribution is. We assume the variable y is centered and has unit variance.

6.5.1 Differential Entropy

The fundamental concept behind many measures of non-Gaussianity is **Entropy**. In information theory, entropy measures the uncertainty or randomness of a random variable. For a continuous random variable Y with probability density function $p(y)$, the **Differential Entropy** $H(Y)$ is defined as:

$$H(Y) = -\int p(y) \log p(y)\, dy \tag{6.5}$$

A crucial theorem for ICA states that for a fixed variance, the **Gaussian distribution maximizes the entropy**. This means the Gaussian distribution is the "most random" or least structured distribution. Distributions that are "spiky" or "flat" (super- or sub-Gaussian) have lower entropy.
Interpretation: Entropy can also be interpreted as a measure of code length. $H(Y)$ relates to the average code length required to encode the variable.

6.5.2 Kurtosis

Kurtosis is the fourth-order cumulant. For a random variable y with unit variance:

$$\operatorname{kurt}(y) = E\{y^4\} - 3(E\{y^2\})^2 = E\{y^4\} - 3 \tag{6.6}$$

- For a Gaussian distribution, $\operatorname{kurt}(y) = 0$.
- **Super-Gaussian** (spiky, e.g., Laplace): $\operatorname{kurt}(y) > 0$.
- **Sub-Gaussian** (flat, e.g., Uniform): $\operatorname{kurt}(y) < 0$.

We optimize the absolute kurtosis $|\operatorname{kurt}(\vec{w}^T\vec{x})|$. While theoretically sound, kurtosis is sensitive to outliers because of the fourth power (y^4).

6.5.3 Negentropy

Entropy $H(y)$ measures randomness. A Gaussian variable has the largest entropy among all distributions with the same variance. Negentropy $J(y)$ is defined to measure the distance to normality:

$$J(y) = H(y_{\text{Gauss}}) - H(y) \tag{6.7}$$

where y_{Gauss} is a Gaussian variable with the same covariance as y. $J(y)$ is always non-negative and zero only if y is Gaussian. Unlike Kurtosis, Negentropy is statistically robust, though computing the entropy requires estimating the pdf, which is computationally difficult. In practice, approximations involving non-quadratic functions G are used:

$$J(y) \approx [E\{G(y)\} - E\{G(y_{\text{Gauss}})\}]^2 \tag{6.8}$$

6.5.4 Mutual Information

Another approach is minimizing the Mutual Information (MI) between the estimated components. MI measures the statistical dependence between variables.

$$\text{MI}(\vec{y}) = \sum_{i=1}^{n} H(y_i) - H(\vec{y}) \tag{6.9}$$

For uncorrelated variables, minimizing Mutual Information is equivalent to maximizing the sum of Negentropies of the components.

6.6 ICA Estimation Algorithm

A standard algorithm (conceptually similar to FastICA) to estimate one independent component is:

Algorithm 2 Basic ICA Estimation Algorithm

```
Apply centering transform
Apply whitening transform
$i \leftarrow 1$
repeat
    Take a random vector $\vec{w}_i$
    Maximize non-Gaussianity of $\vec{w}_i^T \vec{x}$ subject to:
        $\|\vec{w}_i\| = 1$
        $\vec{w}_j^T \vec{w}_i = 0, \quad \forall j < i$
    $i \leftarrow i + 1$
until $i > n$                                  ▷ $n$: number of independent components
Use weight matrix: $W = (\vec{w}_1^T, \vec{w}_2^T, \dots, \vec{w}_n^T)$ to compute $\vec{s}$
Output: independent components $\vec{s}$
```

To do this, you have to choose a measure of non-Gaussianity (as described in 6.5).