

CS 3780 Notes

Jinzhou Wu

October 7, 2025

Contents

1	Supervised Learning and KNN	2
2	Inductive Learning and Decision Trees	5
3	Prediction and Overfitting	8
4	Model Selection and Assessment	10
5	Linear Classifiers	14
6	Perceptron	17
7	Support Vector Machines	18
8	Kernels and Duality	20
9	Regularized Linear Models	25
10	Optimization with Gradient Descent	29
11	Neural Networks	30

1 Supervised Learning and KNN

Date: Aug 28, 2025

1.1 Supervised Learning

In supervised learning, we are given a dataset:

$$S = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

where $x_i \in \mathcal{X}$ is a feature vector and $y_i \in \mathcal{Y}$ is its label. The goal is to learn a hypothesis function:

$$h : \mathcal{X} \rightarrow \mathcal{Y}$$

that approximates the unknown target function $f : \mathcal{X} \rightarrow \mathcal{Y}$.

Key Concepts

- **Instance:** A single feature vector $\mathbf{x} \in \mathcal{X}$.
- **Instance Space \mathcal{X} :** The set of all possible feature vectors.
- **Label y :** The output to be predicted.
- **Label Space \mathcal{Y} :** The set of all possible labels.

Types of Supervised Learning

- **Binary Classification:** $\mathcal{Y} = \{-1, +1\}$
- **Multi-class Classification:** $\mathcal{Y} = \{1, 2, \dots, k\}$
- **Regression:** $\mathcal{Y} \subseteq \mathbb{R}$
- **Structured Output:** $\mathcal{Y} = \text{Object}$ (e.g., protein structures)

1.2 K-Nearest Neighbors (KNN)

KNN is a non-parametric learning algorithm that predicts the label of a new instance \mathbf{x}' using the labels of the k closest points in the training dataset according to a similarity (or distance) measure.

Algorithm

KNN Algorithm:

1. **Input:** Training set $S = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$, similarity function K , number of neighbors k .
2. For a test point \mathbf{x}' , compute $K(\mathbf{x}_i, \mathbf{x}')$ for all i .
3. Find the k nearest neighbors:

$$\text{knn}(\mathbf{x}') = \{i \mid \mathbf{x}_i \text{ among } k \text{ closest to } \mathbf{x}'\}.$$

4. Predict the label:

$$\hat{y} = \arg \max_{y \in \mathcal{Y}} \sum_{i \in \text{knn}(\mathbf{x}')} \mathbf{1}(y_i = y).$$

1.3 Weighted KNN

Weighted KNN assigns higher weights to closer neighbors using the similarity function K .

$$\hat{y} = \arg \max_{y \in \mathcal{Y}} \sum_{i \in \text{knn}(\mathbf{x}')} K(\mathbf{x}_i, \mathbf{x}') \cdot \mathbf{1}(y_i = y)$$

Weighted KNN for Regression

For regression problems, the prediction is a weighted average:

$$h(\mathbf{x}') = \frac{\sum_{i \in \text{knn}(\mathbf{x}')} y_i \cdot K(\mathbf{x}_i, \mathbf{x}')}{\sum_{i \in \text{knn}(\mathbf{x}')} K(\mathbf{x}_i, \mathbf{x}')}$$

1.4 Similarity Measures

Different similarity or distance measures can be used depending on the problem:

- **Gaussian Kernel:**

$$K(\mathbf{x}, \mathbf{x}') = \exp \left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2} \right)$$

- **Laplace Kernel:**

$$K(\mathbf{x}, \mathbf{x}') = \exp (-\|\mathbf{x} - \mathbf{x}'\|)$$

- **Cosine Similarity:**

$$K(\mathbf{x}, \mathbf{x}') = \cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{x}'}{\|\mathbf{x}\| \|\mathbf{x}'\|}$$

1.5 Types of Attributes

- **Categorical:** e.g., EyeColor \in {brown, blue, green}
- **Boolean:** e.g., Alive \in {True, False}
- **Numeric:** e.g., Age, Height
- **Structured:** e.g., sentences, protein sequences

1.6 Properties of KNN

- Simple, intuitive, and non-parametric.
- Requires a meaningful similarity measure.
- Memory-intensive: stores the entire training dataset.
- Computationally expensive for large datasets.
- Suffers from the **curse of dimensionality**.
- KNN is more like a *memorization* method rather than true generalization.

2 Inductive Learning and Decision Trees

Date: Sep 2, 2025

2.1 Inductive Learning

Inductive learning is the process of learning a general rule or hypothesis from specific observed examples. Given a training dataset:

$$S = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

we aim to find a hypothesis h such that $h(x) \approx y$ for unseen examples.

Key Ideas

- Training data provides labeled examples of inputs and outputs.
- The goal is to infer a hypothesis h consistent with as many training examples as possible.
- If multiple hypotheses are consistent, we aim to choose one that generalizes well.

2.2 Version Space

Definition (Version Space). The **version space** is the set of all hypotheses in the hypothesis space H that are consistent with the observed training examples:

$$VS = \{h \in H \mid \forall (x_i, y_i) \in S, h(x_i) = y_i\}.$$

Using Version Space for Learning

- Start with the set of all hypotheses.
- Remove any hypothesis inconsistent with any training example.
- The remaining hypotheses form the version space.

2.3 List-Then-Eliminate Algorithm

Algorithm

Algorithm: *List-Then-Eliminate*

1. Initialize $VS \leftarrow H$ (all hypotheses).
2. For each training example (x_i, y_i) :
 - Remove all $h \in VS$ such that $h(x_i) \neq y_i$.
3. Return the remaining hypotheses in VS .

Takeaway: Tracking the entire version space can be expensive in both time and memory. Instead, we often directly construct a consistent hypothesis, e.g., using decision trees.

2.4 Decision Trees

Definition (Decision Tree). A decision tree represents a function $h : \mathcal{X} \rightarrow \mathcal{Y}$ as a tree structure:

- **Internal nodes:** Test a single feature (e.g., "Color").
- **Branches:** Possible values of that feature (e.g., "Red" or "Green").
- **Leaf nodes:** Assign a label based on the path from root to leaf.

Using a Decision Tree

To classify a new example:

1. Start at the root.
2. At each internal node, test the corresponding feature.
3. Follow the branch matching the feature value.
4. Stop at a leaf and return its label.

2.5 Top-Down Induction of Decision Trees (IDT)

Algorithm

Algorithm: $IDT(S, Features)$

1. If all examples in S have the same label, return a leaf node with that label.
2. If no features remain, return a leaf node with the majority label in S .
3. Otherwise:
 - Choose the best feature A to split on.
 - Partition S into subsets $\{S_v\}$ by the values of A .
 - For each value v of A :
 - Recursively call $IDT(S_v, Features \setminus \{A\})$.

2.6 Choosing the Best Split

Error-Based Split Criterion: To choose the best attribute A to split on:

$$\text{Err}(S) = \min(\#positive, \#negative)$$

$$\text{Err}(S \mid A) = \sum_v \text{Err}(S_v)$$

Select A that maximizes:

$$\Delta\text{Err} = \text{Err}(S) - \text{Err}(S \mid A).$$

2.7 Properties of Decision Trees

- Easy to interpret and visualize.
- Can represent complex decision boundaries.
- Handles both categorical and numerical features.
- Prone to overfitting if the tree grows too deep.
- Typically combined with pruning techniques for better generalization.
- Basis for more advanced models like Random Forests and Gradient Boosted Trees.

3 Prediction and Overfitting

Date: Sep 4, 2025

Learning as Prediction

Goal: Given a training dataset $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$ drawn **i.i.d.** from an unknown distribution \mathcal{D} , learn a hypothesis h such that $h(x) \approx y$ for unseen data.

World as a Distribution

Features (e.g., Farm, Color, Size, Firmness) and labels (e.g., Tasty) are random variables. The underlying distribution \mathcal{D} defines:

- **Joint Distribution:** $P(X, Y)$
- **Marginal Distribution:** $P(X)$
- **Conditional Distribution:** $P(Y|X)$

Sample vs. Prediction Error

Definition (Sample (Empirical) Error).

$$\hat{L}_S(h) = \frac{1}{|S|} \sum_{(x_i, y_i) \in S} \mathbf{1}[h(x_i) \neq y_i]$$

Definition (Prediction (True) Error).

$$L_{\mathcal{D}}(h) = \mathbb{P}_{(x, y) \sim \mathcal{D}}[h(x) \neq y]$$

Goal: Minimize *true prediction error*, not just training error.

Overfitting

Overfitting occurs when a hypothesis h achieves **low training error** but **high test error** because it learns noise and idiosyncrasies of the training set instead of general patterns.

Example: Take an i.i.d. training set $S = \{(x_1, f(x_1)), \dots\}$ and return h_s such that

$$h_s(x) = \begin{cases} f(x_i) & \text{if } x = x_i \text{ for some } i \\ \text{flip a coin} & \text{otherwise} \end{cases}$$

- h_s has zero training error but predicts randomly on unseen data.

Key Characteristics

- Fits the training set too closely, including random noise.
- Poor generalization to unseen data.
- Common when the hypothesis space is very flexible (e.g., deep trees, high-degree polynomials).

Overfitting in Decision Trees

- Fully grown decision trees can perfectly memorize the training set.
- This leads to zero empirical error but poor generalization.
- Needs mechanisms like early stopping or pruning to avoid overfitting.

Mitigating Overfitting in Decision Trees

Strategies

- **Limit Model Complexity:** Restrict tree depth or number of nodes.
- **Early Stopping:** Stop splitting when:
 - Error reduction after splitting is small.
 - Too few examples remain in a node.
- **Pruning:** Grow the full tree, then prune back:
 - Replace a subtree with a leaf if it does not significantly increase prediction error.
 - E.g., reduced-error pruning.

Inductive Bias

Definition (Inductive Bias). An **inductive bias** is a set of assumptions a learning algorithm uses to predict unseen data. Without bias, learning from finite samples would be impossible.

Inductive Bias in Decision Trees

- Standard IDT assumes:
 - The simplest consistent tree is preferred.
 - Features are chosen based on information gain or error reduction.
- If tree depth is restricted, bias increases but variance decreases.

Bias-Variance Tradeoff

Definition (Prediction Error Decomposition).

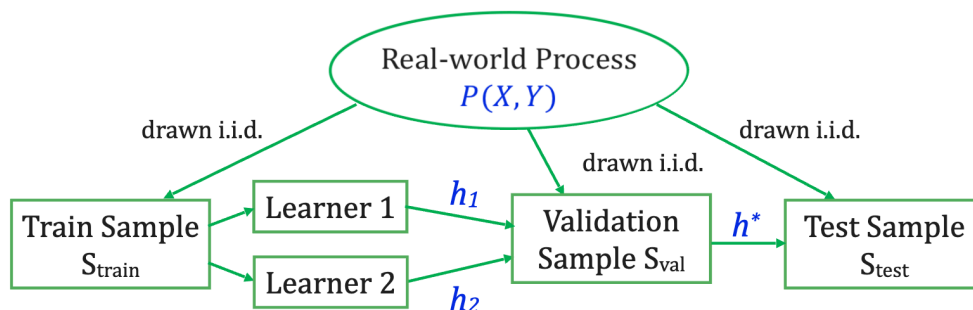
$$\text{Expected Error} = \underbrace{\text{Bias}^2}_{\text{error from assumptions}} + \underbrace{\text{Variance}}_{\text{error from data fluctuations}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

- **Bias** measures error from erroneous assumptions in the learning algorithm. Simple models (left side) have high bias and underfit the data.
- **Variance** measures error from sensitivity to small fluctuations in the training set. Complex models (right side) have high variance and overfit.
- **Total error** is minimized at an intermediate model complexity, balancing bias and variance.

4 Model Selection and Assessment

Date: Sep 9, 2025

4.1 Validation Sample



- **Training:** Run learning algorithm l times (e.g. different parameters).
- **Validation Error:** Errors $err_{S_{val}}(h_i)$ are estimates of $err_p(h_i)$ for each h_i .
- **Selection:** Use h^* with $\min err_{S_{val}}(\hat{h}_i)$ for prediction on test examples.

Two Nested Learning Algorithms

- **Primary Learning Algorithm on S_{train}**
 - For each variant $A_1 \dots A_l$ of learning algorithm, $h_i = A_i(S_{train})$
 - Example: Decision Tree (DT) that stops at i nodes.
- **Secondary Learning Algorithm on S_{val}**
 - Hypothesis space: $H' = \{h_1, \dots, h_l\}$
 - Learning Algorithm: $h^* = \arg \min_{h \in H'} [err_{S_{val}}(h)]$

Typical ML Experiment

- Collect data $S = \{(\vec{x}_1, y_1), \dots, (\vec{x}_m, y_m)\}$
- Split randomly into S_{train} , S_{val} , S_{test}
- REPEAT
 - Train on S_{train}
 - Validate on S_{val}
- UNTIL we think we have a good rule h .
- Test h on S_{test} to evaluate its accuracy/error.

4.2 Cross-Validation

k-fold Cross-Validation:

- **Given:**
 - Training examples S
 - Learning algorithm A_p with parameter p (model architectures or hyperparameters)
- **Compute:**
 - Randomly partition S into k equally sized subsets S_1, \dots, S_k
 - For each value of p :
 - * For i from 1 to k :
 - Train A_p on $S \setminus S_i$ and get h_i
 - Apply h_i to S_i and compute $err_{S_i}(h_i)$
 - * Compute cross-validation error:

$$err_{CV}(A_p) = \frac{1}{k} \sum_i err_{S_i}(h_i)$$

- **Selection:**
 - Pick parameter p^* that minimizes $err_{CV}(A_p)$
 - Train $A_{p^*}(S)$ on full sample S to get final h

4.3 Generalization Error of Hypothesis

- **Given**
 - Samples S_{train} and S_{test} of labeled instances
 - Learning Algorithm A
- **Setup**
 - Train learning algorithm A on S_{train} , result is h
 - Apply h to S_{test} and compare predictions against true labels
- **Test**
 - Error on test sample $err_{S_{\text{test}}}(h)$ is estimate of true error $err_P(h)$
 - Compute confidence interval

4.4 Significance Testing with the Binomial Distribution

- **Goal:** Assess whether the observed error rate of a hypothesis h on a test set provides statistically significant evidence that the true error rate $err_P(h)$ is below (or above) a certain threshold.
- **Null Hypothesis:** Assume $err_P(h) \geq \epsilon$ for some threshold ϵ .

- **Test Statistic:** Let m be the number of test examples, and k the number of observed errors. Under the null hypothesis, the number of errors X follows a Binomial distribution: $X \sim \text{Binomial}(m, \epsilon)$.
- **p-value:** Compute the probability of observing k or fewer errors under the null hypothesis:

$$p\text{-value} = P(X \leq k \mid p = \epsilon, m)$$

- **Decision:** If the p -value is less than the significance level (e.g., 0.05 for 95% confidence), reject the null hypothesis and conclude that there is significant evidence that $\text{err}_P(h) < \epsilon$.
- **Interpretation:** This test quantifies how likely it is to observe the empirical error rate (or lower) if the true error rate were at least ϵ .

4.5 Normal Confidence Intervals

- **Rule of thumb:** When $mp(1-p) \geq 5$, the binomial distribution can be approximated by a normal distribution $N(\mu, \sigma^2)$ with $\mu = mp$ and $\sigma^2 = mp(1-p)$.
- **Normal confidence interval (95% confidence):**

$$\text{err}_P(h) \in \left[\text{err}_S(h) - 1.96\sqrt{\frac{p(1-p)}{m}}, \text{err}_S(h) + 1.96\sqrt{\frac{p(1-p)}{m}} \right]$$

- With approximation $p \approx \text{err}_S(h) \rightarrow$ Called “Standard Error” confidence intervals.

4.6 Hoeffding Bound for Generalization Error

- **Hoeffding Bound:** For any loss function that takes values in $[0, 1]$ and any hypothesis h , with probability at least $1 - \delta$ (where $1 - \delta$ is called the **confidence level**), over the random choice of test samples S of size m , the following holds:

$$\text{err}_P(h) \in \left[\text{err}_S(h) - \sqrt{\frac{-0.5 \ln(\delta/2)}{m}}, \text{err}_S(h) + \sqrt{\frac{-0.5 \ln(\delta/2)}{m}} \right]$$

- This means that, with probability at least $1 - \delta$, the true error $\text{err}_P(h)$ lies within the interval centered at the empirical error $\text{err}_S(h)$, with a margin that decreases as the number of test samples m increases.
- The parameter $1 - \delta$ is the probability that the interval contains the true error. For example, if $\delta = 0.05$, then $1 - \delta = 0.95$ corresponds to a 95% confidence interval.

4.7 McNemar’s Test

- **Given:**
 - Two hypotheses h_1 and h_2
 - Test set S_{test} with m examples
- **Null Hypothesis:** $\text{err}_P(h_1) = \text{err}_P(h_2)$

- Implies that $err_S(h_1)$ and $err_S(h_2)$ come from binomial distributions with the same p
- Implies that the number of wins w (where h_1 is correct and h_2 is not) and losses l (where h_2 is correct and h_1 is not) follow:

$$W \sim \text{Binomial}(p = 0.5, n = w + l)$$

- **Test:**

- Count wins w and losses l on S_{test}
- Reject the null hypothesis with 95% confidence if:

$$P(W \leq w \mid p = 0.5, n = w + l) < 0.025$$

or

$$P(W \geq w \mid p = 0.5, n = w + l) < 0.025$$

- **Why 0.025 instead of 0.05?**

The total significance level is 0.05 for a 95% confidence test, but since McNemar's test is two-sided (we care if either h_1 is significantly better than h_2 or vice versa), we split the significance level equally between both tails of the distribution: 0.025 for each tail.

5 Linear Classifiers

Date: Sep 11, 2025

5.1 Vectors and Hyperplanes

Euclidean Embeddings Data are represented as d dimensional vectors in \mathbb{R}^d . The choice of representation is part of "inductive bias".

Euclidean Norm The Euclidean norm of a vector $x \in \mathbb{R}^d$ is defined as:

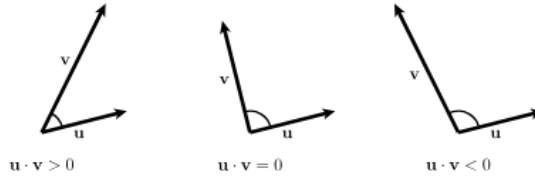
$$||\vec{x}|| = \sqrt{\sum_{i=1}^d x_i^2}$$

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

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

Alternative notation: $\vec{x} \cdot \vec{y}$ or $\vec{x}^T \vec{y}$.

Angle & Projection $\vec{w} \cdot \vec{x} = ||\vec{w}|| ||\vec{x}|| \cos(\theta)$, where θ is the angle between \vec{w} and \vec{x} , and $\frac{\vec{w} \cdot \vec{x}}{||\vec{w}||}$ is the signed length of the projection of \vec{x} onto \vec{w} .



Hyperplanes In d -dimensional space, a **hyperplane** is defined by a vector $\vec{w} \in \mathbb{R}^d$ and a scalar $b \in \mathbb{R}$:

$$\left\{ \vec{x} \in \mathbb{R}^d \mid \vec{w} \cdot \vec{x} + b = 0 \right\}$$

Geometric interpretation:

- The hyperplane is orthogonal to \vec{w} .
- The distance to the origin (along \vec{w}) is $-\frac{b}{||\vec{w}||}$.
- All points on the hyperplane satisfy $\vec{w} \cdot \vec{x} = -b$.

5.2 Linear Classifiers

For a vector $\vec{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, the hypothesis $h_{\vec{w},b} : \mathbb{R}^d \rightarrow \{-1, +1\}$ is called a d dimensional linear classifier and defined as

$$h_{\vec{w},b}(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b) = \begin{cases} +1 & \vec{w} \cdot \vec{x} + b > 0 \\ -1 & \vec{w} \cdot \vec{x} + b \leq 0 \end{cases}$$

Also called linear predictor or halfspace.

Decision Boundaries in Different Dimensions

- **One dimension:** $h_{w,b}(x) = \text{sign}(wx + b)$
 - Decision boundary: a point (1d hyperplane) at $x = -\frac{b}{w}$
- **Two dimensions:** $h_{\vec{w},b}(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b)$
 - Decision boundary: a line (2d hyperplane) defined by $\vec{w} \cdot \vec{x} + b = 0$
- **d dimensions:** $\text{sign}(\vec{w} \cdot \vec{x} + b)$
 - Decision boundary: a hyperplane $\vec{w} \cdot \vec{x} + b = 0$

Homogenous Linear Classifiers A classifier is **homogenous** if $b = 0$ (otherwise non-homogenous).

Fact: Any d dimensional learning problem for linear classifiers has a **homogenous** form in $d + 1$ dimensions.

Non-homogenous:	Homogenous:
$HS^d = \{h_{\vec{w},b} \mid \vec{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$ <ul style="list-style-type: none"> • \vec{x} • \vec{w}, b • $\vec{w} \cdot \vec{x} + b$ 	$HS_{homog}^{d+1} = \{h_{\vec{w},b} \mid \vec{w} \in \mathbb{R}^{d+1}\}$ <ul style="list-style-type: none"> • $\vec{x}' = (\vec{x}, 1)$ • $\vec{w}' = (\vec{w}, b)$ • $\vec{w}' \cdot \vec{x}' = \vec{w} \cdot \vec{x} + b$

Without loss of generality, we will now focus on **homogenous linear classifiers** with $\|\vec{w}_i\| = 1$.

Consistent Linear Classifiers

- Data set of labelled instances $\mathcal{S} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \dots, (\vec{x}_m, y_m)\}$.
- Linear classifier $h_{\vec{w}}$ is **consistent** with \mathcal{S} if for all $(\vec{x}_i, y_i) \in \mathcal{S}$:
 - $\vec{w} \cdot \vec{x}_i > 0$ if $y_i = 1$
 - $\vec{w} \cdot \vec{x}_i \leq 0$ if $y_i = -1$
- Data set \mathcal{S} is **linearly separable** (zero training error) if there is a linear classifier $h_{\vec{w}}$ that is consistent with it.

Margin A data set \mathcal{S} is linearly separable with a **(geometric) margin** γ if:

- There is a linear classifier $h_{\vec{w}}$ that is consistent with \mathcal{S} .
- The distance of any instance in \mathcal{S} to the decision boundary of $h_{\vec{w}}$ is at least γ .

Mathematical definition: There is \vec{w} such that $\|\vec{w}\| = 1$ and for all data points $(\vec{x}_i, y_i) \in \mathcal{S}$:

$$\begin{cases} \vec{w} \cdot \vec{x}_i \geq \gamma & \text{if } y_i = 1 \\ \vec{w} \cdot \vec{x}_i \leq -\gamma & \text{if } y_i = -1 \end{cases} \iff y_i(\vec{w} \cdot \vec{x}_i) \geq \gamma$$

Larger margin \implies Easier to find consistent linear classifier.

5.3 Perceptron Algorithm

Algorithm

The Perceptron Algorithm (homogeneous & batch)

- **Input:** training data $\mathcal{S} = \{(\vec{x}_1, y_1), \dots, (\vec{x}_m, y_m)\}$
- **Initialize** $\vec{w}^{(0)} = (0, \dots, 0)$ and $t = 0$
- **While** there is $i \in [m]$ such that $y_i(\vec{w}^{(t)} \cdot \vec{x}_i) \leq 0$:
 - $\vec{w}^{(t+1)} = \vec{w}^{(t)} + y_i \vec{x}_i$
 - $t \leftarrow t + 1$
- **End While**
- **Output** $\vec{w}^{(t)}$

Good Practice: Initialize $\vec{w}^{(0)}$ randomly. Shuffle \mathcal{S} and check data one-by-one for update condition. Iterate until no updates are needed or maximum iterations are reached.

6 Perceptron

Date: Sep 16, 2025

- **Theorem:** Given a dataset $S = \{(\vec{x}_1, y_1), \dots, (\vec{x}_m, y_m)\}$ and a radius R such that $\|\vec{x}_i\| \leq R$ for all $i \in [m]$. If S is linearly separable with (geometric) **margin** γ , then Perceptron makes at most R^2/γ^2 updates before finding a consistent linear classifier.
- This upper bound holds even if
 - We scale every instance in the training set
 - We shuffle the training set
 - We don't know the value of γ
- Actual number of updates can vary, but we know it cannot be more than R^2/γ^2

7 Support Vector Machines

Date: Sep 18, 2025

- **Assumption:** Training examples are linearly separable.
- **Optimal Hyperplane:** The separating hyperplane that maximizes the distance to the closest training examples.

7.1 Margin of a Linear Classifier

Definition: For a linear classifier $h_{w,b}$, the (functional) margin γ_i of an example (x_i, y_i) with $x \in \mathbb{R}^N$ and $y \in \{-1, +1\}$ is

$$\gamma_i = y_i (\vec{w} \cdot \vec{x}_i + b)$$

Definition: The margin is called **geometric margin**, if $\|\vec{w}\| = 1$. For general \vec{w} , the term functional margin is used to indicate that the norm of \vec{w} is not necessarily 1.

Definition: The (hard) margin of a linear classifier $h_{w,b}$ on sample S is

$$\gamma = \min_{(x_i, y_i) \in S} y_i (\vec{w} \cdot \vec{x}_i + b)$$

7.2 Computing Optimal Hyperplanes

- **Requirement:** Zero training error.

$$\forall (\vec{x}_i, y_i) \in S : y_i (\vec{w} \cdot \vec{x}_i + b) > 0$$

- **Additional Requirement:** Maximize the distance to the closest training examples.

$$\max_{\vec{w}, b, \gamma} \gamma \quad \text{s.t.} \quad \gamma = \min_{(\vec{x}_i, y_i) \in S} \left| \frac{1}{\|\vec{w}\|} (\vec{w} \cdot \vec{x}_i + b) \right|$$

- **Combine:**

$$\max_{\vec{w}, b, \gamma} \gamma \quad \text{s.t.} \quad \gamma = \min_{(\vec{x}_i, y_i) \in S} \left[\frac{y_i}{\|\vec{w}\|} (\vec{w} \cdot \vec{x}_i + b) \right]$$

We can rewrite the minimization as a set of constraints:

$$\max_{\vec{w}, b, \gamma} \gamma \quad \text{s.t.} \quad \forall (\vec{x}_i, y_i) \in S : \frac{y_i}{\|\vec{w}\|} (\vec{w} \cdot \vec{x}_i + b) \geq \gamma$$

This problem is invariant to scaling of \vec{w} and b . We can fix the scale by setting $\|\vec{w}\| = 1/\gamma$:

$$\max_{\vec{w}, b} \frac{1}{\|\vec{w}\|} \quad \text{s.t.} \quad \forall (\vec{x}_i, y_i) \in S : y_i (\vec{w} \cdot \vec{x}_i + b) \geq 1$$

This is equivalent to minimizing $\|\vec{w}\|$:

$$\min_{\vec{w}, b} \|\vec{w}\| \quad \text{s.t.} \quad \forall (\vec{x}_i, y_i) \in S : y_i (\vec{w} \cdot \vec{x}_i + b) \geq 1$$

For mathematical convenience, we minimize $\frac{1}{2} \vec{w} \cdot \vec{w}$:

$$\min_{\vec{w}, b} \frac{1}{2} \vec{w} \cdot \vec{w} \quad \text{s.t.} \quad \forall (\vec{x}_i, y_i) \in S : y_i (\vec{w} \cdot \vec{x}_i + b) \geq 1$$

This is the standard form of the hard-margin SVM optimization problem.

7.3 Hard Margin SVM

- **Goal:** Find the separating hyperplane with the largest distance (margin) to the closest training examples.
- **Optimization Problem:**

$$\min_{\vec{w}, b} \frac{1}{2} \vec{w} \cdot \vec{w} \quad \text{s.t.} \quad y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 \quad \forall i$$

- **Support Vectors:** Training examples that lie exactly on the margin, i.e.,

$$y_i(\vec{w} \cdot \vec{x}_i + b) = 1$$

- The margin γ is the distance from the hyperplane to the closest points (support vectors).
- Limitations: For some training data, there is no separating hyperplane. Complete separation (i.e. zero training error) can lead to suboptimal prediction error.

7.4 Soft Margin SVM

- **Idea:** Maximize margin and minimize training error. Allows some misclassification by introducing slack variables ξ_i .
- **Optimization Problem (Primal):**

$$\min_{\vec{w}, b, \xi} \frac{1}{2} \vec{w} \cdot \vec{w} + C \sum_{i=1}^n \xi_i$$

$$\text{s.t.} \quad y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad \forall i$$

- **Slack variable** ξ_i measures how much (x_i, y_i) fails to achieve the margin.
 - **Idea:** Slack variables ξ_i capture training error.
 - For any training example (\vec{x}_i, y_i) :
 - * $\xi_i \geq 1 \iff y_i(\vec{w} \cdot \vec{x}_i + b) \leq 0$ (error)
 - * $0 < \xi_i < 1 \iff 0 < y_i(\vec{w} \cdot \vec{x}_i + b) < 1$ (correct, but within margin)
 - * $\xi_i = 0 \iff y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1$ (correct)
 - The sum $\sum_i \xi_i$ is an upper bound on the number of training errors.
 - C controls the trade-off between margin size and training error.
- Examples with margin less than or equal to 1 are support vectors.

8 Kernels and Duality

Date: Sep 23, 2025

Non-linear Problems Some tasks have non-linear structure, and no hyperplane is sufficiently accurate.

For those problems, we can extend the feature space. For example, for quadratic features, we can have:

- **Input Space:** $\vec{x} = (x_1, x_2)$ (2 attributes)
- **Feature Space:** $\Phi(\vec{x}) = (x_1^2, x_2^2, x_1, x_2, x_1x_2, 1)$ (6 attributes)

SVM with Feature Map Support Vector Machines (SVMs) can use a feature map $\Phi(\vec{x})$ to transform the input into a higher-dimensional space, making non-linear problems linearly separable.

- **Training:** SVMs solve the following optimization problem:

$$\begin{aligned} \text{minimize: } P(\vec{w}, b, \vec{\xi}) &= \frac{1}{2} \vec{w} \cdot \vec{w} + C \sum_{i=1}^n \xi_i \\ \text{subject to: } y_i [\vec{w} \cdot \Phi(\vec{x}_i) + b] &\geq 1 - \xi_i \quad \forall i \\ \xi_i &> 0 \quad \forall i \end{aligned}$$

- **Classification:** The decision function is:

$$h(\vec{x}) = \text{sign} [\vec{w}^* \cdot \Phi(\vec{x}) + b]$$

Problems:

- **Computational Challenge:** Mapping to high-dimensional feature spaces can be computationally expensive.
- **Overfitting:** Using many features increases the risk of overfitting, so regularization and careful kernel choice are important.

8.1 Dual Perceptron

(Batch) Perceptron Algorithm

- Initialize $\vec{\alpha} = \vec{0}$ (the dual variables)
- Repeat for E epochs:
 - For $i = 1$ to m :
 - * If $y_i \left(\sum_{j=1}^m \alpha_j y_j (\vec{x}_j \cdot \vec{x}_i) \right) \leq 0$ then $\alpha_i = \alpha_i + 1$
- The prediction for a new \vec{x} is:

$$h(\vec{x}) = \text{sign} \left(\sum_{j=1}^m \alpha_j y_j (\vec{x}_j \cdot \vec{x}) \right)$$

$$\vec{w} \cdot \vec{x}_i = \left(\sum_{j=1}^m \alpha_j y_j \vec{x}_j \right) \cdot \vec{x}_i = \sum_{j=1}^m \alpha_j y_j (\vec{x}_j \cdot \vec{x}_i)$$

Instead of tracking the weight, we track the number of updates made by each training example. The hyperplane output by the Perceptron can always be expressed as a linear combination of the training data.

Primal vs. Dual Representation

- Primal Representation:
 - Weight vector $\vec{w} \in \mathbb{R}^d$
 - Threshold $b \in \mathbb{R}$
- Dual Representation:
 - Training data $(\vec{x}_1, y_1), \dots, (\vec{x}_m, y_m)$
 - Vector of dual variables $\vec{\alpha}$
 - Threshold $b \in \mathbb{R}$

What is Duality good for?

1. Dual variables give insight into the data.
2. If dimensionality d of \vec{x} is large, working in the dual representation can be more efficient if $\vec{x}_i \cdot \vec{x}_j$ is efficient.
3. Duality lets us prove theorems about the generalization error of the classifier.

8.2 Dual SVM

- Primal Optimization Problem:

$$\begin{aligned} \text{minimize: } P(\vec{w}, b, \vec{\xi}) &= \frac{1}{2} \vec{w} \cdot \vec{w} + C \sum_{i=1}^n \xi_i \\ \text{subject to: } y_i [\vec{w} \cdot \vec{x}_i + b] &\geq 1 - \xi_i \quad \forall i \\ \xi_i &\geq 0 \quad \forall i \end{aligned}$$

- Dual Optimization Problem:

$$\begin{aligned} \text{maximize: } D(\vec{\alpha}) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i y_j \alpha_i \alpha_j (\vec{x}_i \cdot \vec{x}_j) \\ \text{subject to: } \sum_{i=1}^n y_i \alpha_i &= 0 \\ 0 \leq \alpha_i &\leq C \quad \forall i \end{aligned}$$

- **Theorem:** If $(\vec{w}^*, b^*, \vec{\xi}^*)$ is the solution of the Primal and $\vec{\alpha}^*$ is the solution of the Dual, then

$$\vec{w}^* = \sum_{i=1}^m \alpha_i^* y_i \vec{x}_i \quad \text{and} \quad P(\vec{w}^*, b^*, \vec{\xi}^*) = D(\vec{\alpha}^*)$$

Dual variable α_i is proportional to force on data point. More formally, Dual variable α_i^* indicates the “influence” of training example (\vec{x}_i, y_i) .

- $\vec{w}^* = \sum_{i=1}^m \alpha_i^* y_i \vec{x}_i$
- **Definition:** (\vec{x}_i, y_i) is a *support vector* (SV) if and only if $\alpha_i^* > 0$.
- If $\xi_i^* > 0$, then $\alpha_i^* = C$.
- If $0 \leq \alpha_i^* < C$, then $\xi_i^* = 0$.
- If $0 < \alpha_i^* < C$, then $y_i (\vec{x}_i \cdot \vec{w}^* + b^*)$ (functional margin) = 1.

Leave-One-Out Error and Support Vectors Leave-one-out (LOO) cross validation is a good estimate of the generalization error for large n :

$$err_{loo}(A(S)) \approx err_P(A(S))$$

Theorem [Vapnik]: For any SVM,

$$err_{loo}(SVM(S)) \leq \frac{1}{m} \#SV$$

where $\#SV$ is the number of support vectors.

Theorem [Vapnik]: For a homogeneous hard-margin SVM,

$$err_{loo}(SVM(S)) \leq \frac{1}{m} \frac{R^2}{\gamma^2}$$

where

$$R^2 = \max_{i \in [1..m]} \vec{x}_i \cdot \vec{x}_i$$

and γ is the margin on the training sample S .

8.3 Non-Linear Rules through Kernels

Kernel Trick Instead of explicitly mapping \vec{x} to a high-dimensional feature space, we use a **kernel function** $K(\vec{x}, \vec{z})$ that computes the inner product in the feature space:

$$K(\vec{x}, \vec{z}) = \Phi(\vec{x}) \cdot \Phi(\vec{z})$$

This allows us to run algorithms that depend only on dot products without ever computing $\Phi(\vec{x})$ directly.

Polynomial Kernel Example For $\vec{x} = (x_1, x_2)$, the degree-2 polynomial kernel is:

$$K(\vec{x}, \vec{z}) = (\vec{x} \cdot \vec{z} + 1)^2$$

This corresponds to the feature map:

$$\Phi(\vec{x}) = (x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, 1)^\top$$

so that

$$K(\vec{x}, \vec{z}) = \Phi(\vec{x}) \cdot \Phi(\vec{z})$$

SVM with Kernel

- **Dual Optimization with Kernel:**

$$\text{maximize: } D(\vec{\alpha}) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y_i y_j \alpha_i \alpha_j K(\vec{x}_i, \vec{x}_j)$$

$$\begin{aligned} \text{subject to: } & \sum_{i=1}^n y_i \alpha_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$

- **Classification Rule:**

$$h(\vec{x}) = \text{sign} \left(\sum_{i=1}^n \alpha_i y_i K(\vec{x}_i, \vec{x}) + b \right)$$

- **Common Kernels:**

- Linear: $K(\vec{a}, \vec{b}) = \vec{a} \cdot \vec{b}$
- Polynomial: $K(\vec{a}, \vec{b}) = (\vec{a} \cdot \vec{b} + 1)^k$
- Radial Basis Function (RBF): $K(\vec{a}, \vec{b}) = \exp \left(-\gamma \|\vec{a} - \vec{b}\|^2 \right)$
- Sigmoid: $K(\vec{a}, \vec{b}) = \tanh(\gamma \vec{a} \cdot \vec{b} + c)$

The kernel trick allows SVMs to efficiently learn non-linear decision boundaries by implicitly operating in high-dimensional feature spaces.

8.4 Designing Kernels

Definition: Let X be a nonempty set. A function K is a **valid** kernel in X if for all m and all $x_1, \dots, x_m \in X$, it produces a Gram matrix

$$G_{ij} = K(x_i, x_j)$$

that is symmetric

$$G = G^T$$

and positive semi-definite

$$\forall \vec{\alpha} : \vec{\alpha}^T G \vec{\alpha} \geq 0$$

Any inner product is a kernel. Most properties of inner products also hold for kernels.

How to Construct Kernels

Theorem: Let K_1 and K_2 be valid kernels over $X \times X$, $\alpha \geq 0$, $0 \leq \lambda \leq 1$, f a real-valued function on X , $\Phi : X \rightarrow \mathbb{R}^N$ with a kernel K_3 over $\mathbb{R}^N \times \mathbb{R}^N$, and K a symmetric positive semi-definite matrix. Then the following functions are valid kernels:

$$K(\vec{x}, \vec{z}) = \lambda K_1(\vec{x}, \vec{z}) + (1 - \lambda) K_2(\vec{x}, \vec{z})$$

$$K(\vec{x}, \vec{z}) = \alpha K_1(\vec{x}, \vec{z})$$

$$K(\vec{x}, \vec{z}) = K_1(\vec{x}, \vec{z}) K_2(\vec{x}, \vec{z})$$

$$K(\vec{x}, \vec{z}) = f(\vec{x}) f(\vec{z})$$

$$K(\vec{x}, \vec{z}) = K_3(\Phi(\vec{x}), \Phi(\vec{z}))$$

$$K(\vec{x}, \vec{z}) = \vec{x}^T K \vec{z}$$

These closure properties allow us to construct new valid kernels from existing ones.

8.5 Properties of SVMs with Kernels

- **Expressiveness**

- SVMs with kernels can represent any boolean function (for appropriate choice of kernel).
- SVMs with kernels can represent any sufficiently “smooth” function to arbitrary accuracy (for appropriate choice of kernel).

- **Computational**

- Objective function has no local optima (only one global optimum).
- Independent of dimensionality of feature space.

- **Generalization Error**

- Low leave-one-out error if dual solution is sparse.
- Low leave-one-out error if $\frac{R^2}{\gamma^2}$ is small.

- **Design decisions**

- Kernel type and parameters.
- Value of C .

9 Regularized Linear Models

Date: Sep 25, 2025

9.1 ERM Learning

- **Examples:** KNN, decision trees, Perceptron, SVM.
- **Modelling Step:** Select a family of classification rules \mathcal{H} to consider (hypothesis space, features).
- **Training Principle:**
 - Given training sample $(\vec{x}_1, y_1), \dots, (\vec{x}_n, y_n)$
 - Find $h \in \mathcal{H}$ with lowest training error
 - This is called *Empirical Risk Minimization (ERM)*
- **Argument:** Low training error leads to low prediction error, if overfitting is controlled (*generalization*).

9.2 Bayes Decision Rule (0/1 loss)

- **Assumption:** The decision setting is known: $P(X, Y) = P(Y | X)P(X)$.
- **Goal:** For a given instance \vec{x} , choose \hat{y} to minimize prediction error under 0/1 loss

$$L_{0/1}(\hat{y}, y) = \begin{cases} 1, & \hat{y} \neq y \\ 0, & \hat{y} = y \end{cases}.$$

- **Rule:**

$$h_{\text{Bayes}}(\vec{x}) = \arg \max_{y \in \mathcal{Y}} P(Y = y | X = \vec{x}).$$

9.3 Decision via Bayes Risk

- **Bayes Risk** (expected loss of classifier h under distribution P and loss L):

$$\text{Err}_P(h) = \mathbb{E}_{\vec{x}, y \sim P(X, Y)}[L(h(\vec{x}), y)] = \mathbb{E}_{\vec{x} \sim P(X)}[\mathbb{E}_{y \sim P(Y|\vec{x})}[L(h(\vec{x}), y)]].$$

- **Bayes Decision Rule** minimizes the conditional risk:

$$h_{\text{Bayes}}(\vec{x}) = \arg \min_{\hat{y} \in \mathcal{Y}} \sum_{y \in \mathcal{Y}} L(\hat{y}, y) P(Y = y | X = \vec{x}).$$

- **Minimal risk for 0/1 loss:**

$$\text{Err}_P(h_{\text{Bayes}}) = \mathbb{E}_{\vec{x} \sim P(X)} \left[1 - \max_{y \in \mathcal{Y}} P(Y = y | X = \vec{x}) \right].$$

9.4 Learning Conditional Probabilities

- **Modeling:** Choose a parametric family $P(Y | X, \vec{w})$.
- **Training:** Given $(\vec{x}_i, y_i)_{i=1}^n$, find $\hat{\vec{w}}$ that best fits data:
 - *Maximum Likelihood (ML)* or *Maximum a Posteriori (MAP)*.
- **Classification:** Use Bayes rule with learned $P(Y | X, \hat{\vec{w}})$.
- **Argument:** If the learned conditional distribution is close to the true one, the induced decision rule is accurate.

9.5 Logistic Regression Model (binary $y \in \{-1, +1\}$)

- **Likelihood:** $P(Y_i = y | \vec{x}_i, \vec{w}) = \sigma(y \vec{w} \cdot \vec{x}_i)$, where $\sigma(z) = \frac{1}{1 + e^{-z}}$.
- **Symmetry:** $1 - \sigma(\vec{w} \cdot \vec{x}) = \sigma(-\vec{w} \cdot \vec{x})$.

9.6 Logistic Regression Training (Conditional MLE)

- **Objective:**

$$\hat{\vec{w}} = \arg \max_{\vec{w}} \prod_{i=1}^n P(y_i | \vec{x}_i, \vec{w}) = \arg \min_{\vec{w}} \sum_{i=1}^n \ln(1 + e^{-y_i \vec{w} \cdot \vec{x}_i}).$$

- **Derivation Sketch:**

1. i.i.d. data \Rightarrow factorized likelihood $\prod_i P(y_i | \vec{x}_i, \vec{w})$.
2. Plug logistic form $P(y_i | \vec{x}_i, \vec{w}) = \sigma(y_i \vec{w} \cdot \vec{x}_i)$.
3. Apply $-\ln(\cdot)$ (monotone decreasing) and log-product $\ln \prod = \sum \ln$.
4. Use $\sigma(z) = 1/(1 + e^{-z})$ to obtain logistic loss.

- **Prediction:** $h(\vec{x}) = \text{sign}(\hat{\vec{w}} \cdot \vec{x})$ (equivalently $\arg \max_y P(y | \vec{x}, \hat{\vec{w}})$).
- **Issue (separable data):** If data are linearly separable, the MLE drives $\|\vec{w}\| \rightarrow \infty$ since $y_i \vec{w} \cdot \vec{x}_i > 0$ can be increased without bound.

9.7 Regularized Logistic Regression: Probabilistic View

- **Likelihood:** Same as above.
- **Prior on weights:** $\vec{w} \sim \mathcal{N}(\vec{0}, \sigma^2 I)$, i.e.

$$P(\vec{w}) = \left(\frac{1}{\sigma \sqrt{2\pi}} \right)^d \exp \left(-\frac{\vec{w} \cdot \vec{w}}{2\sigma^2} \right).$$

- **MAP Training:**

$$\hat{\vec{w}} = \arg \max_{\vec{w}} P(\vec{w}) \prod_{i=1}^n P(y_i | \vec{x}_i, \vec{w}) = \arg \min_{\vec{w}} \left\{ \frac{\|\vec{w}\|_2^2}{2\sigma^2} + \sum_{i=1}^n \ln(1 + e^{-y_i \vec{w} \cdot \vec{x}_i}) \right\}.$$

- **Equivalent scaled form (ignore positive constants):**

$$\hat{\vec{w}} = \arg \min_{\vec{w}} \left\{ \frac{1}{2} \|\vec{w}\|_2^2 + \sigma^2 \sum_{i=1}^n \ln(1 + e^{-y_i \vec{w} \cdot \vec{x}_i}) \right\}.$$

- **Interpretation:** Gaussian prior $\Rightarrow \ell_2$ -regularization; controls $\|\vec{w}\|$ and prevents blow-up on separable data.

9.8 Regularized Logistic Regression: Summary

- **Training Objective (ridge-regularized log-loss):**

$$\hat{\vec{w}} = \arg \min_{\vec{w}} \frac{\lambda}{2} \|\vec{w}\|_2^2 + \sum_{i=1}^n \ln(1 + e^{-y_i \vec{w} \cdot \vec{x}_i}), \quad \text{with } \lambda = \frac{1}{\sigma^2}.$$

- **Prediction:** $h(\vec{x}) = \arg \max_y P(y \mid \vec{x}, \hat{\vec{w}}) = \text{sign}(\hat{\vec{w}} \cdot \vec{x})$.

9.9 Linear Regression Model

- **Data:** $\mathcal{S} = \{(\vec{x}_1, y_1), \dots, (\vec{x}_n, y_n)\}$ with $\vec{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.
- **Likelihood model:** $Y_i \mid \vec{x}_i, \vec{w} \sim \mathcal{N}(\vec{w} \cdot \vec{x}_i, \eta^2)$, i.e.

$$P(Y_i = y \mid \vec{x}_i, \vec{w}) = \frac{1}{\eta \sqrt{2\pi}} \exp\left(-\frac{(\vec{w} \cdot \vec{x}_i - y)^2}{2\eta^2}\right).$$

9.10 Linear Regression Training (Conditional MLE)

- **Objective:**

$$\hat{\vec{w}} = \arg \max_{\vec{w}} \prod_{i=1}^n P(y_i \mid \vec{x}_i, \vec{w}) = \arg \min_{\vec{w}} \sum_{i=1}^n \left[-\ln P(y_i \mid \vec{x}_i, \vec{w}) \right].$$

- **Derivation:**

$$\hat{\vec{w}} = \arg \min_{\vec{w}} \sum_{i=1}^n \left[-\ln\left(\frac{1}{\eta \sqrt{2\pi}}\right) + \frac{(\vec{w} \cdot \vec{x}_i - y_i)^2}{2\eta^2} \right] = \arg \min_{\vec{w}} \frac{1}{2\eta^2} \sum_{i=1}^n (\vec{w} \cdot \vec{x}_i - y_i)^2.$$

- **Conclusion:** MLE \iff least-squares (ignoring positive constants).
- **Prediction:** $h(\vec{x}) = \arg \max_y P(y \mid \vec{x}, \hat{\vec{w}}) = \hat{\vec{w}} \cdot \vec{x}$ (posterior mean for Gaussian).

9.11 Ridge Regression (MAP for a Gaussian prior)

- **Likelihood:** same as linear regression: $P(Y_i \mid \vec{x}_i, \vec{w}) = \mathcal{N}(\vec{w} \cdot \vec{x}_i, \eta^2)$.
- **Prior:** $\vec{w} \sim \mathcal{N}(\vec{0}, \sigma^2 I)$ with

$$P(\vec{w}) = \left(\frac{1}{\sigma \sqrt{2\pi}} \right)^d \exp\left(-\frac{\vec{w} \cdot \vec{w}}{2\sigma^2}\right).$$

- **MAP training:**

$$\hat{\vec{w}} = \arg \max_{\vec{w}} P(\vec{w}) \prod_{i=1}^n P(y_i | \vec{x}_i, \vec{w}) = \arg \min_{\vec{w}} \left\{ \frac{\vec{w} \cdot \vec{w}}{2\sigma^2} + \frac{1}{2\eta^2} \sum_{i=1}^n (\vec{w} \cdot \vec{x}_i - y_i)^2 \right\}.$$

- **Scaled form (drop positive constants):**

$$\hat{\vec{w}} = \arg \min_{\vec{w}} \frac{1}{2} \|\vec{w}\|_2^2 + C \sum_{i=1}^n (\vec{w} \cdot \vec{x}_i - y_i)^2, \quad \text{where } C = \frac{\sigma^2}{2\eta^2}.$$

- **Prediction:** $h(\vec{x}) = \hat{\vec{w}} \cdot \vec{x}$.

9.12 Discriminative Training: A Unifying View

- **Template objective:**

$$\min_{\vec{w}} R(\vec{w}) + C \sum_{i=1}^n L(\vec{w} \cdot \vec{x}_i, y_i).$$

- **Examples (classification):**

- *Soft-margin SVM:* $R(\vec{w}) = \frac{1}{2} \vec{w} \cdot \vec{w}$, $L(\hat{y}, y) = \max(0, 1 - y\hat{y})$.
- *Perceptron:* $R(\vec{w}) = 0$, $L(\hat{y}, y) = \max(0, -y\hat{y})$.
- *Reg. Logistic Regression:* $R(\vec{w}) = \frac{1}{2} \vec{w} \cdot \vec{w}$, $L(\hat{y}, y) = \ln(1 + e^{-y\hat{y}})$.

- **Examples (regression):**

- *Linear Regression:* $R(\vec{w}) = 0$, $L(\hat{y}, y) = (y - \hat{y})^2$.
- *Ridge Regression:* $R(\vec{w}) = \frac{1}{2} \vec{w} \cdot \vec{w}$, $L(\hat{y}, y) = (y - \hat{y})^2$.
- *Lasso:* $R(\vec{w}) = \lambda \sum_{j=1}^d |w_j|$, $L(\hat{y}, y) = (y - \hat{y})^2$.

9.13 “45 ML Algorithms on 1 Slide”: Building Blocks

- **Common loss functions L :**

- Hinge: $\max(0, 1 - y\hat{y})$
- Logistic: $\ln(1 + e^{-y\hat{y}})$
- Exponential: $e^{-y\hat{y}}$
- Squared error: $(y - \hat{y})^2$
- Absolute error: $|y - \hat{y}|$

- **Common regularizers R :**

- ℓ_2 : $\vec{w} \cdot \vec{w}$
- ℓ_1 : $\sum_{j=1}^d |w_j|$
- ℓ_0 : $|\{j : w_j \neq 0\}|$

- **Beyond linear scores $\vec{w} \cdot \vec{x}$:**

- *Kernels:* $\vec{w} \cdot \phi(\vec{x})$
- *Deep networks:* $f(\vec{x}; \vec{w})$
- *Boosting:* $\sum_j \alpha_j \text{Tree}_j(\vec{x})$

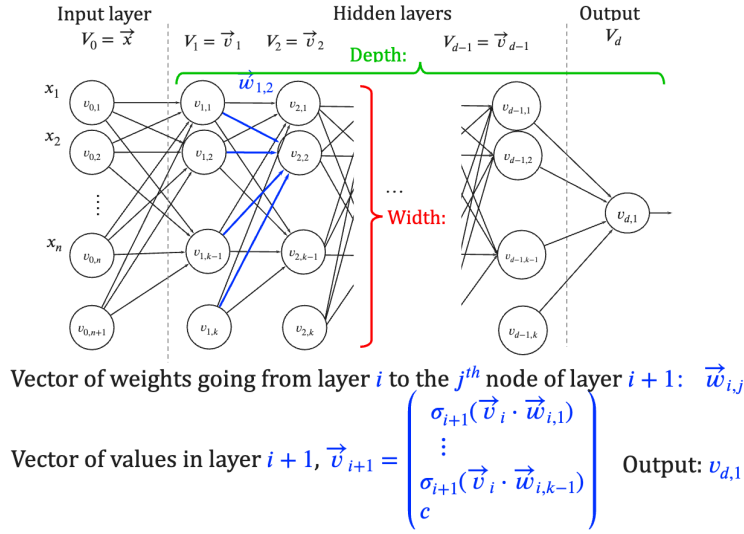
10 Optimization with Gradient Descent

Date: Sep 30, 2025

11 Neural Networks

Date: Oct 2, 2025

11.1 Multi-Layer Neural Networks



A multi-layer neural network (also called a feedforward neural network) consists of an input layer, one or more hidden layers, and an output layer. Each layer is made up of nodes (neurons), and each node in a layer is connected to every node in the next layer.

- Let \vec{v}_i be the vector of values in layer i .
- Let $\vec{w}_{i,j}$ be the vector of weights from layer i to the j^{th} node of layer $i + 1$.
- The value at node j in layer $i + 1$ is computed as:

$$v_{i+1,j} = \sigma_{i+1}(\vec{v}_i \cdot \vec{w}_{i,j})$$

where σ_{i+1} is the activation function for layer $i + 1$.

Concise Matrix Formulation

- Let W_i be the weight matrix for layer i , where each column j is the weight vector $\vec{w}_{i,j}$.
- Layers are fully connected; any missing edge has weight 0.
- The vector of activations for layer $i + 1$ is:

$$\vec{v}_{i+1} = \sigma_{i+1}(W_i^\top \vec{v}_i)$$

where σ_{i+1} is applied elementwise.

The output of a d -layer neural network can be written as a nested composition of linear transformations and activation functions:

$$\sigma_d \left(W_{d-1}^\top \cdots \sigma_3 \left(W_2^\top \sigma_2 \left(W_1^\top \sigma_1 \left(W_0^\top \vec{v}_0 \right) \right) \right) \cdots \right)$$

where each σ_i is applied elementwise, and W_i is the weight matrix for layer i .

Algorithm

Forward Propagation Algorithm:

Input: Neural Network with weight matrices W_0, W_1, \dots, W_{d-1} , activation functions $\sigma_1, \dots, \sigma_d$ and instance \vec{x}

$\vec{v}_0 = \vec{x}$

For $\ell = 1, \dots, d$

- $\vec{s}_\ell = W_{\ell-1}^\top \vec{v}_{\ell-1}$


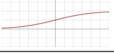
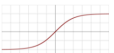

- $\vec{v}_\ell = \sigma_\ell(\vec{s}_\ell)$

End For

Output \vec{v}_d

11.2 Non-linear Activation Functions

Activation functions are applied to the nodes of a hidden layer in a neural network. They introduce non-linearity, allowing the network to learn complex patterns.

Name	Function	Gradient	Graph
Binary step	$\text{sign}(x)$	$\begin{cases} 0 & \text{if } x \neq 0, \\ \text{undefined} & \text{if } x = 0. \end{cases}$	
sigmoid	$\sigma(x) = \frac{1}{1 + \exp(-x)}$	$\sigma(x)(1 - \sigma(x))$	
Tanh	$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$1 - \tanh^2(x)$	
Rectified Linear (ReLU)	$\text{relu}(x) = \max(x, 0)$	$\begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases}$	

$\sigma(x)$ denotes an activation function, not necessarily sigmoid.

- **Binary step:** Outputs 0 or 1, not differentiable at $x = 0$.
- **Sigmoid:** Smooth, outputs between 0 and 1, can cause vanishing gradients.
- **Tanh:** Outputs between -1 and 1, zero-centered.
- **ReLU:** Simple, efficient, helps with vanishing gradient, but can "die" for negative inputs.

Deeper neural networks can express more complex functions only if we use non-linear activation.

If all activation functions are linear, then a multi-layer neural network is equivalent to a single-layer linear model. This is because a linear function of linear functions is still linear.

Example: Suppose the activation in the hidden layer is linear, $\sigma_1(x) = x$, and the output activation is non-linear, e.g., $\sigma_2(x) = \text{sign}(x)$. Then, the output can be written as:

$$\begin{aligned} v_{\text{out}} &= \text{sign}(\bar{w}_1 v_1 + \bar{w}_2 v_2) = \text{sign}(\bar{w}_1(\vec{x} \cdot \vec{w} + b) + \bar{w}_2(\vec{x} \cdot \vec{w}' + b')) \\ &= \text{sign}(\vec{z} \cdot \vec{x} + \beta) \end{aligned}$$

where $\vec{z} = \bar{w}_1 \vec{w} + \bar{w}_2 \vec{w}'$ and $\beta = \bar{w}_1 b + \bar{w}_2 b'$.

11.3 Universal Approximators

A fundamental result in neural networks is the **Universal Approximation Theorem**. It states that a feedforward neural network with a single hidden layer (i.e., a depth-2 network) and a sufficiently large number of neurons (width) can approximate any continuous function on \mathbb{R}^n to arbitrary accuracy, given appropriate activation functions (such as sigmoid or ReLU).

How large does the hidden layer need to be?

- For boolean functions, the required width can be as large as $\exp(n)$, where n is the input dimension.
- If we restrict ourselves to networks of polynomial size (i.e., width and number of parameters grow polynomially with n), then:
 - We cannot approximate all possible functions.
 - However, this restriction helps reduce the risk of overfitting.
 - This tradeoff is known as the **bias-variance tradeoff**: larger networks can fit more complex functions (low bias, high variance), while smaller networks may generalize better (high bias, low variance).
- Instead of fully connected layers, we can use structured networks (e.g., convolutional neural networks) to reduce the number of parameters and exploit structure in the data.