CS 3780 Notes

Jinzhou Wu

Contents

1 Supervised Learning and KNN

1.1 Supervised Learning

In supervised learning, we are given a dataset:

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

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:

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

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

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:** $\mathcal{Y} = \{1, 2, \ldots, k\}$
- **Regression:** $\mathcal{Y} \subseteq \mathbb{R}$
- **Structured Output:** $\mathcal{Y} =$ 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 $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, 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.

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

Weighted KNN for Regression

For regression problems, the prediction is a weighted average:

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

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:**
$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\|\mathbf{x} - \mathbf{x}'\|\right)$$
- **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

2.1 Inductive Learning

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

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

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

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

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

Select A that maximizes:

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

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

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)**.**

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

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

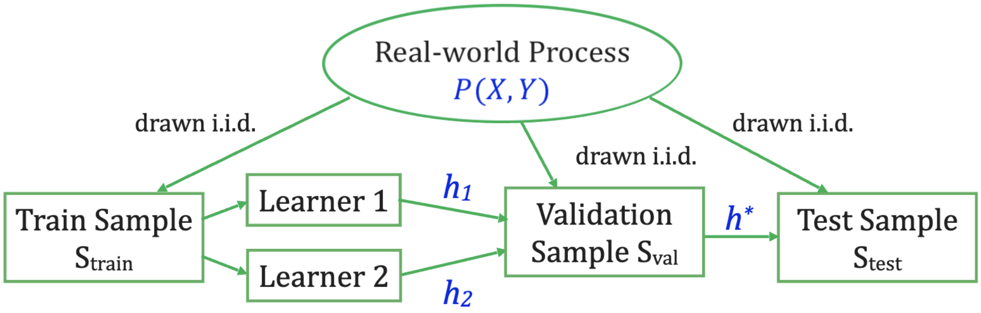
Definition (Prediction Error Decomposition)**.**

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

- **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

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 $A_1 \ldots A_l$ 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: $H' = \{h_1, \ldots, h_l\}$
  - Learning Algorithm: $h^* = \arg\min_{h \in H'} [err_{S_{val}}(h)]$

Typical ML Experiment

- Collect data $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$
- 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 $err_P(h) < \epsilon$.
- **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):**

$$err_P(h) \in \left[err_S(h) - 1.96\sqrt{\frac{p(1-p)}{m}},\ err_S(h) + 1.96\sqrt{\frac{p(1-p)}{m}} \right]$$

- With approximation $p \approx err_S(h)$ → 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:

$$err_P(h) \in \left[err_S(h) - \sqrt{\frac{-0.5\ln(\delta/2)}{m}},\ err_S(h) + \sqrt{\frac{-0.5\ln(\delta/2)}{m}} \right]$$

- This means that, with probability at least $1-\delta$, the true error $err_P(h)$ lies within the interval centered at the empirical error $err_S(h)$, 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:** $err_P(h_1) = err_P(h_2)$

  - 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

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:

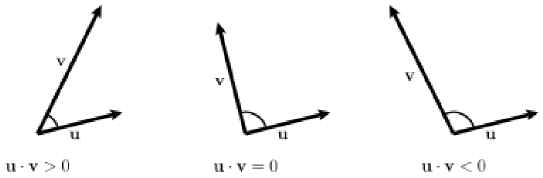
$$||\vec{x}|| = \sqrt{\sum_{i=1}^{d} x_i^2}$$

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 $h_{\vec{w},b} : \mathbb{R}^d \to \{-1, +1\}$ 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:**
$HS^d = \{h_{\vec{w},b} \mid \vec{w} \in \mathbb{R}^d, b \in \mathbb{R}\}$	$HS^{d+1}_{homog} = \{h_{\vec{w},b} \mid \vec{w} \in \mathbb{R}^{d+1}\}$
• $\vec{x}$	• $\vec{x}' = (\vec{x}, 1)$
• $\vec{w}, b$	• $\vec{w}' = (\vec{w}, b)$
• $\vec{w} \cdot \vec{x} + b$	• $\vec{w}' \cdot \vec{x}' = \vec{w} \cdot \vec{x} + b$

Without loss of generality, we will now focus on **homogenous linear classifiers** with $||w_i|| = 1$.

Consistent Linear Classifiers

- Data set of labelled instances $\mathcal{S} = \{(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots, (\vec{x}_m, y_m)\}$.
- 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 $(\vec{x}_i, y_i) \in \mathcal{S}$:

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

Larger margin $\implies$ Easier to find consistent linear classifier.

5.3 Perceptron Algorithm

Algorithm

The Perceptron Algorithm (homogeneous & batch)

- **Input:** training data $\mathcal{S} = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$
- **Initialize** $\vec{w}^{(0)} = (0, \ldots, 0)$ 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

6.1 Setup

We consider binary classification with labels $y_i \in \{-1, +1\}$ and (homogeneous) linear classifiers

$$h_{\vec{w}}(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x}).$$

Radius Assume there is an R such that $\|\vec{x}_i\| \leq R$ for all training points.

Separability with margin The dataset $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$ is **linearly separable with (geometric) margin** $\gamma > 0$ if there exists a unit vector $\vec{w}^\star$ such that

$$\|\vec{w}^\star\| = 1 \qquad \text{and} \qquad y_i(\vec{w}^\star \cdot \vec{x}_i) \geq \gamma \ \ \forall i \in [m].$$

6.2 The Perceptron Algorithm (Homogeneous & Batch)

- Initialize $\vec{w}^{(0)} = \vec{0}$ and $t = 0$.
- While there exists an example $(\vec{x}_i, y_i)$ such that $y_i(\vec{w}^{(t)} \cdot \vec{x}_i) \leq 0$ (a mistake), update

$$\vec{w}^{(t+1)} = \vec{w}^{(t)} + y_i\vec{x}_i, \qquad t \leftarrow t + 1.$$

- Output $\vec{w}^{(t)}$.

Key insight: # mistakes = # updates (= t).

6.3 Convergence of Perceptron

Theorem (Perceptron mistake bound)**.** *Given a dataset $S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}$ and a radius R such that $\|\vec{x}_i\| \leq R$ for all $i \in [m]$. If S is linearly separable with margin γ, then Perceptron makes at most R^2/γ^2 updates before finding a consistent linear classifier.*

6.4 Complete Proof

Proof outline We follow the three-step outline from the slides:

1. Lower bound $\vec{w}^\star \cdot \vec{w}^{(t)}$ in terms of t and γ.
2. Upper bound $\|\vec{w}^{(t)}\|$ in terms of t and R.
3. Combine the two bounds using a cosine argument to get a contradiction if $t > R^2/\gamma^2$.

Step 1: $\vec{w}^\star \cdot \vec{w}^{(t)}$ is large Let $(\vec{x}_i, y_i)$ be the example on which we make the tth mistake (so we perform update t). Using the perceptron update and distributivity of the dot product,

$$\begin{aligned}\vec{w}^\star \cdot \vec{w}^{(t)} &= \vec{w}^\star \cdot \left(\vec{w}^{(t-1)} + y_i\vec{x}_i\right) \\ &= \vec{w}^\star \cdot \vec{w}^{(t-1)} + y_i(\vec{w}^\star \cdot \vec{x}_i) \\ &\geq \vec{w}^\star \cdot \vec{w}^{(t-1)} + \gamma,\end{aligned}$$

where the last inequality uses the margin assumption $y_i(\vec{w}^\star \cdot \vec{x}_i) \geq \gamma$. Recursing yields

$$\vec{w}^\star \cdot \vec{w}^{(t)} \geq \vec{w}^\star \cdot \vec{w}^{(0)} + t\gamma = t\gamma,$$

since $\vec{w}^{(0)} = \vec{0}$.

Step 2: $\|\vec{w}^{(t)}\|$ is not too large Let $(\vec{x}_i, y_i)$ be the example on which we make the tth mistake. Expand the squared norm after the update:

$$\begin{aligned}\|\vec{w}^{(t)}\|^2 &= \|\vec{w}^{(t-1)} + y_i\vec{x}_i\|^2 \\ &= (\vec{w}^{(t-1)} + y_i\vec{x}_i) \cdot (\vec{w}^{(t-1)} + y_i\vec{x}_i) \\ &= \|\vec{w}^{(t-1)}\|^2 + 2y_i(\vec{w}^{(t-1)} \cdot \vec{x}_i) + y_i^2\|\vec{x}_i\|^2 \\ &\le \|\vec{w}^{(t-1)}\|^2 + 0 + R^2.\end{aligned}$$

The inequality uses:

- Mistake condition: $y_i(\vec{w}^{(t-1)} \cdot \vec{x}_i) \le 0$.
- $y_i^2 = 1$ and radius bound $\|\vec{x}_i\|^2 \le R^2$.

Recursing gives

$$\|\vec{w}^{(t)}\|^2 \le \|\vec{w}^{(0)}\|^2 + tR^2 = tR^2, \qquad \text{so} \qquad \|\vec{w}^{(t)}\| \le \sqrt{t}\,R.$$

Step 3: Contradiction via cosine Consider the angle θ between $\vec{w}^\star$ and $\vec{w}^{(t)}$:

$$\cos(\theta) = \frac{\vec{w}^\star \cdot \vec{w}^{(t)}}{\|\vec{w}^\star\|\,\|\vec{w}^{(t)}\|} = \frac{\vec{w}^\star \cdot \vec{w}^{(t)}}{\|\vec{w}^{(t)}\|},$$

since $\|\vec{w}^\star\| = 1$. Using Steps 1 and 2,

$$\cos(\theta) \ge \frac{t\gamma}{\sqrt{t}R} = \sqrt{t}\,\frac{\gamma}{R}.$$

If $t > R^2/\gamma^2$, then $\sqrt{t}\,\gamma/R > 1$, implying $\cos(\theta) > 1$, which is impossible. Therefore $t \le R^2/\gamma^2$, proving the theorem. □

6.5 Remarks on Convergence

- The bound holds regardless of the order of examples; changing the order can change the final classifier and convergence speed, but not the worst-case bound.
- If we scale each $\vec{x}_i \leftarrow c\vec{x}_i$, then $R \leftarrow cR$ and $\gamma \leftarrow c\gamma$, so the ratio R^2/γ^2 is unchanged.
- The algorithm does not need to know γ; the analysis only uses its existence.

6.6 Online Perceptron

Setting Data arrives as a stream $(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots$. At each time step, predict using the current $\vec{w}$, observe y_i, and update if you made a mistake.

Algorithm Initialize $\vec{w}^{(0)} = \vec{0}$ and for $i = 1, 2, \ldots$:

$$\hat{y}_i = \text{sign}(\vec{w}^{(i-1)} \cdot \vec{x}_i), \qquad \text{if } y_i(\vec{w}^{(i-1)} \cdot \vec{x}_i) \le 0 \text{ then } \vec{w}^{(i)} = \vec{w}^{(i-1)} + y_i\vec{x}_i.$$

(Otherwise, $\vec{w}^{(i)} = \vec{w}^{(i-1)}$.)

Mistake bound If the stream has radius R and is separable with margin γ, then the number of online mistakes (equivalently, updates) is at most R^2/γ^2 by the same proof as above.

7 Support Vector Machines

- **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

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

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

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

7.2 Computing Optimal Hyperplanes

- **Requirement:** Zero training error.

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

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

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

- **Combine:**

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

We can rewrite the minimization as a set of constraints:

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

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

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

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

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

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

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

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):**

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

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

- **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

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:

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

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

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

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 $(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)$
  - 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:**

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

- **Dual Optimization Problem:**

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

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

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

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 $y_i\left(\vec{x}_i \cdot \vec{w}^* + b^*\right)$ (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:**

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

- **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 $x_1, \ldots, x_m \in X$, 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, $\Phi : X \to \mathbb{R}^N$ 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:

$$
\begin{aligned}
K(\vec{x}, \vec{z}) &= \lambda K_1(\vec{x}, \vec{z}) + (1-\lambda) K_2(\vec{x}, \vec{z}) \\
K(\vec{x}, \vec{z}) &= \alpha K_1(\vec{x}, \vec{z}) \\
K(\vec{x}, \vec{z}) &= K_1(\vec{x}, \vec{z}) K_2(\vec{x}, \vec{z}) \\
K(\vec{x}, \vec{z}) &= f(\vec{x}) f(\vec{z}) \\
K(\vec{x}, \vec{z}) &= K_3(\Phi(\vec{x}), \Phi(\vec{z})) \\
K(\vec{x}, \vec{z}) &= \vec{x}^T K \vec{z}
\end{aligned}
$$

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

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 $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$
  - 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: $P(X, Y) = P(Y \mid X)P(X)$.
- **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:**

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

9.3 Decision via Bayes Risk

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

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

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

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

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

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

9.4 Learning Conditional Probabilities

- **Modeling:** Choose a parametric family $P(Y \mid X, \vec{w})$.
- **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 $P(Y \mid X, \hat{\vec{w}})$.
- **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:** $P(Y_i = y \mid \vec{x}_i, \vec{w}) = \sigma(y\, \vec{w}\cdot\vec{x}_i)$, where $\sigma(z) = \dfrac{1}{1+e^{-z}}$.
- **Symmetry:** $1 - \sigma(\vec{w}\cdot\vec{x}) = \sigma(-\vec{w}\cdot\vec{x})$.

9.6 Logistic Regression Training (Conditional MLE)

- **Objective:**

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

- **Derivation Sketch:**
  1. i.i.d. data $\Rightarrow$ factorized likelihood $\prod_i P(y_i \mid \vec{x}_i, \vec{w})$.
  2. Plug logistic form $P(y_i \mid \vec{x}_i, \vec{w}) = \sigma(y_i \vec{w}\cdot\vec{x}_i)$.
  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 $\arg\max_y P(y \mid \vec{x}, \hat{\vec{w}})$).
- **Issue (separable data):** If data are linearly separable, the MLE drives $\|\vec{w}\| \to \infty$ 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:**

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

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

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

- **Interpretation:** Gaussian prior ⇒ ℓ_2-regularization; controls $\|\vec{w}\|$ and prevents blow-up on separable data.

9.8 Regularized Logistic Regression: Summary

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

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

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

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:**

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

- **Derivation:**

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

- **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:**

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

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

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

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

9.12 Discriminative Training: A Unifying View

- **Template objective:**

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

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

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

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

- **Common loss functions L:**
  - Hinge: $\max(0, 1 - y\hat{y})$
  - Logistic: $\ln\big(1 + e^{-y\hat{y}}\big)$
  - 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: $\big|\{j : w_j \neq 0\}\big|$

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

10 Optimization with Gradient Descent

10.1 Discriminative Training Objective

Many supervised learning methods fit parameters w by minimizing a regularized empirical risk:

$$\min_{w} \; R(w) + C \sum_{i=1}^{n} L(w \cdot x_i, y_i),$$

where:

- $R(w)$ is a regularizer,
- $L(\cdot, \cdot)$ is a loss,
- C controls the regularization tradeoff.

10.2 Minima and Convexity

Definition. A function $f : \mathbb{R}^d \to \mathbb{R}$ is **convex** if for all $w, w' \in \mathbb{R}^d$ and $\alpha \in [0, 1]$,

$$f(\alpha w + (1 - \alpha)w') \leq \alpha f(w) + (1 - \alpha)f(w').$$

For convex f, every local minimum is a global minimum.

10.3 Gradients

For differentiable $f : \mathbb{R}^d \to \mathbb{R}$, the **gradient** at w is

$$\nabla f(w) = \left(\frac{\partial f(w)}{\partial w_1}, \ldots, \frac{\partial f(w)}{\partial w_d} \right).$$

Definition. Critical points are $w_\star$ such that $\nabla f(w_\star) = 0$.

Critical points can be minima, maxima, or saddle points.

First-order characterization of convexity A differentiable multivariate function f is convex iff

$$f(w') \geq f(w) + \nabla f(w) \cdot (w' - w) \qquad \forall w, w'.$$

For a convex differentiable function, any critical points $w_\star$ with $\nabla f(w_\star) = 0$ is a global minimum.

10.4 Gradient Descent (GD)

Gradient descent iteratively steps in the negative gradient direction:

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla f\left(w^{(t)}\right),$$

where $\eta_t > 0$ is the step size (learning rate).

Algorithm

Gradient Descent

- **Input:** f, horizon T, step sizes $\eta_1, \ldots, \eta_T$
- Initialize $w^{(1)} = \vec{0}$ (or any start point)
- For $t = 1, \ldots, T$: $w^{(t+1)} = w^{(t)} - \eta_t \nabla f(w^{(t)})$
- **Output:** $w^{(T)}$ (common) or the average $\bar{w} = \frac{1}{T}\sum_{t=1}^{T} w^{(t)}$ (more stable)

For convex f, GD has strong theoretical guarantees; the choice of step size is important in theory and in practice.

10.5 Stochastic Gradient Descent (SGD)

When $\nabla f(w)$ is expensive to compute exactly, use a random vector g that is an unbiased gradient estimator:

$$\mathbb{E}[g \mid w] = \nabla f(w).$$

SGD uses g in place of the exact gradient:

$$w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}.$$

Step size is even more important for SGD than for GD.

10.6 Learning Linear Models with GD

For training data $S = \{(x_i, y_i)\}_{i=1}^{n}$, many regularized linear models minimize

$$L_S(w) = R(w) + \frac{C}{n}\sum_{i=1}^{n} L(w \cdot x_i, y_i).$$

For the regularized linear models discussed in class, $L_S(w)$ is convex, so GD/SGD are natural optimization choices.

10.7 Example: GD for SVM

Recall the (primal) SVM objective with hinge loss:

$$\min_{w} \frac{1}{2}\|w\|^2 + \frac{C}{n}\sum_{i=1}^{n} \max\left(0,\, 1 - y_i(w \cdot x_i)\right).$$

One (sub)gradient form gives the GD update:

$$w \leftarrow (1-\eta)w + \eta\frac{C}{n}\sum_{i=1}^{n} y_i x_i \, \mathbf{1}\{y_i(w \cdot x_i) < 1\}.$$

This resembles the perceptron update, but enforces a stricter *margin* condition.

10.8 Optimization for Large-Scale ML

Computing the full gradient can be expensive when:

- the dataset is large (n is large),
- the data/model are high-dimensional (d is large).

Idea: use fewer samples per update, replacing the full sum by an estimate.

10.9 SGD for Learning

Pick $(x, y) \in S$ uniformly at random and update using the single-example gradient:

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla R(w^{(t)}) - \eta_t C \nabla L(w^{(t)} \cdot x, y).$$

Why this works: if (x, y) is chosen uniformly, then

$$\mathbb{E}\left[\nabla R(w) + C\nabla L(w \cdot x, y) \mid w\right] = \nabla L_S(w),$$

so the per-example gradient is an unbiased estimator of the true gradient.

Proof. In SGD, we pick (x, y) uniformly at random from S and use the stochastic gradient:

$$g = \nabla R(w) + C\nabla L(w \cdot x, y)$$

We show that g is an unbiased estimator of $\nabla L_S(w)$:

$$\begin{aligned}\mathbb{E}[g \mid w] &= \nabla R(w) + C\,\mathbb{E}_{(x,y)}[\nabla L(w \cdot x, y)] \\ &= \nabla R(w) + C\,\frac{1}{n}\sum_{i=1}^{n} \nabla L(w \cdot x_i, y_i) \\ &= \nabla R(w) + \frac{C}{n}\sum_{i=1}^{n} \nabla L(w \cdot x_i, y_i) \\ &= \nabla L_S(w)\end{aligned}$$

□

Conclusion: The stochastic gradient g is an unbiased estimator of the true gradient $\nabla L_S(w)$.

In practice It is common to shuffle the dataset and iterate without replacement; sampling with replacement is easier to analyze.

10.10 Example: SGD for SVM

With (x, y) sampled uniformly, an SGD-style update for the SVM objective is:

$$w \leftarrow (1 - \eta)w + \eta C\, yx\, \mathbf{1}\{y(w \cdot x) < 1\}\,.$$

Equivalently:

- If $y(w \cdot x) < 1$, then $w \leftarrow (1 - \eta)w + \eta C yx$.
- Else, $w \leftarrow (1 - \eta)w$.

This is similar to perceptron, but with margin enforcement and geometric “shrinking”.

10.11 Practical Considerations

Non-differentiable Objectives

Non-differentiable convex functions still admit **subgradients** (tangents lying below the function). We can run GD/SGD with subgradients.

Mini-batches

Mini-batch SGD interpolates between SGD (batch size 1) and full GD (batch size n). For a random subset $S_t \subseteq S$:

$$w^{(t+1)} = w^{(t)} - \eta_t \nabla R(w^{(t)}) - \eta_t \frac{C}{|S_t|} \sum_{(x,y)\in S_t} \nabla L(w^{(t)} \cdot x, y).$$

Step Size

- Smaller step sizes: less stochasticity, more GD-like.
- Larger step sizes: more stochasticity (can help progress) but more uncertainty.
- Decaying step sizes are common in practice.

Conditioning

Well-conditioned objectives converge more easily than ill-conditioned ones (elongated level sets can cause slow progress).

Adaptive Gradients

AdaGrad changes the step size for each coordinate based on previous gradients, shrinking it more for parameters that have seen large gradients in the past. For all coordinates $i \in [d]$:

- **Derivative:** $g_{i,t} = \frac{\partial \mathcal{L}(w^{(t)})}{\partial w_i} = [\nabla \mathcal{L}(w^{(t)})]_i$
- **Update:**

  $$w_i^{(t+1)} = w_i^{(t)} - \eta \frac{g_{i,t}}{\sqrt{0.01 + \sum_{k=1}^{t} g_{i,k}^2}}$$

  where 0.01 is a stabilizer and the denominator accumulates the sum of squares of past derivatives.

Momentum Use an exponentially-weighted moving average of gradients to encourage consistent directions:

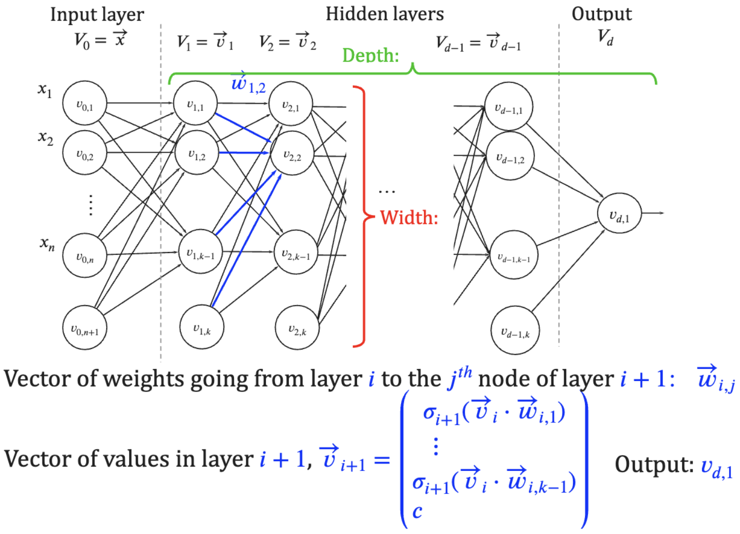
$$G^{(t)} = (1-\beta)G^{(t-1)} + \beta \nabla L_S(w^{(t)}), \qquad w^{(t+1)} = w^{(t)} - \eta G^{(t)}.$$

Non-convexity

Non-convex objectives are challenging in general, but SGD is widely and successfully used in non-convex settings (e.g. neural networks), and under specific assumptions SGD can be shown to converge to good minima.

11 Neural Networks

11.1 Multi-Layer Neural Networks



Vector of weights going from layer i to the j^{th} node of layer $i+1$: $\vec{w}_{i,j}$

Vector of values in layer $i+1$, $\vec{v}_{i+1} = \begin{pmatrix} \sigma_{i+1}(\vec{v}_i \cdot \vec{w}_{i,1}) \\ \vdots \\ \sigma_{i+1}(\vec{v}_i \cdot \vec{w}_{i,k-1}) \\ c \end{pmatrix}$ Output: $v_{d,1}$

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 jth 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:

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

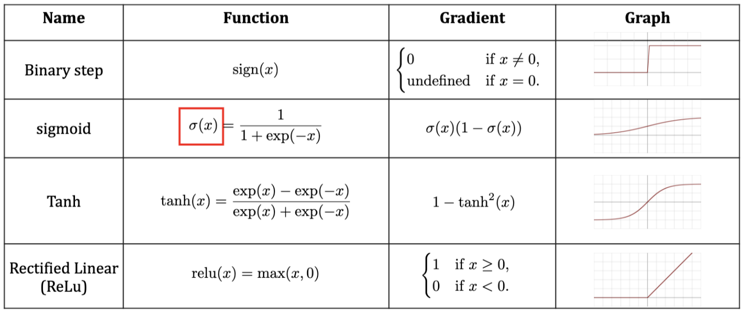
where each σ_i is applied elementwise, and W_i is the weight matrix for layer i.

Algorithm

```
Forward Propagation Algorithm:
Input: Neural Network with weight matrices $W_0, W_1, \ldots, W_{d-1}$, activation functions $\sigma_1, \ldots, \sigma_d$ and instance $\vec{x}$
$\vec{v}_0 = \vec{x}$
For $\ell = 1, \ldots, d$
    • $\vec{s}_\ell = W_{\ell-1}^\top \vec{v}_{\ell-1}$
    • $\vec{v}_\ell = \sigma_\ell(\vec{s}_\ell)$
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	$\mathrm{sign}(x)$	$\begin{cases} 0 & \text{if } x \neq 0, \\ \text{undefined} & \text{if } x = 0. \end{cases}$	
sigmoid	$\boxed{\sigma(x)} = \frac{1}{1+\exp(-x)}$	$\sigma(x)(1-\sigma(x))$	
Tanh	$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$1-\tanh^2(x)$	
Rectified Linear (ReLu)	$\mathrm{relu}(x) = \max(x, 0)$	$\begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{if } x < 0. \end{cases}$	

$\sigma(x)$ 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., $\sigma_2(x) = \mathrm{sign}(x)$. Then, the output can be written as:

$$v_{\text{out}} = \mathrm{sign}(\bar{w}_1 v_1 + \bar{w}_2 v_2) = \mathrm{sign}\left(\bar{w}_1(\vec{x} \cdot \vec{w} + b) + \bar{w}_2(\vec{x} \cdot \vec{w}' + b')\right)$$

$$= \mathrm{sign}(\vec{z} \cdot \vec{x} + \beta)$$

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

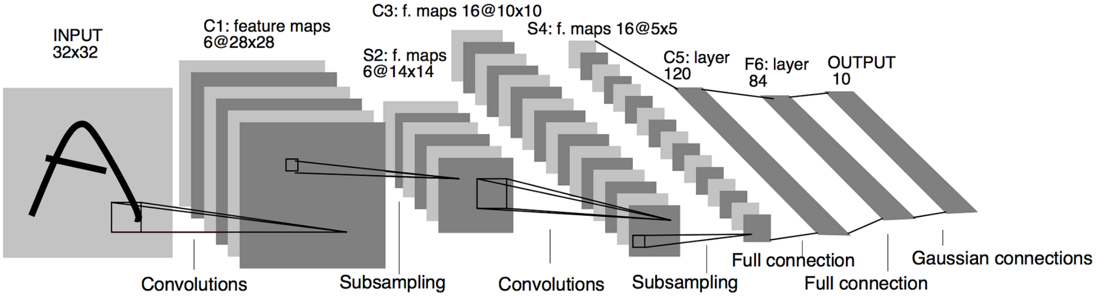
11.3 Universal Approximators

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.

11.4 Convolutional Neural Networks



If we're using neural networks for image data, for a grayscale (1 channel) image of size 1920 × 1080, a fully connected first layer, assuming 1000 hidden units, would require $1920 \times 1080 \times 1000 \approx 2$ billion parameters, which is impractical.

- CNNs are designed to process data with a grid-like topology, such as images.
- Instead of connecting every input pixel to every neuron in the next layer, CNNs use **local connections** via small filters (kernels) that slide over the input.
- Each filter detects specific features (e.g., edges, textures) by performing a **convolution** operation.
- The same filter (set of weights) is applied across the entire input, enabling **weight sharing** and reducing the number of parameters.
- The result of applying a filter is called a **feature map**.

- Multiple filters are used in each layer to detect different features.
- CNNs typically alternate between convolutional layers, non-linear activation functions, and pooling (downsampling) layers.

Convolutions Given an $n \times n$ input matrix and an $f \times f$ filter (kernel), the convolution operation produces an $(n - f + 1) \times (n - f + 1)$ output matrix. Each entry in the output is computed by sliding the filter over the input, multiplying corresponding entries, and summing the results.

- The (i, j)-th entry of the output is the sum of pairwise multiplications of the filter and the $f \times f$ submatrix of the input whose top-left entry is at (i, j).
- The filter is applied across the input with a specified **stride** (how many positions the filter moves at each step; stride 1 means move by 1 entry).
- Convolutions allow for **local feature extraction** and **weight sharing** (the same filter is used everywhere).

In a convolution layer, filter's weights and biases are parameters that will be learned.

Example: If the input is a 5×5 matrix and the filter is 3×3, the output will be 3×3. Each output entry is computed as the sum of elementwise products between the filter and the corresponding 3×3 patch of the input.

Padding Convolution shrinks the size of the matrix, losing corner information. We can pad 0's around the original matrix.

If we pad on each side by $p = \frac{f-1}{2}$ 0's (filter size is often odd), after convolution, the matrix size is still $n \times n$.

Pooling Pooling layers reduce the spatial dimensions (width and height) of the input, helping to control overfitting and reduce computation.

- **Max Pooling:** For each region (e.g., 2×2), output the maximum value.
- **Average Pooling:** For each region, output the average value.
- Pooling is typically applied with a stride equal to the pooling window size (e.g., non-overlapping 2×2 regions).
- No parameters are learned in pooling layers.
- Pooling helps make the representation approximately invariant to small translations in the input.

Pooling is similar to convolution in that it operates on local regions, but instead of a learned filter, it uses a fixed function (max or average).

12 Attention and Transformers

12.1 Text as Sequences

Neural networks operate on vectors, so we first convert text into a sequence of vectors:

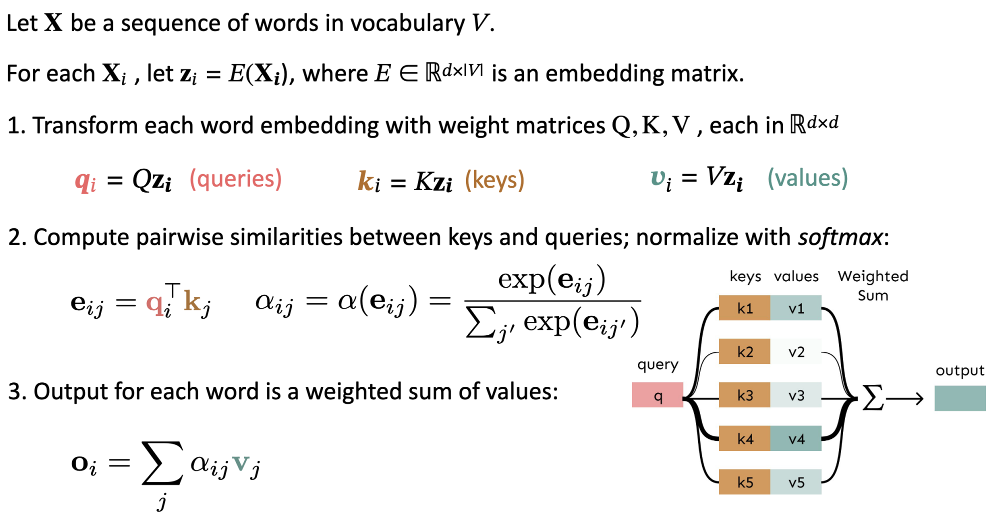
- **Tokenize:** $x = (x_1, \dots, x_n)$ where each $x_i \in \mathcal{V}$ is a token from a vocabulary $\mathcal{V}$.
- **Embed:** an embedding table $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ maps each token to a vector $z_i = E[x_i] \in \mathbb{R}^d$.
- Collect embeddings into a matrix $Z \in \mathbb{R}^{n \times d}$ where row i is $z_i^\top$.

12.2 Attention

Key–Value View Attention can be viewed as fuzzy lookup in a key–value store:

- A **query** is matched against all **keys**.
- Similarities are normalized into weights (via softmax).
- The output is a weighted average of the corresponding **values**.

12.3 Self-Attention



Let $\mathbf{X}$ be a sequence of words in vocabulary V.

For each $\mathbf{X}_i$, let $\mathbf{z}_i = E(\mathbf{X}_i)$, where $E \in \mathbb{R}^{d\times|V|}$ is an embedding matrix.

1. Transform each word embedding with weight matrices Q, K, V , each in $\mathbb{R}^{d\times d}$

$q_i = Q\mathbf{z_i}$ (queries) $k_i = K\mathbf{z_i}$ (keys) $v_i = V\mathbf{z_i}$ (values)

2. Compute pairwise similarities between keys and queries; normalize with *softmax*:

$$\mathbf{e}_{ij} = \mathbf{q}_i^\top \mathbf{k}_j \qquad \alpha_{ij} = \alpha(\mathbf{e}_{ij}) = \frac{\exp(\mathbf{e}_{ij})}{\sum_{j'} \exp(\mathbf{e}_{ij'})}$$

3. Output for each word is a weighted sum of values:

$$\mathbf{o}_i = \sum_j \alpha_{ij} \mathbf{v}_j$$

In **self-attention**, queries/keys/values are all computed from the same sequence representation Z. Let

$$Q = ZW_Q, \quad K = ZW_K, \quad V = ZW_V$$

where $W_Q, W_K \in \mathbb{R}^{d \times d_k}$, $W_V \in \mathbb{R}^{d \times d_v}$, and Z is the concatenation of the input embeddings.

- Each word forms a **query** (Q) to ask what information it needs from the rest of the sequence.
- All words provide **keys** (K) describing what information they offer, and **values** (V) containing the content to share.
- Self-attention computes how well each query matches each key, then blends the values accordingly, letting each word build a context-aware representation from the whole sequence.

Scaled Dot-Product Attention

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

The softmax is applied row-wise so each position attends to all positions in the sequence.

12.4 Positional Embeddings

Self-attention is permutation-invariant in its inputs, so we must represent order. Standard practice: add a learned (or fixed) position vector $p_i \in \mathbb{R}^d$ to each token embedding:

$$\tilde{z}_i = z_i + p_i.$$

12.5 Multi-Head Self-Attention

Rather than computing a single attention operation in the full d-dimensional space, multi-head self-attention uses H parallel attention heads, each operating in a learned d/H-dimensional subspace.

- For each head h, the full d-dimensional input embeddings are linearly projected into queries, keys, and values: $Q^{(h)} = XW_Q^{(h)}, K^{(h)} = XW_K^{(h)}, V^{(h)} = XW_V^{(h)}, W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{d \times d/H}$.
- Each head computes self-attention independently in its projected subspace:

$$O^{(h)} = \mathrm{Attn}(Q^{(h)}, K^{(h)}, V^{(h)}) \in \mathbb{R}^{n \times d/H}.$$

- The head outputs are concatenated and mixed with an output projection:

$$O = \mathrm{Concat}(O^{(1)}, \dots, O^{(H)})W_O, \quad W_O \in \mathbb{R}^{d \times d}.$$

12.6 Transformer Blocks

The **Transformer** is a stack of L blocks. Each block contains:

1. Multi-headed self-attention
2. Residual connection + LayerNorm
3. Position-wise feed-forward network (FFN): two linear layers with ReLU in between, applied independently to each position: $\mathrm{FFN}(x) = \mathrm{ReLU}(xW_1 + b_1)W_2 + b_2$
4. Residual connection + LayerNorm

Residual Connections Residual connections stabilize deep networks by letting layers learn perturbations: $x \mapsto x + \mathrm{Layer}(x)$.

Layer Normalization LayerNorm normalizes each hidden vector to reduce variance across coordinates: $\mathrm{LayerNorm}(x) = \frac{x-\mu}{\sigma} \cdot \gamma + \beta, \quad$ where $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i,\ \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$

12.7 Common Hyperparameters

- Model/hidden width d
- FFN width m
- Head dimension d_k (and d_v)
- Number of heads H
- Depth L
- Context length n
- Optimizer, learning rate schedule, regularization, etc.

13 Scaling Laws

13.1 Motivation

Training large-scale models involves many design choices (architecture, depth/width, optimizer, batch size, dataset size, etc.). **Scaling laws** are simple, predictive rules for model performance that let us:

- Tune on small runs and extrapolate to large runs.
- Compare configurations under fixed compute budgets.
- Plan data/compute requirements for a target performance level.

13.2 Data Scaling Laws

A **data scaling law** is a formula to predict error based on dataset size. Typically, we expect:

- **Monotone, logistic-like curves:** As dataset size increases, error decreases in a predictable way.
- **Power-law decay:** In the main regime, error decreases as a power of dataset size:

  $$\text{Error} = \Theta\left(\frac{1}{n^{\alpha}}\right)$$

  or equivalently,

  $$\log(\text{Error}) = -\alpha \log(n)$$

  where n is the dataset size and α is the scaling exponent.
- **Regions:** For small datasets, error may plateau (small data region). For large datasets, error approaches an irreducible minimum (irreducible error region).

Estimating the Slope Estimate α by fitting a line to $(\log n, \log \text{Error}(n))$ across runs, e.g. via least squares.

13.3 Toy Example: Estimating a Mean

Suppose $x_1, \ldots, x_n \sim \mathcal{N}(\mu, \sigma^2)$ are i.i.d. samples. We estimate the mean by

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The mean squared error of this estimator is:

$$\mathbb{E}\left[\|\hat{\mu} - \mu\|^2\right] = \text{Var}[\hat{\mu}] = \text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} x_i\right) = \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}[x_i] = \frac{\sigma^2}{n}$$

Taking logs:

$$\log(\text{Error}) = \log(\sigma^2) - \log(n)$$

So, plotting log(Error) vs. $\log(n)$ yields a straight line with slope -1. In general, for power-law scaling Error $\sim n^{-\alpha}$, the slope is $-\alpha$ in log-log scale.

13.4 Engineering with Scaling Laws

Once a scaling law is fit, it can be used to:

- Predict the performance of a larger run.
- Decide whether it is worth collecting more data.
- Decide which ablations/hyperparameter searches are most informative under limited compute.

13.5 Model/Compute Scaling

Scaling laws can also compare architecture choices (e.g. depth), optimizer choices, and batch sizes by fitting performance as a function of:

- **Parameters** (model size)
- **Dataset size**
- **Training compute**

Scaling may have diminishing returns past a certain point (perfect scaling vs. ineffective scaling).

Compute Tradeoffs For a fixed compute budget, we often face tradeoffs between model size and data size. Scaling laws help navigate these tradeoffs, but can be difficult to compute correctly.

13.6 Cautions

- Early scaling-law fits can be wrong when extrapolated (famously, early OpenAI scaling laws were later revised).
- For a fixed training compute budget, the optimal model is smaller and trained on much more data (about 20 tokens per parameter). Older models like GPT-3 were undertrained by this standard.
- In practice, most compute is spent on inference (serving), not training. Inference cost scales with model size and tokens served.
- Modern models are trained with far more tokens per parameter than Chinchilla-optimal (e.g., LLaMA 3: 215, Mistral 7B: 110), meaning they are "overtrained" relative to the training-optimal regime.
- Overtraining (high tokens/param) increases upfront training cost but allows for much smaller models, reducing inference cost per token at scale.
- If a model will be used heavily, it is rational to pay more for training to save on long-term inference costs. For low-usage models, Chinchilla-optimal training is more cost-effective.
- Chinchilla is optimal if only training compute matters, but when inference dominates, overtraining smaller models is preferred for lower total cost.

14 Statistical Learning Theory

Let $(X, Y) \sim P$ be drawn from an unknown data-generating distribution. We observe a sample $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ drawn i.i.d. from P and consider a hypothesis class H.

Errors For a loss $\Delta(\hat{y}, y) \in [0, 1]$ (e.g. 0/1 loss), define:

$$\mathrm{err}_S(h) = \frac{1}{m}\sum_{i=1}^{m} \Delta(h(x_i), y_i), \qquad \mathrm{err}_P(h) = \mathbb{E}_{(x,y)\sim P}\left[\Delta(h(x), y)\right].$$

Psychic abilities thought experiment Suppose there are N students; each student "guesses" an m-bit string. If a student is not psychic, assume each bit is guessed independently and incorrectly with probability p. Then the probability a fixed student guesses the *entire* string is $(1-p)^m$.

If students guess independently, the probability that *at least one* student gets the whole string right is

$$1 - (1 - (1-p)^m)^N,$$

e.g., for $N = 200$, $p = 0.5$, and $m = 4$, this is about 0.999. In particular, a simple upper bound (union bound: $\Pr\left(\bigcup_{i=1}^N A_i\right) \leq \sum_{i=1}^N \Pr(A_i)$, where each A_i is the event that student i guesses correctly) is

$$\Pr(\text{some student guesses correctly}) \leq N(1-p)^m.$$

So to make "zero errors" convincing of telepathic abilities at confidence $1-\delta$, it is enough to pick m such that

$$N(1-p)^m \leq \delta.$$

Overfitting intuition If the hypothesis class is extremely expressive (e.g. high-degree polynomials, or "pick the best analyst among many"), then it is easy to find a hypothesis that fits the observed data well by chance. Generalization bounds quantify this effect via complexity terms like $|H|$, VC dimension, or Rademacher complexity.

14.1 Finite Hypothesis Spaces and Zero Training Error

Setting Assume H is finite and there exists $h^\star \in H$ with $\mathrm{err}_P(h^\star) = 0$ (the **realizable** case). Assume the learning algorithm returns a hypothesis $\hat{h}$ with zero training error:

$$\mathrm{err}_S(\hat{h}) = 0.$$

What is the probability that the generalization error of $\hat{h}$ is large, i.e. $\mathrm{err}_P(\hat{h}) \geq \epsilon$?

Generalization Error Bound (Finite H, Zero Training Error)

Proof intuition: define the bad hypotheses and bound the probability that any of them achieves zero training error.

Define the set of **bad** hypotheses:

$$B = \{h \in H : \mathrm{err}_P(h) \geq \epsilon\}.$$

For a fixed $h \in B$, the probability it makes *no* training mistakes is

$$\Pr(\mathrm{err}_S(h) = 0) \leq (1-\epsilon)^m \leq e^{-\epsilon m}.$$

Here we used a useful formula: $1 + x \leq e^x$ for all $x \in \mathbb{R}$. By the union bound,

$$\Pr(\exists h \in B : \mathrm{err}_S(h) = 0) \leq |B|e^{-\epsilon m} \leq |H|e^{-\epsilon m}.$$

Since ERM returns some h with $\mathrm{err}_S(h) = 0$, we obtain

$$\Pr\Big(\mathrm{err}_P(\hat{h}) \geq \epsilon\Big) \leq |H|e^{-\epsilon m}.$$

Sample Complexity Bound

To ensure $\Pr(\mathrm{err}_P(\hat{h}) \leq \epsilon) \geq 1 - \delta$, it suffices that

$$|H|e^{-\epsilon m} \leq \delta \qquad \iff \qquad m \geq \frac{1}{\epsilon}\left(\ln|H| + \ln\frac{1}{\delta}\right).$$

Example: Boolean Conjunctions

Let H be the class of Boolean conjunctions with ℓ literals over N binary variables. Then

$$|H| = \binom{N}{\ell} 2^{\ell},$$

since we choose which ℓ variables appear, and for each chosen variable decide whether it is negated.

Probably Approximately Correct (PAC) Learning

Definition. A concept class C is **PAC-learnable** by a learning algorithm $\mathcal{L}$ using hypothesis class H and a sample S of n examples drawn i.i.d. from some fixed distribution $P(X)$ and labeled by a concept $c \in C$, if for sufficiently large n,

$$P\left(\mathrm{err}_P(h_{\mathcal{L}(S)}) \leq \epsilon\right) \geq 1 - \delta$$

for all $c \in C$, $\epsilon > 0$, $\delta > 0$, and $P(X)$. The algorithm $\mathcal{L}$ is required to run in polynomial time in $1/\epsilon$, $1/\delta$, n, the size of the training examples, and the size of c.

A concept class is a set of functions mapping from the instance space to $\{0, 1\}$. Often $C = H$.

The sample complexity of PAC-learning a finite hypothesis class H in the realizable case is

$$m = O\left(\frac{1}{\epsilon}\left(\ln|H| + \ln\frac{1}{\delta}\right)\right).$$

14.2 Finite Hypothesis Spaces and Non-Zero Training Error

When models have the capacity to fit any dataset, the danger is that they may fail to generalize.

In the **agnostic** setting, we do not assume there is a perfect hypothesis in H. We assume the learning algorithm returns some hypothesis $\hat{h}$ (e.g. via ERM) that minimizes training error. We want a bound that relates $\mathrm{err}_P(h)$ and $\mathrm{err}_S(h)$ for *every* $h \in H$.

What is the probability that the generalization error of $\hat{h}$ exceeds the sample error by more than ϵ, i.e. $|\mathrm{err}_P(\hat{h}) - \mathrm{err}_S(\hat{h})| \geq \epsilon$?

Hoeffding's Inequality

Let $Z_1, \dots, Z_m$ be i.i.d. random variables with values in $[0,1]$ and sample mean $\bar{Z} = \frac{1}{m}\sum_{i=1}^m Z_i$. Then

$$\Pr\big(|\bar{Z} - \mathbb{E}[\bar{Z}]| \geq \epsilon\big) \leq 2e^{-2\epsilon^2 m}.$$

Finite H Generalization Bound (Non-Zero Training Error)

Fix $h \in H$ and define $Z_i = \Delta(h(x_i), y_i) \in [0,1]$. Then $\bar{Z} = \mathrm{err}_S(h)$ and $\mathbb{E}[\bar{Z}] = \mathrm{err}_P(h)$, so

$$\Pr(\mathrm{err}_P(h) - \mathrm{err}_S(h) \geq \epsilon) \leq 2e^{-2\epsilon^2 m}.$$

Apply the union bound over H:

$$\Pr(\exists h \in H : \mathrm{err}_P(h) - \mathrm{err}_S(h) \geq \epsilon) \leq 2|H|e^{-2\epsilon^2 m}.$$

Equivalently, with probability at least $1-\delta$, *simultaneously for all $h \in H$*,

$$\mathrm{err}_P(h) \leq \mathrm{err}_S(h) + \sqrt{\frac{\ln(2|H|) + \ln(1/\delta)}{2m}}.$$

Structural Risk Minimization (SRM)

If we have nested hypothesis classes $H_1 \subset H_2 \subset \cdots$ of increasing complexity, SRM trades off:

- **Training error** (typically decreases with larger classes)
- **Complexity penalty** (typically increases with larger classes)

to reduce both underfitting and overfitting.

Rademacher Complexity

Let $\epsilon_1, \dots, \epsilon_m$ be i.i.d. Rademacher variables (each ±1 with probability 1/2).

Definition. The empirical Rademacher complexity of H on a sample $S = \{(x_i, y_i)\}_{i=1}^m$ is

$$\mathcal{R}_m(H, S) = \mathbb{E}_\epsilon \left[\sup_{h \in H} \frac{1}{m} \sum_{i=1}^m \epsilon_i \, \Delta(h(x_i), y_i) \right].$$

The (expected) Rademacher complexity is $\mathcal{R}_m(H) = \mathbb{E}_S[\mathcal{R}_m(H, S)]$.

Intuition: No matter what randomization I have (expectation: average over all assignments of ϵ), there are still hypotheses that are able to capture the positive epsilons (noise) better than chance, that means the hypothesis class is very expressive and complex.

Inability to Correlate with Noise Guarantees Generalization

Theorem. *For any hypothesis class H,*

$$\mathbb{E}\left[\sup_{h \in H} (\mathrm{err}_P(h) - \mathrm{err}_S(h)) \right] \leq 2\mathcal{R}_m(H)$$

Proof Sketch: Let S' be a "ghost sample" consisting of m additional i.i.d. draws from P.

- $\mathbb{E}[\mathrm{err}_{S'}(h) - \mathrm{err}_S(h) \mid S] = \mathrm{err}_P(h) - \mathrm{err}_S(h)$
- In the expression $\mathrm{err}_{S'}(h) - \mathrm{err}_S(h)$, every time h mislabels a point in $S \cup S'$, it is equally likely to occur with a $+1$ or -1 coefficient.

14.3 Infinite Hypothesis Spaces and VC Dimension

When H is infinite, $|H|$ is not meaningful. Instead we measure an *effective size* of H on m points.

Effective Number of Hypotheses and the Growth Function

How many distinct labelings can H realize on a sample of size m? Define

$$H[S] = \{(h(x_1), \dots, h(x_m)) : h \in H\}, \qquad \Pi_H(m) = \max_{|S|=m} |H[S]|.$$

VC Dimension

Definition. H **shatters** a set of m points if it realizes all 2^m labelings on that set, i.e. $|H[S]| = 2^m$. The **VC dimension** $\mathrm{VC}(H)$ is the largest m such that some set of size m is shattered.

Equivalently, $\mathrm{VC}(H)$ is the largest m for which $\Pi_H(m) = 2^m$.

Sauer–Shelah (growth function bound) If $\mathrm{VC}(H) = d$, then for $m \geq d$,

$$\Pi_H(m) \leq \sum_{i=0}^{d} \binom{m}{i} \leq \left(\frac{em}{d}\right)^d.$$

Generalization via inability to correlate with noise

$$\mathbb{E}\left[\sup_{h \in H} (\mathrm{err}_P(h) - \mathrm{err}_S(h))\right] \leq 2\mathcal{R}_m(H).$$

Massart's Lemma (via the growth function)

$$\mathcal{R}_m(H) \leq \sqrt{\frac{2 \ln \Pi_H(m)}{m}}.$$

Example bounds from the slides:

- If H is the set of subintervals of a line, $\Pi_H(m) = \binom{m}{2} + m + 1$.
- For d-dimensional linear classifiers, $\Pi_H(m) \lesssim \left(\frac{em}{d}\right)^d$, giving an expected gap on the order of $O\left(\sqrt{\frac{d \log m}{m}}\right)$.

14.4 Epilogue: Rethinking Generalization

Modern deep networks can fit random labels (high capacity) yet generalize well on real labels, motivating more nuanced explanations of generalization beyond classical capacity measures alone.

15 Boosting and Ensemble Methods

15.1 Strong vs. Weak Learning

- **Strong learner:** for every distribution P and every $\epsilon > 0$, outputs h with $\mathrm{err}_P(h) \leq \epsilon$ (with probability $1 - \delta$).
- **Weak learner:** for every distribution P, outputs h that is slightly better than random guessing:

  $$\mathrm{err}_P(h) \leq \frac{1}{2} - \gamma$$

  for some fixed $\gamma > 0$ (with probability $1 - \delta$).

15.2 Boosting Idea

Boosting turns a weak learner into a strong learner by calling it repeatedly on reweighted versions of the training data and combining the resulting weak hypotheses via a weighted vote.

15.3 A Generic Boosting Recipe

Given training data $S = \{(x_i, y_i)\}_{i=1}^m$:

1. Initialize a distribution over examples, e.g. $P_1(i) = 1/m$.
2. For rounds $t = 1, \ldots, T$:
   - Train a weak learner on distribution P_t to obtain h_t.
   - Update the weights to emphasize examples that were misclassified by h_t.
3. Output a combined classifier of the form

   $$h_{\text{final}}(x) = \mathrm{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right),$$

   which is a weighted majority vote of the hypotheses.

15.4 AdaBoost

AdaBoost chooses weights and combination coefficients adaptively.

Algorithm

AdaBoost Algorithm (binary labels $y_i \in \{-1, +1\}$):

1. Initialize $P_1(i) = 1/m$.
2. For $t = 1, \ldots, T$:
   (a) Train weak learner on P_t to get h_t.
   (b) Weighted error:
   $$\epsilon_t = \Pr_{i \sim P_t}[h_t(x_i) \neq y_i] = \sum_{i=1}^{m} P_t(i)\mathbf{1}\{h_t(x_i) \neq y_i\}.$$
   (c) Set
   $$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right).$$
   (d) Update weights:
   $$P_{t+1}(i) = \frac{P_t(i)\exp(-\alpha_t y_i h_t(x_i))}{Z_t},$$
   where $Z_t = \sum_{i=1}^{m} P_t(i)\exp(-\alpha_t y_i h_t(x_i))$ normalizes P_{t+1} to sum to 1.

Derivation of α_t The coefficient α_t determines how much weight to assign to the t-th weak hypothesis h_t. It is derived by minimizing the weighted exponential loss on the training data.

- The combined classifier after t rounds is:
$$H_t(x) = \text{sign}\left(\sum_{s=1}^{t} \alpha_s h_s(x)\right)$$
- The exponential loss on the training set is:
$$L = \sum_{i=1}^{m} \exp\left(-y_i \sum_{s=1}^{t} \alpha_s h_s(x_i)\right)$$
- At round t, we want to choose α_t to minimize L, holding previous α_s fixed.
- Let $P_t(i)$ be the normalized weight of example i at round t:
$$P_t(i) = \frac{\exp\left(-y_i \sum_{s=1}^{t-1} \alpha_s h_s(x_i)\right)}{Z_{t-1}}$$
- The loss after adding h_t is:
$$L = Z_{t-1} \sum_{i=1}^{m} P_t(i) \exp\left(-\alpha_t y_i h_t(x_i)\right)$$

- Split the sum into correctly and incorrectly classified examples:

$$L = Z_{t-1}\left[\sum_{i:y_i=h_t(x_i)} P_t(i)e^{-\alpha_t} + \sum_{i:y_i\neq h_t(x_i)} P_t(i)e^{\alpha_t}\right]$$

- Let $\epsilon_t = \sum_{i:y_i\neq h_t(x_i)} P_t(i)$ be the weighted error.
- The loss becomes:

$$L = Z_{t-1}\left[(1-\epsilon_t)e^{-\alpha_t} + \epsilon_t e^{\alpha_t}\right]$$

- Minimize L with respect to α_t:

$$\frac{dL}{d\alpha_t} = 0 \implies -(1-\epsilon_t)e^{-\alpha_t} + \epsilon_t e^{\alpha_t} = 0$$

$$\epsilon_t e^{\alpha_t} = (1-\epsilon_t)e^{-\alpha_t}$$

$$e^{2\alpha_t} = \frac{1-\epsilon_t}{\epsilon_t}$$

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

This choice of α_t ensures that hypotheses with lower error receive higher weight in the final ensemble.

15.5 Generalization Behavior

Although boosting increases model complexity as T grows, AdaBoost often does not overfit early; empirically, training error can drop rapidly while test error may keep improving.

15.6 Ensemble Methods

Ensembles combine multiple models to improve performance: $h_{\text{ensemble}}(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$.

15.7 Bagging (Bootstrap Aggregating)

Bagging reduces variance by training on bootstrap resamples and averaging/voting.

Algorithm

Bagging Algorithm:

1. For $t = 1, \ldots, T$:
    - Sample a bootstrap dataset S_t by sampling with replacement from S.
    - Train a hypothesis h_t on S_t.
2. Output

$$h_{\text{bag}}(x) = \text{sign}\left(\sum_{t=1}^{T} h_t(x)\right)$$

16 Online Learning

16.1 Online Classification Model

We receive examples sequentially and update a hypothesis after each round:

- For $t = 1, \ldots, T$:
  1. Receive x_t
  2. Predict $\hat{y}_t = h_t(x_t)$
  3. Observe true label y_t
  4. Update h_{t+1} based on (x_t, y_t)

16.2 Online Perceptron

Initialize $w = \vec{0}$ and iterate:

$$\hat{y}_t = \operatorname{sign}(w \cdot x_t), \qquad \text{if } y_t(w \cdot x_t) \leq 0 \text{ then } w \leftarrow w + y_t x_t.$$

Theorem. *If $\|x_t\| \leq R$ for all t and there exists a unit vector $w^\star$ with margin $y_i(w^\star \cdot x_i)\gamma > 0$ such that $y_t(w^\star \cdot x_t) \geq \gamma$ for all t, then the online perceptron makes at most $(R/\gamma)^2$ mistakes.*

16.3 Expert Learning and Regret

We have N experts $H = \{h_1, \ldots, h_N\}$. In each round t:

- Expert i incurs loss $\Delta_{t,i}$.
- The algorithm selects an expert (possibly randomly) and incurs corresponding loss.

Regret Let the cumulative loss of expert i be $L_i = \sum_{t=1}^{T} \Delta_{t,i}$ and the algorithm's cumulative loss be L_A. The (static) regret is

$$\text{Regret}(T) = L_A - \min_{i \in [N]} L_i.$$

16.4 Halving Algorithm (Realizable Expert Setting)

Assume there exists a perfect expert with zero mistakes. Maintain a **version space** $V_t \subseteq H$ of experts consistent so far and predict by majority vote over V_t. Whenever the algorithm makes a mistake, remove all experts in V_t that predicted incorrectly.

Theorem. *If a perfect expert exists, the Halving algorithm makes at most $\log_2 N$ mistakes.*

16.5 Weighted Majority Algorithm

Initialize weights $w_1(i) = 1$ for all experts and choose a penalty factor $\beta \in (0, 1)$. At time t, predict by weighted majority vote. After observing losses, downweight incorrect experts:

$$w_{t+1}(i) = \begin{cases} \beta\, w_t(i) & \text{if expert } i \text{ is incorrect at time } t \\ w_t(i) & \text{otherwise.} \end{cases}$$

Theorem. *For $\beta = \frac{1}{2}$, the Weighted Majority algorithm makes at most*

$$M_{\mathrm{WM}} \leq 2.4\,(M^\star + \log_2 N)$$

mistakes, where $M^\star$ is the number of mistakes of the best expert in hindsight.

Proof. Let $W_t = \sum_{i=1}^N w_t(i)$ be the total weight at time t. Initially, $W_1 = N$. Each mistake reduces the total weight by at least a factor of $\frac{3}{4}$ because at least half the weight is on incorrect experts (since the algorithm predicts incorrectly by weighted majority), and those weights are multiplied by $\frac{1}{2}$:

$$W_{t+1} \leq \frac{1}{2} \cdot \frac{W_t}{2} + \frac{W_t}{2} = \frac{3}{4} W_t.$$

Thus, after M_{WM} mistakes, the total weight satisfies

$$W_{T+1} \leq N \left(\frac{3}{4}\right)^{M_{\mathrm{WM}}}.$$

On the other hand, the weight of the best expert after T rounds is at least

$$w_{T+1}(i^\star) = \left(\frac{1}{2}\right)^{M^\star}.$$

Combining these,

$$\left(\frac{1}{2}\right)^{M^\star} \leq W_{T+1} \leq N \left(\frac{3}{4}\right)^{M_{\mathrm{WM}}}.$$

Taking logarithms and rearranging gives the desired bound. □

16.6 Exponentiated Gradient / Hedge

For general bounded losses $\Delta_{t,i} \in [0,1]$, maintain weights and choose a distribution

$$p_t(i) = \frac{w_t(i)}{\sum_{j=1}^N w_t(j)}.$$

Sample an expert according to p_t and update

$$w_{t+1}(i) = w_t(i) \exp(-\eta \Delta_{t,i}).$$

With an appropriate learning rate η, the expected regret is $O(\sqrt{2T \log |H|})$, where $\Delta \in [0,1]$ and $\eta = \sqrt{\frac{2 \log |H|}{T}}$ so average regret vanishes as $T \to \infty$.

17 Generative Classifiers and Naive Bayes

17.1 Bayes Decision Rule (0/1 Loss)

Assume the data-generating distribution $P(X, Y)$ is known. To minimize 0/1 loss, predict

$$h(x) = \arg\max_{y \in \mathcal{Y}} P(y \mid x).$$

Using Bayes' rule,

$$P(y \mid x) = \frac{P(x \mid y)P(y)}{P(x)} \quad \Longrightarrow \quad h(x) = \arg\max_{y} P(x \mid y)P(y).$$

17.2 Generative Modeling vs. Discriminative Modeling

- **Generative:** model $P(x, y)$ (joint distribution), then use Bayes rule to get $P(y \mid x)$.
- **Conditional / discriminative:** model $P(y \mid x)$ directly (e.g. logistic regression).
- **ERM:** pick h to minimize empirical loss on training data.

17.3 Multivariate (Bernoulli) Naive Bayes

Assume $x \in \{0, 1\}^d$ and conditional independence of features given the class:

$$P(x \mid y) = \prod_{j=1}^{d} P(x_j \mid y).$$

The classifier is

$$h(x) = \arg\max_{y \in \{+1,-1\}} P(y) \prod_{j=1}^{d} P(x_j \mid y).$$

In log-space:

$$h(x) = \arg\max_{y} \left[\log P(y) + \sum_{j=1}^{d} \log P(x_j \mid y) \right].$$

Estimators and Smoothing

- **Estimating $P(Y)$:** Use the fraction of positive/negative examples in the training data:

  $$\hat{P}(Y = y) = \frac{m_y}{m}$$

  where m_y is the number of training examples in class y and m is the total number of examples.

- **Estimating $P(X \mid Y)$:**
  - **Maximum Likelihood Estimate:**

    $$\hat{P}(X_i = x \mid Y = y) = \frac{\#(X_i = x, y)}{m_y}$$

    where $\#(X_i = x, y)$ is the number of training examples in class y where feature X_i takes value x.

- **Laplace Smoothing:**

$$\hat{P}(X_i = x \mid Y = y) = \frac{\#(X_i = x, y) + 1}{m_y + |X_i|}$$

where $|X_i|$ is the number of different values that feature X_i can take.

Connection to Linear Classification

$$h_{\text{naive}}(\vec{x}) = \text{sign}\left[\ln \frac{P(Y = +1)}{P(Y = -1)} + \sum_{i=1}^{N} \ln \frac{P(X_i = x_i \mid Y = +1)}{P(X_i = x_i \mid Y = -1)}\right]$$

This shows that the Naive Bayes classifier is a linear classifier in the feature space, i.e.,

$$h(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b)$$

where

$$b = \ln \frac{P(Y = +1)}{P(Y = -1)}, \quad \vec{w}_i = \ln \frac{P(X_i = x_i \mid Y = +1)}{P(X_i = x_i \mid Y = -1)}$$

17.4 Multinomial Naive Bayes (Text)

For documents represented as a bag of words, assume each word is drawn i.i.d. from a class-specific distribution over the vocabulary. If a document has word counts $c(w)$, then

$$P(x \mid y) \propto \prod_{w \in \mathcal{V}} P(w \mid y)^{c(w)},$$

and classification again uses $\arg\max_y P(y)P(x \mid y)$.

17.5 Modeling Continuous Variables: Linear Discriminant Analysis (LDA)

Assume the feature vectors of each class come from a spherical Gaussian distribution:

$$P(X = \vec{x} \mid Y = y) = \mathcal{N}(\vec{\mu}_y, \Sigma_y)$$

where

$$P(X = \vec{x} \mid Y = y) = \frac{1}{(2\pi)^{N/2}|\Sigma_y|^{1/2}} \exp\left(-\frac{1}{2}(\vec{x} - \vec{\mu}_y)^T \Sigma_y^{-1} (\vec{x} - \vec{\mu}_y)\right)$$

For LDA, we assume $\Sigma_+ = \Sigma_- = \Sigma = \text{diag}(1)$ (features are normalized to variance 1).

Classification Rule

$$h_{\text{LDA}}(\vec{x}) = \arg\max_{y \in \mathcal{Y}} \left[P(X = \vec{x} \mid Y = y)P(Y = y)\right]$$

This simplifies to:

$$h_{\text{LDA}}(\vec{x}) = \arg\max_{y \in \mathcal{Y}} \left[\ln P(Y = y) - \frac{1}{2}\|\vec{x} - \vec{\mu}_y\|^2\right]$$

which is a linear classifier:

$$h(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b)$$

where

$$\vec{w} = \vec{\mu}_+ - \vec{\mu}_-, \quad b = \ln \frac{P(Y = +1)}{P(Y = -1)} - \frac{1}{2}(\vec{\mu}_+ + \vec{\mu}_-) \cdot (\vec{\mu}_+ - \vec{\mu}_-)$$

Parameter Estimation

- **Class priors:**
$$\hat{P}(Y = +1) = \frac{m_+}{m}, \quad \hat{P}(Y = -1) = \frac{m_-}{m}$$
- **Class means:**
$$\vec{\mu}_+ = \frac{1}{m_+} \sum_{i:y_i=+1} \vec{x}_i, \quad \vec{\mu}_- = \frac{1}{m_-} \sum_{i:y_i=-1} \vec{x}_i$$

18 Structured Prediction

18.1 Predicting Structured Outputs

In structured prediction, the output y is a complex object rather than a single label/number, e.g.

- Multi-label classification (predict a set of labels)
- Speech transcription (predict a sentence)
- Semantic segmentation (predict a label for each pixel)
- Part-of-speech (POS) tagging (predict a tag sequence)

18.2 Supervised Structured Prediction

We are given data $S = \{(x_i, y_i)\}_{i=1}^m$ where $y_i \in \mathcal{Y}$ is structured and a loss $\Delta(y, \hat{y})$. The Bayes decision rule for 0/1 loss is

$$h(x) = \arg\max_{y \in \mathcal{Y}} P(x, y) = \arg\max_{y \in \mathcal{Y}} P(y \mid x).$$

18.3 Hidden Markov Models (HMMs) for Sequence Labeling

For a sequence $x = (x_1, \ldots, x_T)$ and tags $y = (y_1, \ldots, y_T)$:

- **Start probabilities:** $P(y_1)$
- **Transitions:** $P(y_t \mid y_{t-1})$ (Markov assumption)
- **Emissions:** $P(x_t \mid y_t)$

The joint distribution factorizes as

$$P(x, y) = P(y_1) P(x_1 \mid y_1) \prod_{t=2}^{T} P(y_t \mid y_{t-1}) P(x_t \mid y_t).$$

Estimating HMM Parameters Using maximum likelihood from labeled sequences:

- $P(y_1 = s) = \dfrac{\#\text{ sequences starting in } s}{\#\text{ sequences}}$
- $P(y_t = s \mid y_{t-1} = s') = \dfrac{\#(s' \to s)}{\#(s' \text{ occurs})}$
- $P(x_t = o \mid y_t = s) = \dfrac{\#(o \text{ emitted in } s)}{\#(s \text{ occurs})}$

Smoothing (as in Naive Bayes) is often needed.

HMM Assumptions

- **Markov assumption:** The next state depends only on the previous state.
  - Example (POS Tagging): The next tag depends only on the previous tag, not on earlier/later tags or words.
- **Emission assumption:** The current emission depends only on the current state.
  - Example (POS Tagging): The current word depends only on the current tag, not on earlier/later tags or words.

18.4 Inference: The Viterbi Algorithm

To efficiently compute the most likely sequence of tags $y = (y_1, \dots, y_l)$ for a given observation sequence $x = (x_1, \dots, x_l)$, we want:

$$\hat{y} = \arg \max_{y \in \{y_1, \dots, y_l\}} \left\{ P(y_1) P(x_1 \mid y_1) \prod_{i=2}^{l} P(x_i \mid y_i) P(y_i \mid y_{i-1}) \right\}$$

Directly maximizing over all possible sequences is computationally infeasible for large l, since the number of possible sequences grows exponentially.

Viterbi Algorithm: The Viterbi algorithm is a dynamic programming method to efficiently find the most likely sequence. It works by recursively computing the probability of the most likely path ending in each state at each position.

Define the Viterbi score $\delta_y(i)$ as the probability of the most likely sequence of states ending in state y at position i:

$$\delta_y(1) = P(Y_1 = y) P(X_1 = x_1 \mid Y_1 = y)$$

$$\delta_y(i+1) = \max_{v \in \{s_1, \dots, s_k\}} \delta_v(i) P(Y_{i+1} = y \mid Y_i = v) P(X_{i+1} = x_{i+1} \mid Y_{i+1} = y)$$

At each step, we keep track of the maximizing previous state (the ”backpointer”) for each state and position.

Backtracking: After filling in the dynamic programming table, we recover the most likely sequence by tracing back the maximizing states from the final position.

Complexity: The Viterbi algorithm runs in $O(l|S|^2)$ time, where l is the sequence length and $|S|$ is the number of possible states/tags.

18.5 Beyond Sequences

More general structured prediction includes graph labeling problems (e.g. Markov random fields), where inference can be more complex than the chain-structured Viterbi case.

19 Clustering

19.1 Unsupervised Learning

In unsupervised learning, we are given unlabeled data $\{x_i\}_{i=1}^n$ and aim to discover structure. **Clustering** partitions examples into groups such that:

- points within a cluster are similar,
- points across clusters are dissimilar.

19.2 Similarity Measures

Common similarity/distance measures include:

- **Euclidean distance:** $\|x - x'\|_2$
- ℓ_1 **distance:** $\|x - x'\|_1$
- **Cosine similarity:** $\dfrac{x \cdot x'}{\|x\| \, \|x'\|}$

19.3 K-means Clustering

Objective Assign each point to one of K clusters with centroids $\mu_1, \ldots, \mu_K$. Let $r(i) \in \{1, \ldots, K\}$ be the cluster assignment of x_i. The K-means objective is:

$$\min_{\{\mu_k\}, r} \sum_{i=1}^{n} \left\| x_i - \mu_{r(i)} \right\|_2^2 .$$

Alternating Minimization

- **Assignment step:** with fixed centroids, set $r(i) = \arg\min_k \|x_i - \mu_k\|_2$.
- **Centroid step:** with fixed assignments, set μ_k to the mean of points assigned to cluster k.

These steps monotonically decrease the objective but can get stuck in local minima.

Algorithm

K-means (Lloyd's Algorithm):

1. Initialize centroids $\mu_1, \ldots, \mu_K$ (e.g. randomly).
2. Repeat until convergence:
    - Assign each x_i to its nearest centroid.
    - Update each centroid to the mean of its assigned points.

19.4 Generative Clustering: Gaussian Mixture Models (GMMs)

Generative Model Assume a datapoint $\mathbf{x}$ is generated as follows:

- Choose a cluster z from $\{1, \dots, K\}$ such that $p(z = k) = \pi_k$
- Given z, sample $\mathbf{x}$ from a Gaussian: $p(\mathbf{x} \mid z = k) = \mathcal{N}(\mathbf{x} \mid \mu_k, \Sigma_k)$

This defines a joint distribution:

$$p(\mathbf{x}, z) = p(z)p(\mathbf{x} \mid z)$$

with parameters π_k, μ_k, Σ_k for $k = 1, \dots, K$.

Posterior Inference Given a data point $\mathbf{x}$, the probability that $z = k$ is:

$$p(z = k \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid z = k)p(z = k)}{p(\mathbf{x})}$$

where

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x} \mid \mu_k, \Sigma_k)$$

Parameter Estimation Maximize the likelihood of the data:

$$\arg \max_{\pi_1,\dots,\pi_K,\, \mu_1,\dots,\mu_K,\, \Sigma_1,\dots,\Sigma_K} \sum_{n=1}^{N} \log p(\mathbf{x}^{(n)})$$

where

$$p(\mathbf{x}^{(n)}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}^{(n)} \mid \mu_k, \Sigma_k)$$

No closed form solution → use Expectation Maximization (EM).

Expectation Maximization (EM) Algorithm

- **Initialization:** randomly initialize parameters.
- **E-step:** Compute soft assignments (responsibilities):

$$r_k^{(n)} = p(z^{(n)} = k \mid \mathbf{x}^{(n)}) = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)} \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}^{(n)} \mid \mu_j, \Sigma_j)}$$

- **M-step:** Update parameters using responsibilities:

$$\mu_k = \frac{\sum_{n=1}^{N} r_k^{(n)} \mathbf{x}^{(n)}}{\sum_{n=1}^{N} r_k^{(n)}}, \qquad \pi_k = \frac{1}{N} \sum_{n=1}^{N} r_k^{(n)}, \qquad \Sigma_k = \frac{\sum_{n=1}^{N} r_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^\top}{\sum_{n=1}^{N} r_k^{(n)}}$$

- Repeat E and M steps until convergence.

20 Principal Component Analysis (PCA)

20.1 Setup and Goal

Dimensionality reduction Given a dataset $x_1, \dots, x_n \in \mathbb{R}^d$, we want a lower-dimensional representation that preserves as much structure as possible. PCA finds an *orthonormal* set of directions (principal components) that capture the most variance.

Coordinate vs. principal components Projecting onto coordinate axes corresponds to discarding features (e.g. keep x and drop y). Principal components instead choose *data-dependent* orthogonal directions (e.g. $y = x$ and $y = -x$ in 2D) that better summarize correlated features.

20.2 Principal Components

Principal components are vectors $v_1, \dots, v_d \in \mathbb{R}^d$ such that:

- Unit length: $v_i^\top v_i = 1$ for all i.
- Orthogonal: $v_i^\top v_j = 0$ for $i \neq j$.
- v_1 is the direction of greatest variance in the data.
- v_2 is the direction of greatest variance subject to being orthogonal to v_1.
- etc.

20.3 Calculating Variance in a Direction

Centering Assume data are centered:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i = 0.$$

If the data are not centered, subtract the mean from each data point: $x_i \leftarrow x_i - \mu$.

Variance along a unit direction For a unit vector v with $v^\top v = 1$, the projection of x_i onto v is $v^\top x_i$, and the variance in direction v is

$$\mathrm{Var}(v) = \frac{1}{n} \sum_{i=1}^{n} (v^\top x_i)^2.$$

20.4 The PCA Objective

First principal component

$$v_1 = \arg \max_{v \colon v^\top v = 1} \frac{1}{n} \sum_{i=1}^{n} (v^\top x_i)^2.$$

***k*th principal component**

$$v_k = \arg\max_{v} \frac{1}{n} \sum_{i=1}^{n} (v^\top x_i)^2 \quad \text{s.t.} \quad v^\top v = 1, \quad v^\top v_j = 0 \ \ \forall j < k.$$

20.5 Re-writing the Objective Using the Covariance Matrix

Define the (data) covariance matrix

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top \in \mathbb{R}^{d\times d}.$$

Then

$$\frac{1}{n}\sum_{i=1}^{n}(v^\top x_i)^2 = \frac{1}{n}\sum_{i=1}^{n} v^\top x_i x_i^\top v = v^\top \left(\frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top\right) v = v^\top \Sigma v.$$

So the PCA objective becomes

$$v_1 = \arg\max_{v:\, v^\top v=1} v^\top \Sigma v.$$

20.6 Recap: Eigenvalue Decomposition

The covariance matrix Σ is symmetric and positive semidefinite, so it has an eigenvalue decomposition

$$\Sigma = Q\Lambda Q^\top,$$

where Q is orthogonal ($QQ^\top = I$) and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_d)$ with $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$.

20.7 Solving the PCA Optimization (Derivation)

Start from

$$v_1 = \arg\max_{v:\, v^\top v=1} v^\top \Sigma v = \arg\max_{v:\, v^\top v=1} v^\top Q\Lambda Q^\top v.$$

Let $w = Q^\top v$. Since Q is orthogonal, $\|w\| = \|Q^\top v\| = \|v\| = 1$. Then

$$v^\top Q\Lambda Q^\top v = w^\top \Lambda w = \sum_{i=1}^{d} \lambda_i w_i^2.$$

So the simplified optimization is

$$w_1 = \arg\max_{w:\, w^\top w=1} \sum_{i=1}^{d} \lambda_i w_i^2.$$

Because $\lambda_1 \geq \cdots \geq \lambda_d$ and $\sum_i w_i^2 = 1$, the maximum is achieved by placing all mass on the first coordinate:

$$w_1 = e_1, \qquad v_1 = Qw_1 = Qe_1 = q_1,$$

where q_1 is the top eigenvector of Σ.

All components With the additional orthogonality constraints $w^\top w_j = 0$ for $j < k$, the solution is

$$w_k = e_k, \qquad v_k = Qe_k = q_k.$$

20.8 Variance Along a Component and Explained Variance

Variance along the kth principal component

$$\frac{1}{n}\sum_{i=1}^{n}(v_k^\top x_i)^2 = v_k^\top \Sigma v_k = v_k^\top Q\Lambda Q^\top v_k = e_k^\top \Lambda e_k = \lambda_k.$$

How many components to keep? Total variance is

$$\mathrm{TV} = \sum_{k=1}^{d} \lambda_k,$$

and the fraction of explained variance after projecting onto the top u components is

$$\frac{1}{\mathrm{TV}} \sum_{k=1}^{u} \lambda_k.$$

Rule of thumb: keep 80%–90% of the total variance.

20.9 Two Perspectives: Variance vs. Reconstruction Error

Claim Maximizing variance is equivalent to minimizing reconstruction error.

Reconstruction from a 1D projection If $v^\top v = 1$, then the projection of x_i onto v is $(v^\top x_i)v$, and the reconstruction error is

$$\frac{1}{n} \sum_{i=1}^{n} \left\| x_i - (v^\top x_i)v \right\|^2.$$

Expand the squared norm:

$$\left\| x_i - (v^\top x_i)v \right\|^2 = \|x_i\|^2 - 2(v^\top x_i)\, v^\top x_i + \|(v^\top x_i)v\|^2.$$

Since $\|(v^\top x_i)v\|^2 = (v^\top x_i)^2 \|v\|^2 = (v^\top x_i)^2$, we get

$$\left\| x_i - (v^\top x_i)v \right\|^2 = \|x_i\|^2 - (v^\top x_i)^2.$$

Therefore,

$$\arg \min_{v:\, v^\top v=1} \frac{1}{n} \sum_{i=1}^{n} \left\| x_i - (v^\top x_i)v \right\|^2 = \arg \min_{v:\, v^\top v=1} \frac{1}{n} \sum_{i=1}^{n} \|x_i\|^2 - (v^\top x_i)^2 = \arg \max_{v:\, v^\top v=1} \frac{1}{n} \sum_{i=1}^{n} (v^\top x_i)^2.$$

u-dimensional version Let $V_u = [v_1, \ldots, v_u] \in \mathbb{R}^{d \times u}$ with $V_u^\top V_u = I$. Project and reconstruct via

$$z = V_u^\top x, \qquad \hat{x} = V_u z = V_u V_u^\top x.$$

Then PCA equivalently minimizes total reconstruction error

$$\sum_{i=1}^{n} \|x_i - V_u V_u^\top x_i\|^2,$$

and (for centered data) the variance explained by the top u components is $\sum_{k=1}^{u} \lambda_k$.

21 Autoencoders and Diffusion Models

21.1 Autoencoders

Encoder + decoder An **autoencoder** learns a representation of an input $x \in \mathbb{R}^d$ via:

$$z = f_\theta(x) \in \mathbb{R}^k, \qquad \hat{x} = g_{\theta'}(z) = g_{\theta'}(f_\theta(x)).$$

Reconstruction objective With squared error,

$$\min_{\theta,\theta'} \mathbb{E}\left[\|x - g_{\theta'}(f_\theta(x))\|^2\right].$$

The bottleneck dimension k is typically smaller than d.

21.2 PCA Revisited: Reconstruction Objectives

One component For a unit vector $v \in \mathbb{R}^d$ (with $v^\top v = 1$), the 1D reconstruction of x_i is $(v^\top x_i)v$, and the reconstruction objective is

$$\arg\min_{v:\, v^\top v=1} \frac{1}{n}\sum_{i=1}^n \left\|x_i - (v^\top x_i)v\right\|^2.$$

k **components** Let $V = [v_1, \dots, v_k] \in \mathbb{R}^{d\times k}$ with orthonormal columns $V^\top V = I$. The projection is $V^\top x_i \in \mathbb{R}^k$ and the reconstruction is $VV^\top x_i$. The reconstruction objective becomes

$$\arg\min_{V:\, V^\top V=I} \frac{1}{n}\sum_{i=1}^n \left\|x_i - VV^\top x_i\right\|^2.$$

21.3 PCA is a Linear Autoencoder (Derivation)

Linear encoder/decoder Choose linear maps with a bottleneck:

$$\text{encoder: } a = V^\top x \in \mathbb{R}^k, \qquad \text{decoder: } \hat{x} = Va \in \mathbb{R}^d,$$

so $\hat{x} = VV^\top x$ and the reconstruction error for a sample is $L(x) = \|x - \hat{x}\|^2$.

Connection to PCA Assume centered data and let $\Sigma = \frac{1}{n}\sum_{i=1}^n x_i x_i^\top$. Let $P = VV^\top$. Since $V^\top V = I$, P is an orthogonal projection: $P^\top = P$ and $P^2 = P$. Then for each i,

$$\begin{aligned}\|x_i - Px_i\|^2 &= x_i^\top (I-P)^\top (I-P)x_i \\ &= x_i^\top (I-P)x_i \\ &= \|x_i\|^2 - x_i^\top P x_i.\end{aligned}$$

Thus

$$\frac{1}{n}\sum_{i=1}^n \|x_i - VV^\top x_i\|^2 = \frac{1}{n}\sum_{i=1}^n \|x_i\|^2 - \frac{1}{n}\sum_{i=1}^n x_i^\top VV^\top x_i.$$

Using $\operatorname{tr}(AB) = \operatorname{tr}(BA)$,

$$\frac{1}{n}\sum_{i=1}^n x_i^\top VV^\top x_i = \operatorname{tr}\left(V^\top \left(\frac{1}{n}\sum_{i=1}^n x_i x_i^\top\right) V\right) = \operatorname{tr}(V^\top \Sigma V).$$

The first term $\frac{1}{n}\sum_i \|x_i\|^2$ is constant in V, so minimizing reconstruction error is equivalent to maximizing $\operatorname{tr}(V^\top \Sigma V)$. The solution is $V = [q_1, \dots, q_k]$, the top k eigenvectors of Σ (PCA).

21.4 Non-Linear Autoencoders

Replace linear maps with neural networks. An example architecture:

$$f : \mathbb{R}^d \to \mathbb{R}^k, \qquad f(x) = W_2 \, \mathrm{ReLU}(W_1 x),$$

$$g : \mathbb{R}^k \to \mathbb{R}^d, \qquad g(a) = V_2 \, \mathrm{ReLU}(V_1 a),$$

trained with

$$L(x) = \|x - g(f(x))\|^2.$$

Non-linear autoencoders can capture non-linear manifolds; regularization/constraints help avoid simply learning the identity map.

21.5 Sparse Autoencoders

Encourage sparse codes by adding an ℓ_1 penalty on the latent activations $a = f(x)$:

$$L(x) = \|x - g(f(x))\|^2 + \lambda \sum_{j=1}^{k} |a_j|.$$

21.6 Denoising Autoencoders

Corrupt-then-reconstruct Given a corruption process $\tilde{x} \sim q(\tilde{x} \mid x)$, train the model to recover the clean x:

$$\min \; \mathbb{E}\left[\|x - g(f(\tilde{x}))\|^2\right].$$

Intuition: the denoiser "projects" noisy inputs back toward the data manifold.

21.7 Denoising Diffusion

Diffusion models can be viewed as iterated denoising with a denoising autoencoder conditioned on a noise level.

Training (as in the slides) Let $\mathrm{DAE}_w(\tilde{x}, \alpha)$ be a denoising autoencoder (parameters w) that takes a noisy input and a noise level $\alpha \in [0, 1]$.

- Sample clean data $x_0 \sim p(\text{data})$.
- Sample $\alpha \sim \mathrm{Uniform}(0, 1)$ and noise $\varepsilon \sim \mathcal{N}(0, I)$.
- Form noisy data $\tilde{x} = \alpha x_0 + (1 - \alpha)\varepsilon$.
- Update w to minimize
$$\|x_0 - \mathrm{DAE}_w(\tilde{x}, \alpha)\|^2.$$

Generation (as in the slides)

- Initialize $x_T \sim \mathcal{N}(0, I)$.
- For $t = T, \dots, 1$:
  - Set $\alpha_t = (T - t)/T$ and predict $\tilde{x}_0 = \mathrm{DAE}_w(x_t, \alpha_t)$.
  - Sample $z \sim \mathcal{N}(0, I)$ and set $\alpha_{t-1} = (T - (t-1))/T$.
  - Update
    $$x_{t-1} = \alpha_{t-1}\tilde{x}_0 + (1 - \alpha_{t-1})z.$$
- Return x_0.

21.8 Extensions of Diffusion

Diffusion can be adapted for guided image synthesis/editing by starting from a noised version of an input and running the denoising process (high-level idea).

21.9 Latent Diffusion

Motivation Diffusion in pixel space can be expensive: it takes T denoising steps and is especially costly when x is high-dimensional.

Idea Run diffusion in a lower-dimensional latent space:

- Train an autoencoder $x \mapsto z = f(x)$ and $x \approx g(z)$ with z much lower-dimensional than x.
- Train a diffusion model over latent codes z (instead of pixels).
- Sample in latent space and decode back to pixel space with g.

22 Language Modeling and LLMs

22.1 Autoregressive Modeling

Tokenization Represent data as a sequence of discrete tokens

$$x = (x_1, \ldots, x_T), \qquad x_t \in \mathcal{V}.$$

Autoregressive factorization Model a joint distribution by factoring into conditional distributions:

$$p(x) = \prod_{t=1}^{T} p(x_t \mid x_{<t}).$$

Taking logs yields

$$\log p(x) = \sum_{t=1}^{T} \log p(x_t \mid x_{<t}).$$

22.2 The Next-Token Prediction Task

Conditional likelihood modeling We learn a predictor $\hat{p}_\theta(x_t \mid x_{<t})$ from data. Given a dataset of N sequences $\{x^{(i)}\}_{i=1}^N$ (each of length T for simplicity), the average log-likelihood objective is

$$\max_\theta \ \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \log \hat{p}_\theta\left(x_t^{(i)} \mid x_{<t}^{(i)}\right).$$

Equivalently, minimize the negative log-likelihood (cross-entropy loss):

$$\mathcal{L}(\theta) = -\frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} \log \hat{p}_\theta\left(x_t^{(i)} \mid x_{<t}^{(i)}\right).$$

22.3 The Softmax Prediction Head

Embedding + network + softmax To parameterize $\hat{p}_\theta(x_t \mid x_{<t})$ with a neural network:

- Embed tokens with $E : \mathcal{V} \to \mathbb{R}^d$ so that

$$(x_1, \ldots, x_{t-1}) \mapsto (E(x_1), \ldots, E(x_{t-1})) \in \mathbb{R}^{(t-1)\times d}.$$

- Compute logits with a neural network $f : \mathbb{R}^{(t-1)\times d} \to \mathbb{R}^{|\mathcal{V}|}$.

- Normalize with softmax:

$$\hat{p}_\theta(x_t \mid x_{<t}) = \mathrm{softmax}(f(E(x_{<t})))_{x_t}.$$

$$\mathrm{softmax}(u)_k = \frac{e^{u_k}}{\sum_{j=1}^{|\mathcal{V}|} e^{u_j}}.$$

22.4 Transformer Language Models

One network for all positions We do not want a different network for each prefix length. Instead, use a single transformer that maps a length-T embedded sequence to a length-T sequence of hidden states:

$$f : \mathbb{R}^{T\times d} \to \mathbb{R}^{T\times d}.$$

Compose it with an output head $o : \mathbb{R}^d \to \mathbb{R}^{|\mathcal{V}|}$ applied at each position:

$$\text{logits}_t = o(f(E(x))_t) \in \mathbb{R}^{|\mathcal{V}|}.$$

$$\hat{p}_\theta(x_{t+1} \mid x_{\leq t}) = \text{softmax}(\text{logits}_t).$$

Causal masking Use a causal (triangular) attention mask so position t can only attend to positions $\leq t$ (no "looking ahead").

22.5 Shifted Targets (Training on a Whole Sequence)

Given input tokens $(x_1, \ldots, x_T)$, predict a shifted set of labels $(x_2, \ldots, x_{T+1})$. Intuitively, the output at position t predicts the next token.

22.6 Training Data Preprocessing and Sequence Packing

We often have a corpus of documents of varying length, but we want fixed-length sequences for minibatch SGD.

- Concatenate the corpus into one long token stream.
- Insert a special sequence-separator token between documents.
- Cut the stream into equal-length segments of length T.

The segment length T is the **context length**.

22.7 Training and Generation

Training Learn the conditional distributions $\hat{p}_\theta(x_t \mid x_{<t})$ by maximizing conditional likelihood (equivalently minimizing cross-entropy).

Generation Iteratively sample tokens

$$\hat{x}_t \sim \hat{p}_\theta(\cdot \mid \hat{x}_{<t}),$$

either starting from a special separator token (base case) or from a prefix/prompt $(\hat{x}_1, \ldots, \hat{x}_{t_0})$ (warm start).

22.8 Exposure Bias

During training, condition on ground-truth prefixes $x_{<t}$. During generation, condition on model samples $\hat{x}_{<t}$. This mismatch is **exposure bias**; it becomes less problematic as the next-token predictor improves.

22.9 Prompting and Post-Training

Prompting If the next-token predictor is sufficiently good, many tasks can be solved by choosing a prompt so that the most likely continuation is the desired answer.

Post-training Instruction tuning / supervised fine-tuning can make models follow prompts better. Safety tuning further reduces harmful outputs, but must be balanced against over-refusals (overly safe behavior on benign questions).

22.10 Generalized Next-Token Prediction

Autoregressive modeling applies beyond text:

- Convert data (e.g. images) into a sequence of discrete codes (tokens).
- Train the same next-token objective on the code sequence.

Is this practical? (high-level recipe)

- Tokenize an image into a sequence of patches/codes.
- Use a *discrete autoencoder* to map patches to discrete symbols.
- Train an autoregressive model over the resulting symbol sequence.

23 Anticipation and Music Language Models

23.1 Control and Controllable Generation

Modern generative models can be turned into tools by conditioning on user-specified controls (classes, text prompts, melodies, beginnings/endings, etc.).

23.2 Symbolic Music as a Temporal Point Process

Symbolic music can be represented as a sequence of events with stochastic arrival times:

$$e_i = (\mathrm{time}_i, \mathrm{note}_i).$$

This is a **temporal point process**: events arrive over time with content.

23.3 Controllable Generation

We may want to condition on control signals (e.g. melody, beginning, ending) and generate missing portions. The key question is: **how to serialize events and controls** into a sequence for autoregressive modeling.

23.4 Stopping Times and Anticipation

If controls do not appear at *stopping times* (times identifiable when they occur), naive interleavings can prevent the model from planning based on upcoming controls. The lecture introduces an **anticipatory order** to ensure the serialization corresponds to stopping times, enabling the model to anticipate future controls while generating.

23.5 Anticipatory Music Transformer (AMT)

AMT uses GPT-2-style decoder-only Transformers (causal masking) to model serialized symbolic music with controls. Lecture details:

- Model scales: 128M / 360M / 780M parameters (GPT-2 blueprint).
- Training data: Lakh MIDI (176,581 MIDI files; large event/token corpus via event tokenization).
- Compute: training on TPU v3-32 resources.

23.6 Looking Forward

Generative models for music outpace user interfaces; key questions involve giving users meaningful agency in creative workflows rather than full automation.

24 Watermarks for Language Models

24.1 Motivation: Detection vs. Watermarking

Pure detection of machine-generated text becomes impossible as models improve. An alternative is to **watermark** generated text so that it can be detected later.

24.2 Hash-Based Watermarks

Idea (Kirchenbauer et al., ICML 2023):

- At each step, use a hash of the preceding context to seed a pseudo-random partition of the vocabulary into a **red-list** and its complement.
- Modify sampling so the model preferentially chooses tokens not in the red-list.
- Detection: under the null hypothesis (no watermark), about 50% of tokens fall in the red-list; if a document has suspiciously *few* red-list tokens, it is likely watermarked.

This approach creates a tradeoff between **distortion** and **detectability**.

24.3 Distortion-Free Watermarking via a Key

Goal: watermark without changing the distribution of the model outputs (at least on first use). Use a secret **watermark key** (a seeded RNG sequence) to induce correlations between RNG values and sampled tokens.

(One-shot) Distortion-Free Property The watermarked generator behaves like the unwatermarked generator the first time it is run; detection is only possible if the detector knows the key.

24.4 Robust Detection

Practical challenges:

- The detector may not know the original prompt.
- The output may be cropped/edited.

Detection can align the RNG sequence with the observed text and use edit-distance (Levenshtein distance) robustness to substitutions/insertions/deletions.

24.5 Inverse Transform Sampling (ITS)

ITS provides a decoder that maps a probability vector over the vocabulary and a uniform random value $u \sim \text{Uniform}(0,1)$ to a sampled token. This induces detectable correlations between u and the chosen token when the key is reused, enabling hypothesis testing.

24.6 Theory (High-level)

Detectability improves with longer text (via concentration bounds such as Hoeffding) and worsens with longer RNG sequences (via a union bound over possible alignments). Low-entropy sequences are harder to watermark.