# CS 480/680 Winter 2024:

## Lecture Notes

Lecture notes taken, unless otherwise specified, by myself during section 002 of the Winter 2024 offering of CS 480/680, taught by Hongyang Zheng.

## Lectures

# Chapter 1

# Classic Machine Learning

## 1 Introduction

*Lecture 1*
*Jan 9*

There have been three historical AI booms:

1. 1950s–1970s: search-based algorithms (e.g., chess), failed when they realized AI is actually a hard problem
2. 1980s–1990s: expert systems
3. 2012 – present: deep learning

Machine learning is the subset of AI where a program can learn from experience.

Major learning paradigms of machine learning:

- Supervised learning: teacher/human labels answers (e.g., classification, ranking, etc.)
- Unsupervised learning: without labels (e.g., clustering, representation, generation, etc.)
- Reinforcement learning: rewards given for actions (e.g., gaming, pricing, etc.)
- Others: semi-supervised, active learning, etc.

Active focuses in machine learning research:

- Representation: improving the encoding of data into a space
- Generalization: improving the use of the model on new distributions
- Interpretation: understanding how deep learning actually works
- Complexity: improving time/space requirements
- Efficiency: reducing the amount of samples required
- Privacy: respecting legal/ethical concerns of data sourcing
- Robustness: gracefully failing under errors or malicious attack
- Applications

*Lecture 2*
*Jan 11*

A machine learning algorithm has three phases: training, prediction, and evaluation.

**Definition 1.1** (dataset)

A dataset consists of a list of features $\mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{x}'_1, \dots, \mathbf{x}'_m \in \mathbb{R}^d$ which are $d$-dimensional vectors and a label vector $\mathbf{y}^\top \in \mathbb{R}^n$.

Each training sample $\mathbf{x}_i$ is associated with a label $y_i$. A test sample $\mathbf{x}'_i$ may or may not be labelled.

**Example 1.2** (email filtering)**.** Suppose we have a list $D$ of $d$ English words.

Define the training set $X = [\mathbf{x}_1, \dots, \mathbf{x}_n] \in \mathbb{R}^{d \times n}$ and $\mathbf{y} = [y_1, \dots, y_n] \in \{\pm 1\}^n$ such that $\mathbf{x}_{ij} = 1$ if the word $j \in D$ appears in email $i$ (this is the bag-of-words representation):

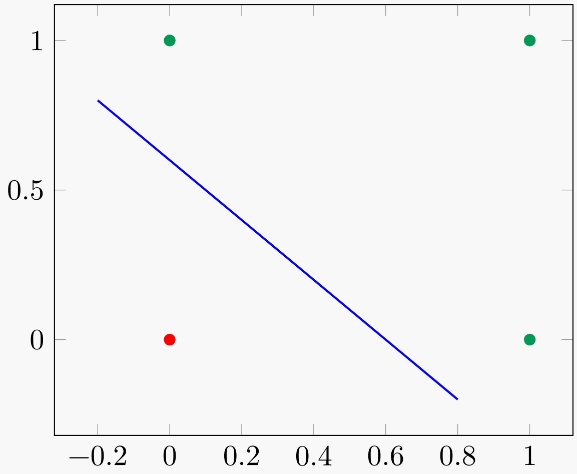
| | $\mathbf{x}_1$ | $\mathbf{x}_2$ | $\mathbf{x}_3$ | $\mathbf{x}_4$ | $\mathbf{x}_5$ | $\mathbf{x}_6$ | $\mathbf{x}'$ |
|---|---|---|---|---|---|---|---|
| and | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| viagra | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| the | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| of | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| nigeria | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| $y$ | + | − | + | − | + | − | ? |

Then, given a new email $\mathbf{x}'_1$, we must determine if it is spam or not.

**Example 1.3** (OR dataset)**.** We want to train the OR function:

| | $\mathbf{x}_1$ | $\mathbf{x}_2$ | $\mathbf{x}_3$ | $\mathbf{x}_4$ |
|---|---|---|---|---|
| | 0 | 1 | 0 | 1 |
| | 0 | 0 | 1 | 1 |
| $y$ | − | + | + | + |

This can be represented graphically by finding a line dividing the points:



# 2 Perceptron

**Definition 2.1**

The inner product of vectors $\langle \mathbf{a}, \mathbf{b} \rangle$ is the sum of the element-wise product $\sum_j a_j b_j$.

A linear function is a function $f : \mathbb{R}^d \to \mathbb{R}^d$ such that for all $\alpha, \beta \in \mathbb{R}$, $\mathbf{x}, \mathbf{z} \in \mathbb{R}^d$, $f(\alpha\mathbf{x}+\beta\mathbf{z}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{z})$.

**Theorem 2.2** (linear duality)

A function is linear if and only if there exists $\mathbf{w} \in \mathbb{R}^d$ such that $f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{w} \rangle$.

*Proof.* ($\Rightarrow$) Suppose $f$ is linear. Let $\mathbf{w} := [f(\mathbf{e}_1), \dots, f(\mathbf{e}_d)]$ where $\mathbf{e}_i$ are coordinate vectors. Then:

$$\begin{aligned} f(\mathbf{x}) &= f(x_1\mathbf{e}_1 + \cdots + x_d\mathbf{e}_d) \\ &= x_1 f(\mathbf{e}_1) + \cdots + x_d f(\mathbf{e}_d) \\ &= \langle \mathbf{x}, \mathbf{w} \rangle \end{aligned}$$

by linearity of $f$.

($\Leftarrow$) Suppose there exists $\mathbf{w}$ such that $f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{w} \rangle$. Then:

$$\begin{aligned} f(\alpha\mathbf{x} + \beta\mathbf{z}) &= \langle \alpha\mathbf{x} + \beta\mathbf{z}, \mathbf{w}, \alpha\mathbf{x} + \beta\mathbf{z}, \mathbf{w} \rangle \\ &= \alpha \langle \mathbf{x}, \mathbf{w} \rangle + \beta \langle \mathbf{x}, \mathbf{w} \rangle \\ &= \alpha f(\mathbf{x}) + \beta f(\mathbf{z}) \end{aligned}$$

since inner products are linear in the first argument. □

**Definition 2.3** (affine function)

A function $f(\mathbf{x})$ where there exist $\mathbf{w} \in \mathbb{R}^d$ and bias $b \in \mathbb{R}$ such that $f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{w} \rangle + b$.

**Definition 2.4** (sign function)

$$\operatorname{sgn}(t) = \begin{cases} +1 & t > 0 \\ -1 & t \leq 0 \end{cases}$$

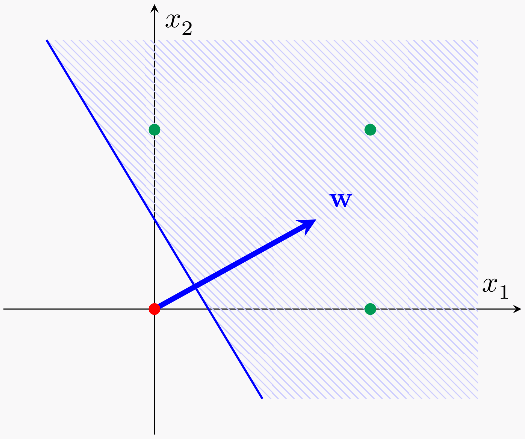
It does not matter what $\operatorname{sgn}(0)$ is defined as.

**Definition 2.5** (linear classifier)

$\hat{y} = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{w} \rangle + b)$

The parameters $\mathbf{w}$ and $b$ will uniquely determine the linear classifier.

**Example 2.6** (geometric interpretation)**.** We can interpret $\hat{y} > 0$ as a halfspace (see CO 250). Then, we can draw something like:



**Proposition 2.7**

The vector $\mathbf{w}$ is orthogonal to the decision boundary $H$.

*Proof.* Let $\mathbf{x}, \mathbf{x}' \in H$ be vectors on the boundary $H = \{x : \langle \mathbf{w}, \mathbf{x} \rangle + b = 0\}$. Then, we must show $\mathbf{x}' - \mathbf{x} = \overrightarrow{\mathbf{x}\mathbf{x}'} \perp \mathbf{w}$.

We can calculate $\langle \mathbf{w}, \mathbf{x}' - \mathbf{x} \rangle = \langle \mathbf{w}, \mathbf{x} \rangle - \langle \mathbf{w}, \mathbf{x}' \rangle = -b - (-b) = 0$. □

Originally, the inventor of the perceptron thought it could do anything. He was (obviously) wrong.

**Algorithm 1** Training Perceptron

```
Require: Dataset $(\mathbf{x}_i, \mathsf{y}_i) \in \mathbb{R}^d \times \{\pm 1\}$, initialization $\mathbf{w}_0 \in \mathbb{R}^d$, $b_0 \in \mathbb{R}$.
Ensure: $\mathbf{w}$ and $b$ for linear classifier $\text{sgn}(\langle \mathbf{x}, \mathbf{w} \rangle + b)$
  for $t = 1, 2, \ldots$ do
      receive index $I_t \in \{1, \ldots, n\}$
      if $\mathsf{y}_{I_t}\left(\left\langle \mathbf{x}_{I_t}, \mathbf{w} \right\rangle + b\right) \leq 0$ then
          $\mathbf{w} \leftarrow \mathbf{w} + \mathsf{y}_{I_t}\mathbf{x}_{I_t}$
          $b \leftarrow b + \mathsf{y}_{I_t}$
```

In a perceptron, we train by adjusting $\mathbf{w}$ and $b$ whenever a training data feature is classified "wrong" (i.e., $\mathsf{score}_{\mathbf{w},b}(\mathbf{x}) := \mathsf{y}\hat{y} < 0 \iff$ the signs disagree).

The perceptron solves the feasibility problem

$$\text{Find } \mathbf{w} \in \mathbb{R}^d,\ b \in \mathbb{R} \text{ such that } \forall i, \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) > 0$$

by iterating one-by-one. It will converge "faster" (with fewer $t$-iterations) if the data is "easy".

Consider what happens when there is a "wrong" classification. Let $\mathbf{w}_{k+1} = w_k + \mathsf{y}\mathbf{x}$ and $b_{k+1} = b_k + \mathsf{y}$.

Then, the updated score is:

$$\begin{aligned}
\mathsf{score}_{\mathbf{w}_{k+1},b_{k+1}}(\mathbf{x}) &= \mathsf{y}\cdot(\langle \mathbf{x},\mathbf{w}_{k+1}\rangle + b_{k+1}) \\
&= \mathsf{y}\cdot(\langle \mathbf{x},\mathbf{w}_{k}+\mathsf{y}\mathbf{x}\rangle + b_{k}+\mathsf{y}) \\
&= \mathsf{y}\cdot(\langle \mathbf{x},\mathbf{w}_{k}\rangle + b_{k}) + \langle \mathbf{x},\mathbf{x}\rangle + 1 \\
&= \mathsf{y}\cdot(\langle \mathbf{x},\mathbf{w}_{k}\rangle + b_{k}) + \underbrace{\|\mathbf{x}\|_2^2 + 1}_{\text{always positive}}
\end{aligned}$$

which is always an increase over the previous "wrong" score.

↓ *Lectures 3 and 4 taken slides and Neysa since I was sick* ↓

*Lecture 3*
*Jan 16*

Instead of writing the affine function $\langle \mathbf{x},\mathbf{w}\rangle + b$, write $\langle \mathbf{x},\mathbf{w}\rangle = \left\langle \begin{pmatrix}\mathbf{x}\\1\end{pmatrix}, \begin{pmatrix}\mathbf{w}\\b\end{pmatrix}\right\rangle$.

Then, the update rule becomes $\mathbf{w} \leftarrow \mathbf{w} + \mathsf{y}\mathbf{x}$.

**Theorem 2.8** (convergence theorem)

Suppose there exists $\mathbf{w}^*$ such that $\mathsf{y}_i \langle \mathbf{x}_i, \mathbf{w}^*, \mathbf{x}_i, \mathbf{w}^*\rangle > 0$ for all $i$. Assume that $\|\mathbf{x}_i\|_2 \leq C$ for all $i$, and we normalize the $\mathbf{w}^*$ such that $\|\mathbf{w}^*\|_2 = 1$. Define the margin $\gamma := \min_i |\langle \mathbf{x}_i, \mathbf{w}^*\rangle|$.

Then, the perceptron algorithm converges after $C^2/\gamma^2$ mistakes.

*Proof.* Recall the update on the mistake $(\mathbf{x},\mathsf{y})$ is $\mathbf{w} \leftarrow \mathbf{w} + \mathsf{y}\mathbf{x}$.

Then, the inner product $\langle \mathbf{w},\mathbf{w}^*\rangle$ is

$$\begin{aligned}
\langle \mathbf{w}+\mathsf{y}\mathbf{x},\mathbf{w}^*\rangle &= \langle \mathbf{w},\mathbf{w}^*\rangle + \mathsf{y}\langle \mathbf{x},\mathbf{w}^*\rangle \\
&= \langle \mathbf{w},\mathbf{w}^*\rangle + |\langle \mathbf{x},\mathbf{w}^*\rangle| \\
&\geq \langle \mathbf{w},\mathbf{w}^*\rangle + \gamma
\end{aligned}$$

because $\mathsf{y}\langle \mathbf{x},\mathbf{w}^*\rangle$ must be positive if $\mathbf{w}^*$ is optimal. So for each update, $\langle \mathbf{w},\mathbf{w}^*\rangle$ grows by at least $\gamma > 0$. That is, after $M$ updates, $\langle \mathbf{w},\mathbf{w}^*\rangle \geq M\gamma$.

Likewise, the inner product $\langle \mathbf{w},\mathbf{w}\rangle$ is

$$\begin{aligned}
\langle \mathbf{w}+\mathsf{y}\mathbf{x},\mathbf{w}+\mathsf{y}\mathbf{x}\rangle &= \langle \mathbf{w},\mathbf{w}\rangle + \underbrace{2\mathsf{y}\langle \mathbf{w},\mathbf{x}\rangle}_{<0\text{ because an update means it's wrong}} + \overbrace{\mathsf{y}^2\langle \mathbf{w},\mathbf{w}\rangle}^{\in[0,C^2]\text{ by construction}} \\
&\leq \langle \mathbf{w},\mathbf{w}\rangle + C^2
\end{aligned}$$

so each update grows $\langle \mathbf{w},\mathbf{w}\rangle$ by at most $C^2$, meaning that after $M$ updates, $\langle \mathbf{w},\mathbf{w}\rangle \leq MC^2$.

Finally, recall from linear algebra that $1 \geq \cos(\mathbf{w},\mathbf{w}^*) = \frac{\langle \mathbf{w},\mathbf{w}^*\rangle}{\|\mathbf{w}\|_2\|\mathbf{w}^*\|_2}$. Then,

$$\begin{aligned}
1 &\geq \frac{\langle \mathbf{w},\mathbf{w}^*\rangle}{\|\mathbf{w}\|_2\cdot\|\mathbf{w}^*\|_2} \\
&\geq \frac{M\gamma}{\sqrt{MC^2}\cdot 1} \\
&= \sqrt{M}\frac{\gamma}{C}
\end{aligned}$$

which implies $M \leq C^2/\gamma^2$. □

Therefore, the larger the margin $\gamma$ is, the more linearly separable the data is, and the faster the perceptron algorithm will converge.

**Optimization perspective** We can equivalently characterize the perceptron algorithm as an optimization problem. Given the linear classifier $\hat{y} = \operatorname{sgn}(\langle \mathbf{w}, \mathbf{x} \rangle)$, we want to minimize the perceptron loss

$$\begin{aligned} \ell(\mathbf{w}, \mathbf{x}_t, \mathsf{y}_t) &= -\mathsf{y}_t \langle \mathbf{w}, \mathbf{x}_t \rangle \cdot \mathbb{I}[\text{mistake on } \mathbf{x}_t] \\ &= -\min\{\mathsf{y}_t \langle \mathbf{w}, \mathbf{x}_t \rangle, 0\} \\ L(\mathbf{w}) &= -\frac{1}{n} \sum_{t=1}^{n} (\mathsf{y}_t \langle \mathbf{w}, \mathbf{x}_t \rangle \cdot \mathbb{I}[\text{mistake on } \mathbf{x}_t]) \end{aligned}$$

Then, the gradient descent update (see section 8) is

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}} \ell(\mathbf{w}_t, \mathbf{x}_t, \mathsf{y}_t) \\ &= \mathbf{w}_t + \eta_t \mathsf{y}_t \mathbf{x}_t \cdot \mathbb{I}[\text{mistake on } \mathbf{x}_t] \end{aligned}$$

With step size $\eta_t = 1$, we recover the update rule $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathsf{y}_t \mathbf{x}_t$.

**Remark 2.9.** The solution to perceptron is not unique, since there are many possible lines separating the data.

To pick the "best" line, we can maximize the margin $\gamma$. This leads to support vector machines (see sections 5 and 6).

**Example 2.10** (XOR dataset)**.** Consider the XOR function

| | $\mathbf{x}_1$ | $\mathbf{x}_2$ | $\mathbf{x}_3$ | $\mathbf{x}_4$ |
|---|---|---|---|---|
| | 0 | 1 | 0 | 1 |
| | 0 | 0 | 1 | 1 |
| $\mathsf{y}$ | $-$ | $+$ | $+$ | $+$ |

There is no separating hyperplane.

*Proof.* Suppose there exist $\mathbf{w}$ and $b$ such that $\mathsf{y}(\langle \mathbf{x}, \mathbf{w} \rangle + b) > 0$. Then,

$$\begin{aligned} x_1 &= (0,0), \mathsf{y}_1 = - \implies b < 0 \\ x_2 &= (1,0), \mathsf{y}_2 = + \implies w_1 + b > 0 \\ x_3 &= (0,1), \mathsf{y}_3 = + \implies w_1 + b > 0 \implies w_1 + w_2 + 2b > 0 \\ x_4 &= (1,1), \mathsf{y}_4 = - \implies w_1 + w_2 + b < 0 \implies b > 0 \end{aligned}$$

which is a contradiction. □

This leads us to a theorem.

**Theorem 2.11**

If there is no perfect separating hyperplane, then the perceptron algorithm cycles.

The proof is really complicated, and we will not cover it.

In this case, we can allow some wrong answers by setting a reasonable loss $\ell$ and regularizer reg:

$$\min_{\mathbf{w}} \hat{\mathbb{E}}[\ell(\mathsf{y}\hat{y}) + \mathrm{reg}(\mathbf{w})] \quad \text{s.t.} \quad \hat{y} := \langle \mathbf{x}, \mathbf{w} \rangle + b$$

We stop running perceptron when either:

- the maximum number of iterations is reached (i.e., we keep a constant maxiter),
- the maximum allowed runtime is reached,
- the training error stops changing, or
- the validation error stops decreasing.

If we have multiple classes ($c$ of them), we can run perceptron as either one-vs.-all or one-vs.-one.

In one-vs.-all perceptron, for each class $k$, let it be positive, and all others be negative. We train weights $\mathbf{w}_k$ to get $c$ imbalanced perceptrons. Then, predict according to the highest score

$$\hat{\mathsf{y}} := \arg\max_k \langle \mathbf{x}, \mathbf{w}_k \rangle .$$

In one-vs.-one perceptron, for each pair of classes $(k, l)$, let $k$ be positive, $l$ be negative, and ignore all other classes. Then, train weights $\mathbf{w}_{k,l}$ for a total of $\binom{c}{2}$ balanced perceptrons. We predict by majority vote

$$\hat{\mathsf{y}} := \arg\max_k \sum_{l:l\neq k} \langle \mathbf{x}, \mathbf{w}_{k,l} \rangle .$$

## 3 Linear Regression

*Lecture 4*
*Jan 18*

**Problem 3.1** (regression)

Given training data $(\mathbf{x}_i, \mathsf{y}_i) \in \mathbb{R}^{d+t}$, find $f : \mathcal{X} \to \mathcal{Y}$ such that $f(\mathbf{x}_i) \approx \mathsf{y}_i$.

The problem is that for finite training data, there are an infinite number of functions that exactly hit each point.

**Theorem 3.2** (exact interpolation is always possible)

For any finite training data $(\mathbf{x}_i, \mathsf{y}_i) : i = 1, \dots, n$ such that $\mathbf{x}_i \neq \mathbf{x}_j$ for all $i \neq j$, there exist infinitely many functions $f : \mathbb{R}^d \to \mathbb{R}^t$ such that for all $i$, $f(\mathbf{x}_i) = \mathsf{y}_i$.

TODO: ...up to slide 14 (geometry of linear regression)

↑ *Lectures 3 and 4 taken from slides and Neysa since I was sick* ↑

*Lecture 5*
*Jan 23*

**Theorem 3.3** (Fermat's necessary condition for optimality)

If $\mathbf{w}$ is a minimizer/maximizer of a differentiable function $f$ over an open set, then $f'(\mathbf{w}) = \mathbf{0}$.

We can use this property to solve linear regression.

Recall the loss is $\text{Loss}(\mathbf{W}) = \frac{1}{n}\|\mathbf{W}\mathbf{X} - \mathsf{Y}\|_F^2$. Then, the derivative $\nabla_{\mathbf{W}}\,\text{Loss}(\mathbf{W}) = \frac{2}{n}(\mathbf{W}\mathbf{X} - \mathsf{Y})\mathbf{X}^\top$.

We can derive the normal equation:

$$\begin{aligned}
\frac{2}{n}(\mathbf{W}\mathbf{X} - \mathsf{Y})\mathbf{X}^\top &= 0 \\
\mathbf{W}\mathbf{X}\mathbf{X}^\top - \mathsf{Y}\mathbf{X}^\top &= 0 \\
\boxed{\mathbf{W}\mathbf{X}\mathbf{X}^\top = \mathsf{Y}\mathbf{X}^\top} \\
\mathbf{W} &= \mathsf{Y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}
\end{aligned}$$

Once we find $\mathbf{W}$, we can predict on unseen data $\mathbf{X}_{test}$ with $\hat{\mathsf{Y}}_{test} = \mathbf{W}\mathbf{X}_{test}$.

Then,

Suppose $\mathbf{X} = \begin{bmatrix} 0 & \epsilon \\ 1 & 1 \end{bmatrix}$ and $\mathsf{y} = \begin{bmatrix} 1 & -1 \end{bmatrix}$.

Then, solving the linear least squares regression we get $\mathbf{w} = \mathsf{y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1} = \begin{bmatrix} -2/\epsilon & 1 \end{bmatrix}$. This is chaotic!

Why does this happen? As $\epsilon \to 0$, two columns in $\mathbf{X}$ become almost linearly dependent with incongruent corresponding $y$-values. This leads to a contradiction and an unstable $\mathbf{w}$.

To solve this, we add a $\lambda\|\mathbf{W}\|_F^2$ term.

**Definition 3.4** (ridge regression)

Take the linear regression and add a regularization term:

$$\min_{\mathbf{W}} \frac{1}{n}\|\mathbf{W}\mathbf{X} - \mathsf{Y}\|_F^2 + \lambda\|\mathbf{W}\|_F^2$$

This gives a new normal equation:

$$\begin{aligned}
\text{Loss}(\mathbf{W}) &= \frac{1}{n}\|\mathbf{W}\mathbf{X} - \mathsf{Y}\|_F^2 + \lambda\|\mathbf{W}\|_F^2 \\
\nabla_{\mathbf{W}}\,\text{Loss}(\mathbf{W}) &= \frac{2}{n}(\mathbf{W}\mathbf{X} - \mathsf{Y})\mathbf{X}^\top + 2\lambda\mathbf{W} \\
0 &= \frac{2}{n}(\mathbf{W}\mathbf{X} - \mathsf{Y})\mathbf{X}^\top + 2\lambda\mathbf{W} \\
\boxed{\mathbf{W}(\mathbf{X}\mathbf{X}^\top + n\lambda I) = \mathsf{Y}\mathbf{X}^\top} \\
\mathbf{W} &= \mathsf{Y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + n\lambda I)^{-1}
\end{aligned}$$

**Proposition 3.5**

$\mathbf{X}\mathbf{X}^\top + n\lambda I$ is far from rank-deficient for large $\lambda$.

*Proof.* Recall from linear algebra that we can always take the singular value decomposition of any matrix $M = U\Sigma V^\top$ where $U$ and $V$ are orthogonal and $\Sigma$ is non-negative diagonal where the rank is the number of non-zero entries in $\Sigma$.

Consider the SVD of $\mathbf{X}$:

$$\begin{aligned}\mathbf{X} &= U\Sigma V^\top \\ \mathbf{X}\mathbf{X}^\top &= U\Sigma V^\top V \Sigma^\top U^\top = U\Sigma^2 U^\top \\ \mathbf{X}\mathbf{X}^\top + n\lambda I &= U\Sigma^2 U^\top + U(n\lambda I)U^\top \\ &= U(\Sigma^2 + n\lambda I)U^\top\end{aligned}$$

The matrix $\Sigma^2 + n\lambda I$ is a diagonal matrix with strictly positive elements for sufficiently large $\lambda$. Therefore, $\mathbf{X}\mathbf{X}^\top + n\lambda I$ has full rank and thus no singular values. □

**Remark 3.6.** Performing a ridge regularization is identical to augmenting the data.

Notice that

$$\frac{1}{n}\|\mathbf{W}\mathbf{X} - \mathsf{Y}\|_F^2 + \lambda\|\mathbf{W}\|_F^2 = \frac{1}{n}\left\|\mathbf{W}\begin{bmatrix}\mathbf{X} & \sqrt{n\lambda}I\end{bmatrix} - \begin{bmatrix}\mathsf{Y} & \mathbf{0}\end{bmatrix}\right\|_F^2$$

so if we augment $\mathbf{X}$ with $\sqrt{n\lambda}I$ and $\mathsf{Y}$ with $\mathbf{0}$, i.e., $p$ data points $\mathbf{x}_j = \sqrt{n\lambda}\mathbf{e}_j$ and $\mathsf{y}_j = 0$.

# 4 Logistic Regression

Return to the linear classification problem.

Recall that we took $\hat{\mathsf{y}} = \operatorname{sgn}(\langle \mathbf{x}, \mathbf{w}\rangle)$ where $\mathbf{x} = \begin{pmatrix}\mathbf{x} \\ 1\end{pmatrix}$ and $\mathbf{w} = \begin{pmatrix}\mathbf{w} \\ b\end{pmatrix}$ in $\mathbb{R}^{d+1}$.

How confident are we in our prediction $\hat{\mathsf{y}}$? We can use the margin (or logit) $|\langle \mathbf{x}, \mathbf{w}\rangle|$ ("how far away is the point from the decision boundary?").

The margin is unnormalized with respect to the data, so we cannot really interpret it until we somehow cram it into $[0, 1]$.

We can try directly learning hte confidence.

Let $\mathcal{Y} = \{0, 1\}$. Consider confidence $p(\mathbf{x}; \mathbf{w}) := \Pr[\mathsf{Y} = 1 \mid \mathsf{X} = \mathbf{x}]$. Given independent $(\mathbf{x}_i, \mathsf{y}_i)$:

$$\begin{aligned}&\Pr[\mathsf{Y}_1 = \mathsf{y}_1, \dots, \mathsf{Y}_n = \mathsf{y}_n \mid \mathsf{X}_1 = \mathbf{x}_1, \dots, \mathsf{X}_n = \mathbf{x}_n] \\ =&\prod_{i=1}^n \Pr[\mathsf{Y}_i = \mathsf{y}_i \mid \mathsf{X}_i = \mathbf{x}_i] \\ =&\prod_{i=1}^n [p(\mathbf{x}_i; \mathbf{w})]^{\mathsf{y}_i}[1 - p(\mathbf{x}_i; \mathbf{w})]^{1-\mathsf{y}_i}\end{aligned}$$

and we can get our maximum likelihood estimation

**Definition 4.1** (maximum likelihood estimation)

$$\max_{\mathbf{w}} \prod_{i=1}^{n} [p(\mathbf{x}_i; \mathbf{w})]^{\mathsf{y}_i} [1 - p(\mathbf{x}_i; \mathbf{w})]^{1-\mathsf{y}_i}$$

or equivalently the minimum minus log-likelihood

$$\min_{\mathbf{w}} \sum_{i=1}^{n} [-\mathsf{y}_i \log p(\mathbf{x}_i; \mathbf{w}) - (1 - \mathsf{y}_i) \log(1 - p(\mathbf{x}_i; \mathbf{w}))]$$

Now, how do we define the probability $p$ based on $\mathbf{w}$?

We will assume that the log of the odds ratio $\log \frac{\text{probability of event}}{\text{probability of no event}} = \log \frac{p(\mathbf{x};\mathbf{w})}{1-p(\mathbf{x};\mathbf{w})} = \langle \mathbf{x}, \mathbf{w} \rangle$ is linear.

This leads us to the sigmoid transformation.

**Definition 4.2** (sigmoid transformation)

$$p(\mathbf{x}; \mathbf{w}) = \frac{1}{1 + \exp(-\langle \mathbf{x}, \mathbf{w} \rangle)}$$

If we return now to the MLE we defined earlier, we get

$$\begin{aligned}
&\min_{\mathbf{w}} \sum_{i=1}^{n} [-\mathsf{y}_i \log p(\mathbf{x}_i; \mathbf{w}) - (1 - \mathsf{y}_i) \log(1 - p(\mathbf{x}_i; \mathbf{w}))] \\
&= \min_{\mathbf{w}} \sum_{i=1}^{n} \left[ -\mathsf{y}_i \log \frac{1}{1 + \exp(-\langle \mathbf{x}, \mathbf{w} \rangle)} - (1 - \mathsf{y}_i) \frac{\exp\{-\langle \mathbf{x}, \mathbf{w} \rangle\}}{1 + \exp(-\langle \mathbf{x}, \mathbf{w} \rangle)} \right] \\
&= \min_{\mathbf{w}} \sum_{i=1}^{n} [\mathsf{y}_i \log(1 + \exp(-\langle \mathbf{x}, \mathbf{w} \rangle)) + (1 - \mathsf{y}_i) \log(1 + \exp(-\langle \mathbf{x}, \mathbf{w} \rangle)) + (1 - \mathsf{y}_i) \langle \mathbf{x}, \mathbf{w} \rangle] \\
&= \min_{\mathbf{w}} \sum_{i=1}^{n} \log[1 + \exp(-\langle \mathbf{x}_i, \mathbf{w} \rangle)] + (1 - \mathsf{y}_i)(\langle \mathbf{x}_i, \mathbf{w} \rangle)
\end{aligned}$$

If we redefine $\mathsf{y}'_i = \frac{\mathsf{y}_i + 1}{2}$, i.e., $\mathsf{y}' \in \{\pm 1\}$, then we get the logistic loss

$$\min_{\mathbf{w}} \sum_{i=1}^{n} \log[1 + \exp(-\mathsf{y}'_i \langle \mathbf{x}, \mathbf{w} \rangle)] \tag{4.a}$$

There is no closed form solution for this problem, so we use the gradient descent algorithm (covered in section 8).

Suppose we have found an optimal $\mathbf{w}$. Then, we can set $\hat{y} = 1 \iff p(\mathbf{x}; \mathbf{w}) = \Pr[\mathsf{Y} = 1 \mid \mathsf{X} = \mathbf{x}] > \frac{1}{2}$. The value of $p(\mathbf{x}; \mathbf{w})$ is our confidence.

Remember: All this is under the assumption that the log of the odds ratio is linear. Everything is meaningless if it is not.

**Extending to the multiclass case** Suppose we instead have $\mathsf{y} \in \{1, \dots, c\}$ and we need to learn $\mathbf{w}_i$ for each class. The sigmoid function becomes the softmax function

$$\Pr[\mathsf{Y} = k \mid \mathsf{X} = \mathbf{x}; \mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_c]] = \frac{\exp\langle \mathbf{x}, \mathbf{w}_k \rangle}{\sum_{l=1}^{c} \exp\langle \mathbf{x}, \mathbf{w}_l \rangle} \tag{4.b}$$

This maps the real-valued vector $\mathbf{x}$ to a probability vector. Notice that the softmax values for each class are all non-negative and sum to 1.

To train, we use the MLE again

To predict, pick the index of the highest softmax value

$$\hat{\mathsf{y}} = \arg\max_k \Pr[\mathsf{Y} = k \mid \mathsf{X} = \mathbf{x}; \mathbf{W} = [\mathbf{w}_1, \dots, \mathbf{w}_c]]$$

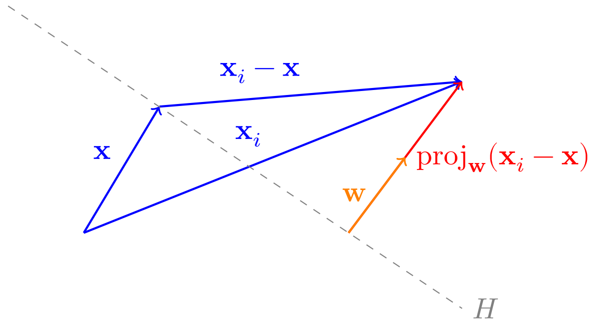
# 5 Hard-Margin Support Vector Machines

*Lecture 6*
*Jan 25*

Recall that the perceptron is a feasibility program, i.e., a linear program with $\mathbf{c}^\top \mathbf{x} = \mathbf{0}$. It has infinite solutions.

Naturally, some are much better than others. To take advantage of better algorithms, we can instead maximize the separation.

Let $H$ be a the hyperplane defined by $\langle \mathbf{x}, \mathbf{w} \rangle + b = 0$. The separation (distance) between a point $\mathbf{x}_i$ and $H$ is the length of the projection of $\mathbf{x}_i - \mathbf{x}$ onto the normal vector $\mathbf{w}$.


$\mathbf{x}_i - \mathbf{x}$
$\mathbf{x}$
$\mathbf{x}_i$
$\mathrm{proj}_{\mathbf{w}}(\mathbf{x}_i - \mathbf{x})$
$\mathbf{w}$
$H$

Simplfiying, we can express this as

$$\begin{aligned} \frac{\langle \mathbf{x}_i - \mathbf{x}, \mathbf{w} \rangle}{\|\mathbf{w}\|_2} &= \frac{\langle \mathbf{x}_i, \mathbf{w} \rangle - \langle \mathbf{x}, \mathbf{w} \rangle}{\|\mathbf{w}\|_2} && \text{(linearity)} \\ &= \frac{\langle \mathbf{x}_i, \mathbf{w} \rangle + b}{\|\mathbf{w}\|_2} && (\mathbf{x} \in H \Leftrightarrow \langle \mathbf{x}, \mathbf{w} \rangle + b = 0) \\ &= \frac{\mathsf{y}_i \hat{y}_i}{\|\mathbf{w}\|_2} \end{aligned}$$

We now have something to maximize.

**Definition 5.1** (margin)

Given a hyperplane $H := \{\mathbf{x} : \langle \mathbf{x}, \mathbf{w} \rangle + b = 0\}$ separating the data, the margin is the smallest distance between a data point $\mathbf{x}_i$ and $H$.

That is, $\min_i \frac{\mathsf{y}_i \hat{y}_i}{\|\mathbf{w}\|_2}$.

The goal is the maximize the margin across all possible hyperplanes:

$$\max_{\mathbf{w},b} \min_i \frac{\mathsf{y}_i \hat{y}_i}{\|\mathbf{w}\|_2} \quad \text{s.t.} \quad \forall i, \mathsf{y}_i \hat{y}_i > 0 \quad \text{where} \quad \hat{y}_i := \langle \mathbf{x}_i, \mathbf{w} \rangle + b$$

We claim that we can arbitrarily scale the numerator. Let $c > 0$. Then, $(\mathbf{w}, b)$ has the same loss as $(c\mathbf{w}, cb)$ because $\frac{\langle \mathbf{x}_i, c\mathbf{w} \rangle + cb}{\|c\mathbf{w}\|_2} = \frac{c\langle \mathbf{x}_i, \mathbf{w} \rangle + cb}{c\|\mathbf{w}\|_2} = \frac{\langle \mathbf{x}_i, \mathbf{w} \rangle + b}{\|\mathbf{w}\|_2}$.

Therefore, we can equivalently write

$$\max_{\mathbf{w},b} \frac{1}{\|\mathbf{w}\|_2} \quad \text{s.t.} \quad \min_i \mathsf{y}_i \hat{y}_i = 1 \quad \text{where} \quad \hat{y}_i := \langle \mathbf{x}_i, \mathbf{w} \rangle + b$$

or even better:

$$\min_{\mathbf{w},b} \|\mathbf{w}\|_2^2 \quad \text{s.t.} \quad \forall i, \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1 \tag{5.a}$$

Finally, consider the points that are closest to the boundary.

**Definition 5.2**

For the separating hyperplane $H = \{\langle \mathbf{x}_i, \mathbf{w} \rangle + b = 0\}$, the two supporting hyperplanes are the parallel hyperplanes $H_+ := \{\langle \mathbf{x}_i, \mathbf{w} \rangle + b = 1\}$ and $H_- := \{\langle \mathbf{x}_i, \mathbf{w} \rangle + b = -1\}$ which represent the margin boundaries.

A support vector is a data point $\mathbf{x}_i \in H_+ \cup H_-$.

The support vectors are rare, but decisive because they reach the boundary of the constraint.

**Explanation from the dual perspective** Recall the SVM quadratic program

$$\min_{\mathbf{w}_b} \frac{1}{2}\|\mathbf{w}\|_2^2 \quad \text{s.t.} \quad \forall i, \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1$$

Introduce Lagrangian multipliers (dual variables) $\boldsymbol{\alpha} \in \mathbb{R}^n$.

$$\begin{aligned}
&\min_{\mathbf{w},b} \max_{\boldsymbol{\alpha} > \mathbf{0}} \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_i \alpha_i[\mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) - 1] \\
&= \min_{\mathbf{w},b} \begin{cases} +\infty & \exists i, \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) < 1 (\text{set } \alpha_i \text{ as } +\infty) \\ \frac{1}{2}\|\mathbf{w}\|_2^2 & \forall i, \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1 (\text{set all } \alpha_i \text{ as } 0) \end{cases} \\
&= \min_{\mathbf{w}_b} \frac{1}{2}\|\mathbf{w}\|_2^2, \quad s.t. \forall i, \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w} \rangle + b) \geq 1
\end{aligned}$$

Therefore, we only need to study the minimax problem. Assuming that the problem is convex (which it is, outside the scope of the course), we can express this as

$$\max_{\boldsymbol{\alpha}>\mathbf{0}} \overbrace{\min_{\mathbf{w},b} \underbrace{\frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_i \alpha_i[\mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b) - 1]}_{\text{Loss}(\mathbf{w},b,\alpha)}}^{\text{Loss}(\alpha)}$$

and take the derivative of the interior with respect to $\mathbf{w}$ and $b$:

$$\begin{aligned}
\frac{\partial\,\text{Loss}(\mathbf{w},b,\alpha)}{\partial \mathbf{w}} &= \mathbf{w} - \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i = 0 \\
\mathbf{w}^* &= \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i \\
\frac{\partial\,\text{Loss}(\mathbf{w},b,\alpha)}{\partial b} &= -\sum_i \alpha_i \mathsf{y}_i = 0 \\
\sum_i \alpha_i \mathsf{y}_i &= 0
\end{aligned}$$

Substitute back into Loss($\alpha$):

$$\begin{aligned}
\text{Loss}(\alpha) &:= \min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_i \alpha[\mathsf{y}_i(\langle \mathbf{x}, \mathbf{w}\rangle + b) - 1] \\
&= \min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 - \left\langle \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i, \mathbf{w} \right\rangle - b\sum_i \alpha_i \mathsf{y}_i + \sum_i \alpha_i \\
&= \frac{1}{2}\left\|\sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i\right\|_2^2 - \left\langle \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i, \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i \right\rangle + \sum_i \alpha_i && \left(\text{s.t. } \textstyle\sum_i \alpha_i \mathsf{y}_i = 0\right) \\
&= -\frac{1}{2}\left\|\sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i\right\|_2^2 + \sum_i \alpha_i && \left(\text{s.t. } \textstyle\sum_i \alpha_i \mathsf{y}_i = 0\right)
\end{aligned}$$

Therefore, we can write the dual problem as

$$\min_{\boldsymbol{\alpha}\geq 0} -\sum_i \alpha_i + \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j \mathsf{y}_i \mathsf{y}_j \left\langle \mathbf{x}_i, \mathbf{x}_j \right\rangle \quad \text{s.t.} \quad \sum_i \alpha_i \mathsf{y}_i = 0$$

We prefer this dual problem because it admits a very easy way to use a non-linear mapping $\mathbf{x} \xrightarrow{\phi} \phi(\mathbf{x})$ to transform non-linearly separable data $\mathbf{x}$ into linearly separable $\phi(\mathbf{x})$. After applying the unknown non-linear mapping, we get

$$\min_{\boldsymbol{\alpha}\geq 0} -\sum_i \alpha_i + \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j \mathsf{y}_i \mathsf{y}_j \left\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \right\rangle \quad \text{s.t.} \quad \sum_i \alpha_i \mathsf{y}_i = 0$$

which we can find *without explicitly applying* $\phi$ by using the "kernel trick" from section 7, writing the inner product directly as a non-linear function.

## 6 Soft-Margin Support Vector Machines

*Lecture 7*
*Jan 30*

One of the drawbacks of the hard-margin SVM is that the data must be linearly separable. That is, there must exist a non-zero margin between the data.

If we have a small number of outliers on the wrong side of the decision boundary, we can instead just penalize it in the loss. We do this by relaxing the constraint in hard-margin SVM and including failures in the objective function.

**Definition 6.1** (hinge loss)
Given label $\mathsf{y} \in \{-1, +1\}$ and score $\hat{y} := \langle \mathbf{x}, \mathbf{w} \rangle + b$, let $\mathsf{y}\hat{y}$ be the confidence.

Define $\ell_{\text{hinge}} = (1 - \mathsf{y}\hat{y})^+ = \begin{cases} 1 - \mathsf{y}\hat{y} & \mathsf{y}\hat{y} < 1 \\ 0 & \text{otherwise} \end{cases}$

In general, notate $x^+$ to mean $\max\{x, 0\}$.

Now, we can formulate the soft-margin SVM as

$$\boxed{\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 + C \cdot \sum_i (1 - \mathsf{y}_i\hat{y}_i)^+ \quad \text{s.t.} \quad \hat{y}_i = \langle \mathbf{x}_i, \mathbf{w} \rangle + b} \tag{6.a}$$

(margin maximization, regularization hyperparameter, error penalty). Notice that the hard-margin SVM is the limiting behaviour of the soft-margin SVM as $C \to \infty$.

**Why do we use the hinge loss?** Consider the probability that $\mathsf{Y} \neq \operatorname{sgn}(\hat{\mathsf{Y}})$

$$\Pr\left[\mathsf{Y} \neq \operatorname{sgn}(\hat{\mathsf{Y}})\right] = \Pr\left[\mathsf{Y}\hat{\mathsf{Y}} \le 0\right] = \mathbb{E}[\mathbb{I}[\mathsf{Y}\hat{\mathsf{Y}} \le 0]] =: \mathbb{E}[\ell_{0-1}(\mathsf{Y}\hat{\mathsf{Y}})]$$

We want to minimize $\mathbb{E}[\ell_{0-1}(\mathsf{Y}\hat{\mathsf{Y}})]$. Minimizing this value is hard because $\ell_{0-1}$ is discontinuous at 0 and has gradient $\mathbf{0}$ almost everywhere.

By Bayes' rule, we can rewrite as $\mathbb{E}_{\mathsf{X}}\,\mathbb{E}_{\mathsf{Y}|\mathsf{X}}[\ell_{0-1}(\mathsf{Y}\hat{\mathsf{Y}})]$. Then, we can minimize instead

$$\eta(\mathbf{x}) = \operatorname*{arg\,min}_{\hat{y} \in \mathbb{R}} \mathbb{E}_{\mathsf{Y}|\mathsf{X}=\mathbf{x}}[\ell_{0-1}(\mathsf{Y}\hat{y})]$$

since setting $\mathsf{Y} = \eta(\mathsf{X})$.

**Definition 6.2** (classification calibrated)
We say a loss function $\ell(\mathsf{y}\hat{y})$ is classification calibrated if for all $\mathbf{x}$,

$$\hat{\mathsf{y}}(\mathbf{x}) := \operatorname*{arg\,min}_{\hat{y} \in \mathbb{R}} \mathbb{E}_{\mathsf{Y}|\mathbf{X}=\mathbf{x}}[\ell(Y\hat{y})]$$

has the same sign as the Bayes rule $\eta(\mathbf{x})$.

Due to Bartlett, we have a helpful theorem

**Theorem 6.3** (characterization under convexity)
Any convex loss $\ell$ is classification calibrated if and only if $\ell$ is differentiable at 0 and $\ell'(0) < 0$.

**Corollary 6.4.** A classifier that minimizes the expected hinge loss also minimizes the expected 0-1 loss.

This theorem is also one of the big reasons why the perceptron cannot generalize well.

**Remark 6.5.** The perceptron loss $\ell(\mathsf{y}\hat{y}) = -\min\{\mathsf{y}\hat{y}, 0\}$ is not differentiable at 0, so it is not classification calibrated and cannot generalize.

**Generating the dual** Recall the soft-margin SVM

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 + C \cdot \sum_i (1 - \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b))^+$$

Notice that we can write $C \cdot (t)^+ = \max\{Ct, 0\} = \max_{0\le\alpha\le C} \alpha t$ to get

$$\min_{\mathbf{w},b} \max_{0\le\boldsymbol{\alpha}\le C} \frac{1}{2}\|\mathbf{w}\|_2^2 + \sum_i \alpha_i(1 - \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b))$$

As before, swap min with max:

$$\max_{0\le\boldsymbol{\alpha}\le C} \overbrace{\min_{\mathbf{w},b} \underbrace{\frac{1}{2}\|\mathbf{w}\|_2^2 + \sum_i \alpha_i(1 - \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b))}_{\text{Loss}(\mathbf{w},b,\alpha)}}^{\text{Loss}(\alpha)}$$

Now, set our optimality conditions

$$\frac{\partial \text{Loss}(\mathbf{w}, b, \alpha)}{\partial \mathbf{w}} = \mathbf{w} - \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i = \mathbf{0} \qquad \frac{\partial \text{Loss}(\mathbf{w}, b, \alpha)}{\partial b} = -\sum_i \alpha_i \mathsf{y}_i = 0$$

$$\mathbf{w} = \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i \qquad \sum_i \alpha_i \mathsf{y}_i = 0$$

and substitute into Loss($\alpha$):

$$\begin{aligned}
\text{Loss}(\alpha) &:= \frac{1}{2}\|\mathbf{w}\|_2^2 + \sum_i \alpha_i(1 - \mathsf{y}_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b)) \\
&= \frac{1}{2}\left\|\sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i\right\|_2^2 + \sum_i \alpha_i - \left\langle \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i, \sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i \right\rangle \\
&= -\frac{1}{2}\left\|\sum_i \alpha_i \mathsf{y}_i \mathbf{x}_i\right\|_2^2 + \sum_i \alpha_i
\end{aligned}$$

Switching from max to min and expanding the norm, we get

$$\boxed{\min_{0\le\boldsymbol{\alpha}\le C} -\sum_i \alpha_i + \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \quad \text{s.t.} \quad \sum_i \alpha_i \mathsf{y}_i = 0} \tag{6.b}$$

which is identical to the hard-margin SVM dual with an upper bound $C$ on $\boldsymbol{\alpha}$.

*Lecture 8*
*Feb 1*

Suppose we solve the dual (eq. 6.b) with optimal solution $\boldsymbol{\alpha}^*$. Then,

$$\mathbf{w}^* = \sum_i \alpha_i^* \mathsf{y}_i \mathbf{x}_i. \tag{6.c}$$

If we have a point on $H_{\pm 1}$, i.e., $\mathsf{y}\hat{y} = 1$, we can recover $b^*$ as $\mathsf{y} - \langle \mathbf{x}, \mathbf{w}^* \rangle$.

**Training by gradient descent** Suppose we have a minimization problem $\min_{\mathbf{x}} f(\mathbf{x})$. Then, to make a guess $\mathbf{x}$ better, set $\mathbf{x} \leftarrow \mathbf{x} - \eta \cdot \nabla_{\mathbf{x}} f(\mathbf{x})$ for some learning rate $\eta > 0$.

Given the problem

$$\min_{\mathbf{w}, b} \frac{1}{2\lambda} \|\mathbf{w}\|_2^2 + C \sum_i \ell(\mathsf{y}_i \hat{y}_i) \quad \text{where} \quad \hat{y}_i = \langle \mathbf{x}_i, \mathbf{w}, \mathbf{x}_i, \mathbf{w} \rangle + b$$

with loss function $\ell$, the gradient descent steps are

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} \left( \frac{1}{2\lambda} \|\mathbf{w}\|_2^2 + C \sum_i \ell(\mathsf{y}_i \hat{y}_i) \right) \\
&= \mathbf{w} - \eta \left[ \frac{\mathbf{w}}{\lambda} + C \sum_i \ell'(\mathsf{y}_i \hat{y}_i) \mathsf{y}_i \mathbf{x}_i \right] \\
b &\leftarrow b - \eta \cdot \nabla_b \left( \frac{1}{2\lambda} \|\mathbf{w}\|_2^2 + C \sum_i \ell(\mathsf{y}_i \hat{y}_i) \right) \\
&= b - \eta \left[ C \sum_i \ell'(\mathsf{y}_i \hat{y}_i) \mathsf{y}_i \right]
\end{aligned}$$

because $\nabla_{\mathbf{w}} \ell(\mathsf{y}_i \hat{y}_i) = \ell'(\mathsf{y}_i \hat{y}_i) \cdot \mathsf{y}_i \nabla_{\mathbf{w}}(\hat{y}_i) = \ell'(\mathsf{y}_i \hat{y}_i) \mathsf{y}_i \mathbf{x}_i$ and $\nabla_b \ell(\mathsf{y}_i \hat{y}_i) = \ell'(\mathsf{y}_i \hat{y}_i) \cdot \mathsf{y}_i \nabla_b(\hat{y}_i) = \ell'(\mathsf{y}_i \hat{y}_i) \cdot \mathsf{y}_i$.

If $\ell$ is hinge loss, we define the derivative $\ell'(t) = \begin{cases} -1 & t \leq 1 \\ 0 & t > 1 \end{cases}$.

If $\ell$ is perceptron loss, we define $\ell'(t) = \begin{cases} -1 & t \leq 0 \\ 0 & t > 1 \end{cases}$.

All other common loss functions are easily differentiable.

## 7 Reproducing Kernels

We have dealt with data that is perfectly linearly separable (hard-margin SVM) and mostly linearly separable (soft-margin SVM).

**Problem 7.1**

How can we use our existing techniques to classify a fully non-linearly separable dataset?

In the linear classifier, we used an affine function $\langle \mathbf{w}, \mathbf{x} \rangle + b$. Now, we define a quadratic classifier.

**Definition 7.2** (quadratic classifier)

A function $f : \mathbb{R}^d \to \mathbb{R}^d$ of the form $f(\mathbf{x}) = \langle \mathbf{x}, Q\mathbf{x} \rangle + \sqrt{2} \langle \mathbf{x}, \mathbf{p} \rangle + b$ where the weights to be learned are $Q \in \mathbb{R}^{d \times d}$, $\mathbf{p} \in \mathbb{R}^d$, and $b \in \mathbb{R}$.

Recall from linear algebra that for all $A$, $B$, $C$, $\langle AB, C \rangle = \left\langle B, A^\top C \right\rangle$ and $\langle A, BC \rangle = \left\langle AB^\top, C \right\rangle$.

**Definition 7.3** (matrix vectorization)

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, let $\overrightarrow{\mathbf{A}} \in \mathbb{R}^{mn}$ be its vectorization. That is,

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \implies \overrightarrow{\mathbf{A}} = \begin{bmatrix} a_{11} \\ a_{12} \\ \vdots \\ a_{1n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

Then, we can write the quadratic classifier as:

$$\begin{aligned} f(\mathbf{x}) &= \langle \mathbf{x}, Q\mathbf{x} \rangle + \sqrt{2} \langle \mathbf{x}, \mathbf{p} \rangle + b \\ &= \left\langle \mathbf{x}\mathbf{x}^\top, Q \right\rangle + \left\langle \sqrt{2}\mathbf{x}, \mathbf{p} \right\rangle + b \\ &= \left\langle \begin{bmatrix} \overrightarrow{\mathbf{x}\mathbf{x}^\top} \\ \sqrt{2}\mathbf{x} \\ 1 \end{bmatrix}, \begin{bmatrix} \overrightarrow{Q} \\ \mathbf{p} \\ b \end{bmatrix} \right\rangle \end{aligned}$$

If we write $\phi(\mathbf{x}) = (\overrightarrow{\mathbf{x}\mathbf{x}^\top}, \sqrt{2}\mathbf{x}, 1)^\top$ and $\mathbf{w} = (\overrightarrow{Q}, \mathbf{p}, b)^\top$, then we can write $f$ as

$$f(\mathbf{x}) = \langle \phi(\mathbf{x}), \mathbf{w} \rangle$$

but this really blows up the dimension to $\mathbb{R}^{d^2+d+1}$. Recall that in the dual forms of SVM, all we need is to know the inner product $\langle \phi(\mathbf{x}), \phi(\mathbf{w}) \rangle$. With our new $\phi$, we get

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) := \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle &= \left\langle \begin{bmatrix} \overrightarrow{\mathbf{x}\mathbf{x}^\top} \\ \sqrt{2}\mathbf{x} \\ 1 \end{bmatrix}, \begin{bmatrix} \overrightarrow{\mathbf{z}\mathbf{z}^\top} \\ \sqrt{2}\mathbf{z} \\ 1 \end{bmatrix} \right\rangle \\ &= \left\langle \overrightarrow{\mathbf{x}\mathbf{x}^\top}, \overrightarrow{\mathbf{z}\mathbf{z}^\top} \right\rangle + \left\langle \sqrt{2}\mathbf{x}, \sqrt{2}\mathbf{z} \right\rangle + 1 \\ &= \langle \mathbf{x}, \mathbf{z} \rangle^2 + 2 \langle \mathbf{x}, \mathbf{z} \rangle + 1 \\ &= (\langle \mathbf{x}, \mathbf{z} \rangle + 1)^2 \end{aligned}$$

This process is easily reproducable for a given $\phi$. What about the other direction?

**Definition 7.4** (reproducing kernel)

We call $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ a reproducing kernel if there exists some $\phi : \mathcal{X} \to \mathcal{H}$ so that $\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle = k(\mathbf{x}, \mathbf{z})$.

**Remark 7.5.** When such a kernel exists, it may not be unique.

For example, the kernels $\phi(\mathbf{x}) = [x_1^2, \sqrt{2}x_1x_2, x_2^2] \in \mathbb{R}^3$ and $\psi(\mathbf{x}) = [x_1^2, x_1x_2, x_1x_2, x_2^2] \in \mathbb{R}^4$ have the same inner product $\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle = \langle \psi(\mathbf{x}), \psi(\mathbf{z}) \rangle$.

**Theorem 7.6** (Mercer's theorem)
$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel if and only if for any $n \in \mathbb{N}$ and any $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{X}$, the kernel matrix $K_{ij} := k(\mathbf{x}_i, \mathbf{x}_j)$ is symmetric and positive semi-definite.

Recall from linear algebra: $K$ is symmetric if $K_{ij} = K_{ji}$ for all indices, and positive semi-definite if $\langle \boldsymbol{\alpha}, K\boldsymbol{\alpha} \rangle \geq 0$ for all vectors $\boldsymbol{\alpha}$.

The proof is extremely convoluted and well beyond the scope of the course.

**Example 7.7.** The following are kernels:

- the polynomial kernel $k(\mathbf{x}, \mathbf{z}) = (\langle \mathbf{x}, \mathbf{z} \rangle + 1)^p$ for hyperparameter $p$,
- the Gaussian kernel $k(\mathbf{x}, \mathbf{z}) = \exp\big(-\|\mathbf{x} - \mathbf{z}\|_2^2 / \sigma\big)$ for hyperparameter $\sigma$, and
- the Laplace kernel $k(\mathbf{x}, \mathbf{z}) = \exp(-\|\mathbf{x} - \mathbf{z}\|_2 / \sigma)$ for hyperparameter $\sigma$

Now, we can substitute our expression for the inner product to eqs. 6.a and 6.b, the primal and dual of the soft-margin SVM:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 + C \cdot \sum_i (1 - \mathsf{y}_i \hat{y}_i)^+ \quad \text{s.t.} \quad \hat{y}_i = \langle \phi(\mathbf{x}_i), \mathbf{w} \rangle$$

$$\min_{0 \leq \boldsymbol{\alpha} \leq C} -\sum_i \alpha_i + \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \quad \text{s.t.} \quad \sum_i \alpha_i \mathsf{y}_i = 0$$

Once we solve $\boldsymbol{\alpha}^*$, we can try to recover $\mathbf{w}^*$ as in eq. 6.c

$$\mathbf{w}^* = \sum \alpha_i^* \mathsf{y}_i \phi(\mathbf{x}_i)$$

but this will not work since we do not know $\phi$ explicitly. Instead, we only need to compute the score function

$$\begin{aligned} f(\mathbf{x}) &:= \langle \phi(\mathbf{x}), \mathbf{w}^* \rangle \\ &= \Big\langle \phi(\mathbf{x}), \sum \alpha_i^* \mathsf{y}_i \phi(\mathbf{x}_i) \Big\rangle \\ &= \sum \alpha_i^* \mathsf{y}_i \, \langle \phi(\mathbf{x}), \phi(\mathbf{x}_i) \rangle \\ &= \sum \alpha_i^* \mathsf{y}_i k(\mathbf{x}, \mathbf{x}_i) \end{aligned}$$

and return $\operatorname{sgn}(f(\mathbf{x}))$.

# 8 Gradient Descent

*Lecture 9*
*Feb 6*

All of our machine learning models so far have been expressed as optimization problems (eqs. 4.a, 5.a and 6.a).

**Remark 8.1.** Optimization problems are identical up to constants. That is,

$$\min_{\mathbf{x}} f(\mathbf{x}) = \min_{\mathbf{x}} c \cdot f(x)$$

if $c$ has no $\mathbf{x}$-dependence.

We can consider now a generic optimization problem $\min\limits_{\mathbf{x}} f(\mathbf{x})$.

Assume that $f(\mathbf{x})$ is differentiable with gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$.

**Notation.** Given the generic optimization problem, write $f^* := \min_{\mathbf{x}} f(x)$ for the optimal value and $x^* := \arg\min_{\mathbf{x}} f(x)$ for the optimal parameter.

Then, we can define gradient descent.

**Definition 8.2** (gradient descent)
Choose an initial point $\mathbf{x}^{(0)} \in \mathbb{R}^d$ and repeat

$$x^{(k)} = x^{(k-1)} - \underbrace{t}_{\text{step size}} \cdot \nabla f(x^{(k-1)})$$

$k = 1, 2, \ldots$ for some step size $t > 0$ until satisfied.

Intuitively, we are walking "down" the function by checking for a downwards slope and taking a $t$-sized step down that slope.

For example, the perceptron (section 2) with optimization problem

$$\min_{\mathbf{w}} f(\mathbf{w}) = \min_{\mathbf{w}} -\frac{1}{n} \sum_i \mathsf{y}_i \langle \mathbf{w}, \mathbf{x}_i \rangle \mathbb{I}[\text{mistake on } \mathbf{x}_i]$$

with gradient

$$\nabla_{\mathbf{w}} f(\mathbf{w}) = -\frac{1}{n} \sum_i \mathsf{y}_i \mathbf{x}_i \mathbb{I}[\text{mistake on } \mathbf{x}_i]$$

leads us to the gradient descent update

$$\mathbf{w} \leftarrow \mathbf{w} + t \left[ \frac{1}{n} \sum_i \mathsf{y}_i \mathbf{x}_i \mathbb{I}[\text{mistake on } \mathbf{x}_i] \right]$$

This is very expensive, since we need to iterate over our entire training data for each update. Since the gradient is just a sample mean, we can make an estimation

$$\widetilde{\nabla_{\mathbf{w}} f(\mathbf{w})} = \mathsf{y}_I \mathbf{x}_I \mathbb{I}[\text{mistake on } \mathbf{x}_I]$$

after picking a random index $I \in_{\mathsf{R}} \{1, \dots, n\}$. This is an unbiased estimator of the sample mean. Doing this, i.e.,

$$\mathbf{w} \leftarrow \mathbf{w} + t\mathsf{y}_I \mathbf{x}_I \mathbb{I}[\text{mistake on } \mathbf{x}_I]$$

is called stochastic gradient descent. Since it is (very) inaccurate, it will take many more iterations to converge.

For a more complex example, consider the soft-margin SVM (section 6) with optimization problem

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_i \ell_{\mathsf{hinge}}(1 - \mathsf{y}_i \hat{y}_i) \quad \text{s.t.} \quad \hat{y}_i = \langle \mathbf{x}_i, \mathbf{w} \rangle + b$$

We calculate two gradients $\nabla_{\mathbf{w}}$ and $\nabla_b$ to get

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - t\left[\mathbf{w} + C\sum_i \ell'_{\mathsf{hinge}}(\mathsf{y}_i\hat{y}_i)\mathsf{y}_i\mathbf{x}_i\right] \\ b &\leftarrow b - t\left[C\sum_i \ell'_{\mathsf{hinge}}(\mathsf{y}_i\hat{y}_i)\mathsf{y}_i\right] \end{aligned}$$

**Motivating gradient descent** Suppose we take the Taylor expansion of $f$ at the current iterate $\mathbf{x}$. Then, we can say

$$f(\mathbf{y}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{1}{2t}\|\mathbf{y} - \mathbf{x}\|_2^2$$

and take the minimization with respect to $\mathbf{y}$ on both sides

$$\min_{\mathbf{y}} f(\mathbf{y}) \approx \min_{\mathbf{y}} \left[\underbrace{f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{1}{2t}\|\mathbf{y} - \mathbf{x}\|_2^2}_{g(\mathbf{y})}\right]$$

so that we can write

$$\begin{aligned} \frac{\partial g}{\partial \mathbf{y}} = 0 + \nabla f(\mathbf{x}) + \frac{1}{t}(y - x) &= 0 \\ t\nabla f(\mathbf{x}) + \mathbf{y} - \mathbf{x} &= 0 \\ \mathbf{y} &= \mathbf{x} - t\nabla f(\mathbf{x}) \end{aligned}$$

which is our gradient descent formula.

**Applying gradient descent** We cannot set the step size too large (it will diverge) or too small (it will be too slow). How do we choose the step size?

**Definition 8.3** (convexity)
A function $f$ is convex if $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top(\mathbf{y} - \mathbf{x})$ for any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$.

We also want to characterize the smoothness.

**Definition 8.4** (Lipschitz continuity)
Given convex and differentiable $f$, we say $f$ is $L$-smooth or $L$-Lipschitz continuous for $L > 0$ if the matrix

$$LI - \nabla^2 f(\mathbf{x})$$

is positive semi-definite for every $x$ (we write $LI \succeq \nabla^2 f(x)$).

Then, we can characterize the convergence rate.

**Theorem 8.5** (convergence rate for convex case)
Gradient descent with fixed step size $t \leq 1/L$ satisfies

$$f(\mathbf{x}^{(k)}) - f^* \leq \frac{\left\|\mathbf{x}^{(0)} - \mathbf{x}^*\right\|_2^2}{2tk}$$

We say gradient descent has convergence rate $\mathcal{O}(1/k)$ (i.e., a bound of $f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*) \leq \varepsilon$ takes $\mathcal{O}(1/\varepsilon)$ iterations).

*Proof.* Recall the mean value theorem allows us to write the Lagrangian

$$f(\mathbf{y}) = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{1}{2}(\mathbf{y} - \mathbf{x})^\top \nabla^2 f(\mathbf{a})(\mathbf{y} - \mathbf{x})$$

where $\mathbf{a}$ is on the line between $\mathbf{x}$ and $\mathbf{y}$. Then, since $LI \succeq \nabla^2 f(\mathbf{a})$, we have

$$\begin{aligned} f(\mathbf{y}) &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{L}{2}(\mathbf{y} - \mathbf{x})^\top (\mathbf{y} - \mathbf{x}) \\ &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|_2^2 \end{aligned}$$

Now, plug in $\mathbf{y} = \mathbf{x}^+ := \mathbf{x} - t\nabla f(\mathbf{x})$ (i.e., do the gradient update) to get

$$\begin{aligned} f(\mathbf{x}^+) &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{x} - t\nabla f(\mathbf{x}) - \mathbf{x}) + \frac{L}{2}\|\mathbf{x} - t(\nabla f(\mathbf{x})) - \mathbf{x}\|_2^2 \\ &= f(\mathbf{x}) - t\|\nabla f(\mathbf{x})\|_2^2 + \frac{Lt^2}{2}\|\nabla f(\mathbf{x})\|_2^2 \\ &= f(\mathbf{x}) - (1 - \frac{1}{2}Lt)t\|\nabla f(\mathbf{x})\|_2^2 \end{aligned}$$

Since $t \leq \frac{1}{L}$, we have $(1 - \frac{1}{2}Lt) \geq \frac{1}{2}$ and we can conclude that

$$f(\mathbf{x}^+) \leq f(\mathbf{x}) - \frac{1}{2}t\|\nabla f(\mathbf{x})\|_2^2 \qquad (\star)$$

which means that we have decreased the function value by at least $\frac{t}{2}\|\nabla f(\mathbf{x})\|_2^2$.

Recall that $f$ is convex. Then, by definition, $f(\mathbf{x}^*) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{x}^* - \mathbf{x})$. Equivalently,

$$f(\mathbf{x}) \leq f(\mathbf{x}^*) + \nabla f(\mathbf{x})^\top (\mathbf{x} - \mathbf{x}^*)$$

and by ($\star$) we can say

$$\begin{aligned}
f(\mathbf{x}^+) &\leq f(\mathbf{x}^*) + \nabla f(\mathbf{x})^\top(\mathbf{x}-\mathbf{x}^*) - \frac{t}{2}\|\nabla f(\mathbf{x})\|_2^2 \\
f(\mathbf{x}^+) - f(\mathbf{x}^*) &\leq \nabla f(\mathbf{x})^\top(\mathbf{x}-\mathbf{x}^*) - \frac{t}{2}\|\nabla f(\mathbf{x})\|_2^2 \\
&= \frac{1}{2t}\left(2t\nabla f(\mathbf{x})^\top(\mathbf{x}-\mathbf{x}^*) - t^2\|\nabla f(\mathbf{x})\|_2^2\right) \\
&= \frac{1}{2t}\left((2t\nabla f(\mathbf{x})^\top(\mathbf{x}-\mathbf{x}^*) - t^2\|\nabla f(\mathbf{x})\|_2^2 - \|\mathbf{x}-\mathbf{x}^*\|_2^2) + \|\mathbf{x}-\mathbf{x}^*\|_2^2\right) \\
&= \frac{1}{2t}\left(-\|\mathbf{x}-t\nabla f(\mathbf{x})-\mathbf{x}^*\| + \|\mathbf{x}-\mathbf{x}^*\|_2^2\right) \\
&= \frac{1}{2t}\left(\|\mathbf{x}-\mathbf{x}^*\|_2^2 - \|\mathbf{x}^+-\mathbf{x}^*\|_2^2\right)
\end{aligned}$$

If we define $\mathbf{x}^+ := \mathbf{x}^{(i)}$ and $\mathbf{x} := \mathbf{x}^{(i-1)}$, we have

$$\begin{aligned}
f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*) &\leq \frac{1}{2t}\left(\left\|\mathbf{x}^{(i-1)}-\mathbf{x}^*\right\|_2^2 - \left\|\mathbf{x}^{(i)}-\mathbf{x}^*\right\|_2^2\right) \\
\sum_{i=1}^k \left[f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*)\right] &\leq \sum_{i=1}^k \frac{1}{2t}\left(\left\|\mathbf{x}^{(i-1)}-\mathbf{x}^*\right\|_2^2 - \left\|\mathbf{x}^{(i)}-\mathbf{x}^*\right\|_2^2\right) \\
\sum_{i=1}^k f(\mathbf{x}^{(i)}) - kf(\mathbf{x}^*) &\leq \frac{1}{2t}\left(\left\|\mathbf{x}^{(0)}-\mathbf{x}^*\right\|_2^2 - \left\|\mathbf{x}^{(k)}-\mathbf{x}^*\right\|_2^2\right) \\
&\leq \frac{1}{2t}\left(\left\|\mathbf{x}^{(0)}-\mathbf{x}^*\right\|_2^2\right) \\
\frac{1}{k}\sum_{i=1}^k f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*) &\leq \frac{1}{2tk}\left(\left\|\mathbf{x}^{(0)}-\mathbf{x}^*\right\|_2^2\right)
\end{aligned}$$

Finally, because each step decreases, we must have $f(\mathbf{x}^{(k)}) \leq \frac{1}{k}\sum_{i=1}^k f(\mathbf{x}^{(i)})$. That is,

$$f(\mathbf{x}^{(k)}) - f^* \leq \frac{1}{k}\sum_{i=1}^k f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*) \leq \frac{1}{2tk}\left(\left\|\mathbf{x}^{(0)}-\mathbf{x}^*\right\|_2^2\right)$$

as desired. □

*Lecture 10*
*Feb 8*

We have a stronger sense of convexity that gives a stronger convergence rate.

**Definition 8.6** ($m$-strong convexity)
For some $m > 0$, $f$ is m-strong convex if $f(\mathbf{x}) - m\|\mathbf{x}\|_2^2$ is convex. We write $LI \succeq \nabla^2 f(\mathbf{x}) \succeq mI$.

**Theorem 8.7** (convergence rate for strong convexity)
Let $f$ be differentiable, $m$-strong convex, and $L$-smooth. Then, gradient descent with fixed step size $t \leq 2/(m+L)$ satisfies

$$f(\mathbf{x}^{(k)}) - f^* \leq \gamma^k \frac{L}{2}\left\|\mathbf{x}^{(0)}-\mathbf{x}^*\right\|_2^2$$

where $0 < \gamma < 1$.

The rate here is $\mathcal{O}(\gamma^k)$ which is exponentially fast. That is, a bound $f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*) < \varepsilon$ can be achieved using only $\mathcal{O}\left(\log_{1/\gamma}(1/\varepsilon)\right)$ iterations, much better than before.

Alternatively, we can make a weaker assumption and ask for a weaker result. In a non-convex function, there are (potentially many) local minima. Instead of asking for small $\left\|f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*)\right\|_2$, we only need $\|\nabla f(\mathbf{x})\|$.

**Theorem 8.8** (convergence rate for non-convex case)

Suppose $f$ is differentiable, $L$-smooth, and non-convex. Then, gradient descent with fixed step size $t \leq 1/L$ satisfies

$$\min_{i=0,\dots,k} \left\|\nabla f(\mathbf{x}^{(i)})\right\|_2 \leq \sqrt{\frac{2(f(\mathbf{x}^{(0)}) - f^*)}{t(k+1)}}$$

The rate $\mathcal{O}\left(1/\sqrt{k}\right)$ for finding stationary points cannot be improved by any deterministic algorithm.

However, all these require that the gradient $\nabla f(\mathbf{x})$ is known to us.

**Stochastic gradient descent** Recall that we introduced the case for perceptron where we update using one data point instead of the full dataset.

Consider some decomposable optimization with unreasonably large $n$

$$\min_{\mathbf{w}} \frac{1}{n} \sum_i f_i(\mathbf{w})$$

where we assume $\nabla f_i(\mathbf{w})$ exists for all $i$. Then, the two gradient descent updates

$$\mathbf{w} \leftarrow \mathbf{w} - t\frac{1}{n}\sum_i \nabla f_i(\mathbf{w})$$
$$\mathbf{w} \leftarrow \mathbf{w} - t \cdot \nabla f_I(\mathbf{w})$$

(where $I$ is a uniformly random index) have the same expected value. Notice that the “full” gradient descent will have true time complexity $\mathcal{O}(n/\varepsilon)$ because each step takes $\mathcal{O}(n)$ time to calculate.

The stochastic version takes just $\mathcal{O}(1/\varepsilon^2)$ time.

To summarize these theorems:

| Case | Hessian assumption | Iterations for $\varepsilon$ error | Step size |
|---:|:---:|:---:|:---:|
| Non-convex | $LI \succeq \nabla^2 f(\mathbf{x})$ | $\mathcal{O}(1/\varepsilon^2)$ | $t \leq 1/L$ |
| Convex | $LI \succeq \nabla^2 f(\mathbf{x})$ | $\mathcal{O}(1/\varepsilon)$ | $t \leq 1/L$ |
| $m$-strong convex | $LI \succeq \nabla^2 f(\mathbf{x}) \succeq mI$ | $\mathcal{O}(\log(1/\varepsilon))$ | $t \leq 2/(m+L)$ |
| Stochastic convex | $LI \succeq \nabla^2 f(\mathbf{x})$ | $\mathcal{O}(1/\varepsilon^2)$ | $t = 1/k$ |

In general, we will want to use stochastic gradient descent when $n > C_1/\varepsilon$ and full gradient descent when $n < C_2/\varepsilon$ for some constants $C_1$, $C_2$.

# Chapter 2

# Neural Networks

We can finally progress from 30- to 60-year old algorithms to stuff people actually use now. Recall the XOR dataset (ex. 2.10). We showed that it is not linearly separable, so it cannot be learned by perceptron (thm. 2.11).

One way to deal with this is to use a richer model (e.g., a quadratic classifier) or to lift the data through some feature map $\phi$. These two approaches are equivalent due to reproducing kernels.

A neural network tries to learn the feature map *and* the linear classifier simultaneously.

## 9 Multilayer Perceptron

We can set up the following layers:

- input layer $\mathbf{x} \in \mathbb{R}^2$
- linear layer $\mathbf{z} = \mathbf{U}\mathbf{x} + \mathbf{c}$ for learnable parameters $\mathbf{U} \in \mathbb{R}^{2\times 2}$ and $\mathbf{c} \in \mathbb{R}^2$
- hidden layer $\mathbf{h} = \sigma(\mathbf{z})$ for some non-linear $\sigma$
- prediction layer $\hat{y} = \langle \mathbf{h}, \mathbf{w} \rangle + b$ for learnable parameters $\mathbf{w} \in \mathbb{R}^2$ and $b \in \mathbb{R}$
- output layer $\operatorname{sgn}(\hat{y})$ or $\operatorname{sigmoid}(\hat{y})$

In total, we need to learn $\mathbf{U}$, $\mathbf{c}$, $\mathbf{w}$, and $b$ (here, 9 parameters).

> **Example 9.1.** XOR dataset is learnable with a 2-layer neural network. Let
>
> $$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} 2 \\ -4 \end{bmatrix}, \quad b = -1$$
>
> and let $\sigma(t) = \max\{t, 0\}$ (the ReLU activation function).
>
> Then, $\operatorname{sgn}(\langle \sigma(\mathbf{U}\mathbf{x} + \mathbf{c}), \mathbf{w} \rangle + b)$ works.

To do a multi-class classification, simply have a bunch of $\hat{y}$'s in a vector $\hat{\mathbf{y}} = \mathbf{W}\mathbf{h} + \mathbf{b}$ and make a prediction vector $\hat{\mathbf{p}} = \operatorname{softmax}(\hat{\mathbf{y}})$.

**Remark 9.2.** The hidden layer $\sigma$ *must* be non-linear. Otherwise, the composition of linear layers is just a linear layer and we gain nothing.

There are a lot of options for $\sigma$:

- $\texttt{relu}(t) = t_+$
- $\texttt{elu}(t) = t_+ + t_-(\exp(t) - 1)$
- $\texttt{sgm}(t) = 1/(1 + \exp(-t))$
- $\tanh(t) = 1 - 2\texttt{sgm}(t)$

We can also stack several layers together, repeating the pattern of linear layer + non-linear layer.

To train, we need a loss function $\ell$ and a dataset $\mathcal{D} = \{(\mathbf{x}_i, \mathsf{y}_i)\}$

**Notation.** Write $[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i, \mathbf{w})$ to mean $\ell[f(\mathbf{x}_i, \mathbf{w}), \mathsf{y}_i]$.

We can express the neural network as a minimization problem

$$\min_{\mathbf{w}} \frac{1}{n} \sum_i [\ell \circ f](\mathbf{x}_i, \mathsf{y}_i, \mathbf{w}) \tag{9.a}$$

which gives the gradient descent rule

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{1}{n} \sum_i \nabla[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i, \mathbf{w})$$

for learning rate $\eta$. This requires a full pass over the dataset for each step.

Instead of doing ordinary stochastic gradient descent, we can minibatch by picking a random subset $B \subseteq \{1, \dots, n\}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{1}{|B|} \sum_{i \in B} \nabla[\ell \circ f](\mathbf{x}_i, \mathsf{y}_i, \mathbf{w})$$

which trades off variance and computation cost.

*Lecture 11*
*Feb 13*

The learning rate has diminishing returns. Instead of keeping a constant $\eta$, we can paramaterize $\eta_t$ and say something like

$$\eta_t = \begin{cases} \eta_0 & t \leq t_0 \\ \eta_0/10 & t_0 < t \leq t_1 \\ \eta_0/100 & t_1 < t \end{cases}$$

for an initial $\eta_0$ and specific epochs $t_0, t_1$. Alterntaively, we can use sublinear decay $\eta_t = \eta_0/(1 + ct)$ or $\eta_t = \eta_0/\sqrt{1 + ct}$ for some constant $c$.

We need to calculate a lot of partial derivatives with respect to matrices.

**Definition 9.3**

Let $y(\mathbf{X}) \in \mathbb{R}$ and $\mathbf{X} = [X_{ij}] \in \mathbb{R}^{m\times n}$. Then, we define the partial derivative of $y$ w.r.t. $\mathbf{X}$ as

$$\frac{\partial y}{\partial \mathbf{X}} = \left[\frac{\partial y}{\partial X_{ij}}\right] = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} & \frac{\partial y}{\partial X_{12}} & \cdots & \frac{\partial y}{\partial X_{1n}} \\ \frac{\partial y}{\partial X_{21}} & \frac{\partial y}{\partial X_{22}} & \cdots & \frac{\partial y}{\partial X_{2n}} \\ \vdots & \vdots & & \vdots \\ \frac{\partial y}{\partial X_{m1}} & \frac{\partial y}{\partial X_{m2}} & \cdots & \frac{\partial y}{\partial X_{mn}} \end{bmatrix} \in \mathbb{R}^{m\times n}$$

as a matrix.

The best way to do this is to just "guess" analogous to scalar calculus, then check that the dimension is right (i.e., $\dim \frac{\partial y}{\partial \mathbf{X}} = \dim \mathbf{X}$)

Consider the forward pass for NN width $k$ and output dimension $c$:

$$\begin{aligned} \mathbf{x} &= \text{input} & \mathbf{x} &\in \mathbb{R}^{d\times 1} \\ \mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b}_1 & \mathbf{W} &\in \mathbb{R}^{k\times d},\ \mathbf{z}, \mathbf{b}_1 \in \mathbb{R}^{k\times 1} \\ \mathbf{h} &= \mathrm{ReLU}(\mathbf{z}) & \mathbf{h} &\in \mathbb{R}^{k\times 1} \\ \boldsymbol{\theta} &= \mathbf{U}\mathbf{h} + \mathbf{b}_2 & \mathbf{U} &\in \mathbb{R}^{c\times k},\ \boldsymbol{\theta}, \mathbf{b}_2 \in \mathbb{R}^{c\times 1} \\ J &= \frac{1}{2}\|\boldsymbol{\theta} - \mathbf{y}\|_2^2 & \mathrm{y} &\in \mathbb{R}^{c\times 1},\ J \in \mathbb{R} \end{aligned}$$

Now, we can apply the chain rule to find our desired gradients:

$$\begin{aligned} \frac{\partial J}{\partial \boldsymbol{\theta}} &= \boldsymbol{\theta} - \mathbf{y} \\ \frac{\partial J}{\partial \mathbf{U}} &= \frac{\partial J}{\partial \boldsymbol{\theta}} \circ \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{U}} = \underbrace{(\boldsymbol{\theta} - \mathbf{y})}_{c\times 1} \underbrace{\mathbf{h}^\top}_{1\times k} && \text{(to get } c \times k\text{)} \\ \frac{\partial J}{\partial \mathbf{b}_2} &= \frac{\partial J}{\partial \boldsymbol{\theta}} \circ \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{b}_2} = \underbrace{\boldsymbol{\theta} - \mathbf{y}}_{c\times 1} && \text{(already has right dimensions)} \\ \frac{\partial J}{\partial \mathbf{h}} &= \frac{\partial J}{\partial \boldsymbol{\theta}} \circ \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{h}} = \underbrace{\mathbf{U}^\top}_{k\times c} \underbrace{(\boldsymbol{\theta} - \mathbf{y})}_{c\times 1} && \text{(to get } k \times 1\text{)} \\ \frac{\partial J}{\partial \mathbf{z}} &= \frac{\partial J}{\partial \mathbf{h}} \circ \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \underbrace{\mathbf{U}^\top(\boldsymbol{\theta} - \mathbf{y})}_{k\times 1} \odot \underbrace{\mathrm{ReLU}'(\mathbf{z})}_{k\times 1} && \text{(using } \odot \text{ to keep the dimension)} \\ \frac{\partial J}{\partial \mathbf{W}} &= \frac{\partial J}{\partial \mathbf{z}} \circ \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \underbrace{(\mathbf{U}^\top(\boldsymbol{\theta} - \mathbf{y}) \odot \mathrm{ReLU}'(\mathbf{z}))}_{k\times 1} \underbrace{\mathbf{x}^\top}_{1\times d} && \text{(to get } k \times d\text{)} \\ \frac{\partial J}{\partial \mathbf{b}_1} &= \frac{\partial J}{\partial \mathbf{z}} \circ \frac{\partial \mathbf{z}}{\partial \mathbf{b}_1} = \underbrace{(\mathbf{U}^\top(\boldsymbol{\theta} - \mathbf{y}) \odot \mathrm{ReLU}'(\mathbf{z}))}_{k\times 1} && \text{(already has right dimensions)} \end{aligned}$$

where $\odot$ is the Hadamard (element-wise) product, i.e.,

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \end{bmatrix} \odot \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_d \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_d b_d \end{bmatrix}$$

for two matrices of identical dimension.

Existing frameworks like TensorFlow will automatically do this.

**Theorem 9.4** (universal approximation theorem by 2-layer NNs)
For any continuous function $f : \mathbb{R}^d \to \mathbb{R}^c$ and any $\varepsilon > 0$, there exists $k \in \mathbb{N}$, $\mathbf{W} \in \mathbb{R}^{k \times d}$, $\mathbf{b} \in \mathbb{R}^k$, and $\mathbf{U} \in \mathbb{R}^{c \times k}$ such that

$$\sup_{\mathbf{x}} \|f(\mathbf{x}) - g(\mathbf{x})\|_2 < \varepsilon$$

where $g(\mathbf{x}) = \mathbf{U}(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}))$ and $\sigma$ is element-wise ReLU.

Informally, a 2-layer NN can approximate any continuous function arbitrarily closely provided it is wide enough with a large number of parameters.

However, it's not very efficient. In the worst case, a 2-layer MLP needs $k = \exp(1/\varepsilon)$ but a 3-layer MLP can get away with $k = \text{poly}(1/\varepsilon)$. Deeper networks will have even smaller dimensionality requirements.

To help avoid overfitting, we can apply dropout. For each minibatch, randomly select some hidden neurons to be active with probability $q$ (and pretend the rest of them don't exist). Then, each training minibatch gets a "different" network, so it's harder for neurons to "collude" to get overfitting. To make sure that dropout does not affect the overall expectation, multiply each $\mathbf{h}$ by $1/q$ during the back-propagation.

We can also do batch normalization to ensure that the mean and variance of all the minibatches are the same.

## 10 Convolutional Neural Networks

*Lecture 12*
*Feb 15*

An MLP has a lot of parameters to learn. Instead of densely connecting every node in the input layer to the hidden layer, only connect some of them (i.e., make $\mathbf{W}$ sparse).

Also, to reduce the number of parameters even more, make a bunch of the weights the same. Following a certain pattern, we get a convolution. These are useful for image processing/classification/segmentation but not for NLP.

The layers of CNN are roughly:

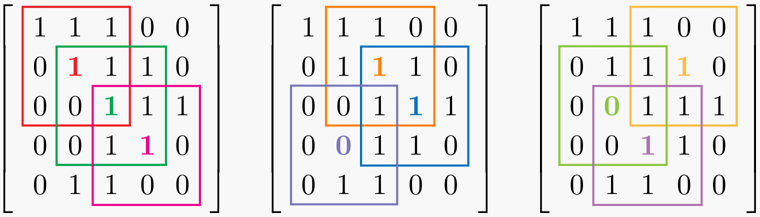
- feature extraction: a series of convolutions + ReLUs. We use a sliding window to reduce the dimensions of the input while pooling inputs together to increase width to make up for decreased size.
- vectorization: convert the matrix into a vector
- classification: a fully connected layer (i.e., MLP)
- probabilistic distribution: a softmax activation function

To process an image, split into sepraate channels for RGB values, then treat as a matrix of values. We will learn a kernel for the convolution with stochastic gradient descent.

**Example 10.1.** To calculate the convolution

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

we can find each coloured value by taking the tensor inner product (i.e., the inner product of the vectorization) of the kernel with the kernel-sized region around a value:



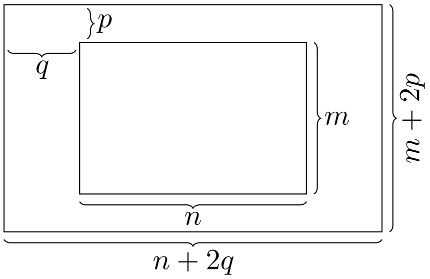
Convolutions have been shown to represent human visual cognition. Traditional image processing also uses convolutions. For example, edge detection and Gaussian smoothing.

For multi-channel input, "stack" the channels and use a "cube" (tensor) kernel. We can also apply a bias term $b \in \mathbb{R}$ to the output tensor (add $b$ to every element).

In a CNN layer, we increase channels to account for decreased resolution. For example, with 3 RGB input channels, we might learn 5 different $3 \times 3 \times 3$ kernels. Then, we will end up with 5 output channels.

We can also control the size of the step taken during convolution. Instead of always moving 1-left and 1-down, we can have a larger stride. However, we want overlap between windows, so always make sure that the stride is less than the kernel size. We can also control the padding, adding 0s as necessary to keep boundary information.

Suppose we have input size $\overbrace{m \times n}^{\text{typical } m=n=224} \times c_{in}$, kernel size $\overbrace{a \times b}^{\text{typical } a=b=5} \times c_{in}$, stride $\overbrace{s \times t}^{\text{typical } s=t=1,2}$, and padding $\overbrace{p \times q}^{\text{typical } p=q}$ so that the preprocesssed input looks like



Then, the output size will be

$$\left\lfloor 1 + \frac{m + 2p - a}{s} \right\rfloor \times \left\lfloor 1 + \frac{n + 2q - b}{t} \right\rfloor$$

If we want the input and output to have the "same" size, set

$$p = \left\lceil \frac{m(s-a) + a - s}{2} \right\rceil \quad \text{and} \quad q = \left\lceil \frac{n(t-1) + b - t}{2} \right\rceil$$

---

*...one reading week later...*

---

*Lecture 13*
*Feb 27*

Recall the convolution of $\mathbf{X} = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}$ and $\mathbf{W} = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix}$:

$$\mathbf{W} * \mathbf{X} = \begin{bmatrix} w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11} & w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12} \\ w_{00}x_{10} + w_{01}x_{11} + w_{10}x_{20} + w_{11}x_{21} & w_{00}x_{11} + w_{01}x_{12} + w_{10}x_{21} + w_{11}x_{22} \end{bmatrix}$$

such that the vectorization is

$$\operatorname{Vector}(\mathbf{W} * \mathbf{X}) = \begin{bmatrix} w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11} \\ w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12} \\ w_{00}x_{10} + w_{01}x_{11} + w_{10}x_{20} + w_{11}x_{21} \\ w_{00}x_{11} + w_{01}x_{12} + w_{10}x_{21} + w_{11}x_{22} \end{bmatrix}$$

This is a linear transformation. Therefore, we can design a <u>circulant matrix</u> $\mathbf{W}_{\mathsf{circ}}$ such that $\mathbf{W}_{\mathsf{circ}} \operatorname{Vector}(\mathbf{X}) = \operatorname{Vector}(\mathbf{W} * \mathbf{X})$. Define

$$\mathbf{W}_{\mathsf{circ}} = \begin{bmatrix} w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 & 0 & 0 & 0 \\ 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 \\ 0 & 0 & 0 & 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} \end{bmatrix}$$

and it is clear that $\mathbf{W}_{\mathsf{circ}} \operatorname{Vector}(\mathbf{X}) = \operatorname{Vector}(\mathbf{W} * \mathbf{X})$.

Now, notice that we only need to learn $|\mathbf{W}| = 4$ weights instead of $|\mathbf{W}_{\mathsf{circ}}| = 9 \times 4 = 36$ weights.

We can also down-sample the input size using <u>pooling</u>. Just like convolution, we take a sliding window with some fixed size and stride and apply a transformation. Instead of the inner product, we can do <u>max-pooling</u> (take the max of the window) or <u>average-pooling</u> (take the mean of the window). <u>Global pooling</u> is where the window is the whole input, so we output a single scalar.

## Architecture Examples

**LeNet** Given an input of size $32^2$,

- Convolve with six $5^2$ kernels to 6 @ $28^2$
- Subsample down by half to 6 @ $14^2$
- Convolve with sixteen $5^2$ kernels to 16 @ $10^2$
- Subsample down by half to 16 @ $5^5$
- Fully connect to a 120-wide layer
- Fully connect to an 84-wide layer
- Gaussian connect to a 10-wide output

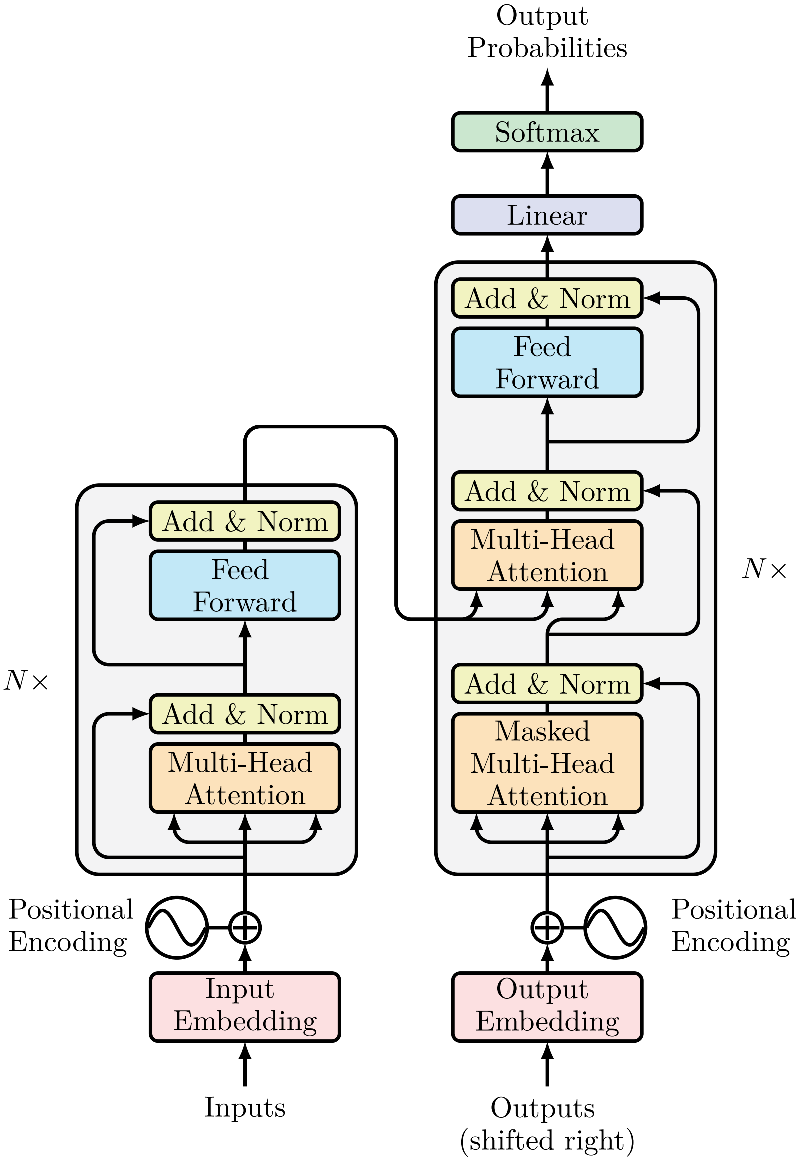
**AlexNet** Given an input of size 3 @ 224 × 224:

- Convolve with 96 kernels to 96 @ 55 × 55

*Lecture 14*
*Mar 5*

## 11 Transformers

TODO: up to slide 11



Our goal is given a sequence of tokens (the prompt) $X = (\mathbf{x}_1, \dots, \mathbf{x}_n)$, to find a sequence $Y = (\mathbf{y}_1, \dots, \mathbf{y}_m)$ such that the maximum likelihood

$$\arg\max_Y p(\mathbf{y}_1, \dots, \mathbf{y}_m \mid \mathbf{x}_1, \dots, \mathbf{x}_n)$$

is achieved. We use an auto-regressive model where we greedily take

$$\arg\max_{\mathbf{y}_k} p(\mathbf{y}_k \mid \mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{y}_1, \dots, \mathbf{y}_{k-1})$$

i.e., one token at a time. Note that $m$ is not pre-defined; we keep generating until we reach the [END] token.

At each step, we input the **embeddings** of the prompt and the partially generated text. The text is converted to tokens, which are the smallest elements the model can understand. Then, the tokens are embedded in a high-dimensional vector space (typically, $d = 512$). The embedding should map similar words to similar locations.

The output of the prompt embedding is $X = [\mathbf{x}_1, \dots, \mathbf{x}_n] \in \mathbb{R}^{n\times d}$ and the auto-regressive outputs $[\mathbf{y}_1, \dots, \mathbf{y}_k] \in \mathbb{R}^{k\times d}$

Since word order matters, we also add a **positional encoding**. We define the matrix $W^p \in \mathbb{R}^{n\times d}$ as

$$W^p_{t,2i} = \sin\left(t/10000^{2i/d}\right), \quad W^p_{t,2i+1} = \cos\left(t/10000^{2i/d}\right), \quad i = 0, \dots, \tfrac{d}{2} - 1$$

This is a fixed part of the model, and we simply add $W^p$ to $X$. The auto-regressive output is also similarly positionally encoded.

These are then sent to **attention layers**.

The Attention function has an input value $V \in \mathbb{R}^{n\times d}$, a key $K \in \mathbb{R}^{n\times d}$, and a query $Q \in \mathbb{R}^{m\times d}$. It outputs an $\mathbb{R}^{m\times d}$ matrix.

Recall the softmax function (eq. 4.b) as applied to vectors:

$$\operatorname{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_i \exp(z_i)}, \dots, \frac{\exp(z_n)}{\sum_i \exp(z_n)}\right]$$

Then, writing $\mathbf{v}_i^\top$, $\mathbf{k}_i^\top$, and $\mathbf{q}_i^\top$ as the rows of $V$, $K$, and $Q$:

$$\begin{aligned}
&\operatorname{Attention}(V, K, Q) \\
&= \operatorname{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \\
&= \begin{bmatrix}
\operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_1\rangle}{\sqrt{d}}\right) & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_2\rangle}{\sqrt{d}}\right) & \cdots & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_n\rangle}{\sqrt{d}}\right) \\
\operatorname{softmax}\left(\frac{\langle \mathbf{q}_2, \mathbf{k}_1\rangle}{\sqrt{d}}\right) & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_2, \mathbf{k}_2\rangle}{\sqrt{d}}\right) & \cdots & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_2, \mathbf{k}_n\rangle}{\sqrt{d}}\right) \\
\vdots & \vdots & \ddots & \vdots \\
\operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_1\rangle}{\sqrt{d}}\right) & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_2\rangle}{\sqrt{d}}\right) & \cdots & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_n\rangle}{\sqrt{d}}\right)
\end{bmatrix} V \\
&= \begin{bmatrix}
\operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_1\rangle}{\sqrt{d}}\right)\mathbf{v}_1^\top + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_2\rangle}{\sqrt{d}}\right)\mathbf{v}_2^\top + \cdots + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_n\rangle}{\sqrt{d}}\right)\mathbf{v}_n^\top \\
\operatorname{softmax}\left(\frac{\langle \mathbf{q}_2, \mathbf{k}_1\rangle}{\sqrt{d}}\right)\mathbf{v}_1^\top + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_2, \mathbf{k}_2\rangle}{\sqrt{d}}\right)\mathbf{v}_2^\top + \cdots + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_2, \mathbf{k}_n\rangle}{\sqrt{d}}\right)\mathbf{v}_n^\top \\
\vdots \\
\operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_1\rangle}{\sqrt{d}}\right)\mathbf{v}_1^\top + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_2\rangle}{\sqrt{d}}\right)\mathbf{v}_2^\top + \cdots + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_n\rangle}{\sqrt{d}}\right)\mathbf{v}_n^\top
\end{bmatrix} \in \mathbb{R}^{m\times d}
\end{aligned}$$

Each output "value" (i.e., row) here is a convex combination of the rows of $V$.

In the self-attention case, $Q = K = V =$ whatever the input is.

So far, there are no learnable parameters. To add learnable parameters, we do a linear layer with each of $V$, $K$, and $Q$. That is, $W_i^q, W_i^k, W_i^v \in \mathbb{R}^{512\times 64}$ can be learnable linear layers such that $\text{Attention}_i = \text{softmax}\left(\frac{QW_i^q(KW_i^k)^\top}{\sqrt{d}}\right)VW_i^v$.

Each of these $i = 1, \dots, h$ triplets is called a head. Typically $h = 8$. A multi-head attention layer concatenates each $\text{Attention}_i$ and sends that through a final learnable linear layer.

A masked attention layer just ignores future tokens $\mathbf{y}_k, \dots, \mathbf{y}_m$ during training.

The **feed forward layers** are just two-layer MLPs with ReLU activation:

$$\max(0, \mathbf{x}^\top W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2$$

where $W_1 \in \mathbb{R}^{d\times 4d}$ and $W_2 \in \mathbb{R}^{4d\times d}$. They also have **residual connections** and **layer normalization**.

**Summary** There are three tunable hyperparameters: layers $N = 6$, output dimensions $d = 512$, and heads $h = 8$.

The cross-attention module has $V = K =$ encoder and $Q =$ decoder. The other attention modules are self-attentive, so $V = K = Q$.

We train by minimizing the log-loss between true next words and predicted next words

$$\min_W \hat{\mathbb{E}}[-\left\langle Y, \log \hat{Y}\right\rangle]$$

where $Y = [\mathbf{y}_1, \dots, \mathbf{y}_l]$ is the one-hot output sequence and $\hat{Y} = [\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_l]$ are the predicted probabilities.

# Chapter 3

# Modern Machine Learning

## 12 Large Language Models

TOOD: up to slide 9

### Generative Pre-Training (GPT-1)

GPT-1 is an open-source 12-layer decoder with 110M parameters. It is pre-trained unsupervised on next-word prediction. Then, fine-tuning is done on task-dependent architecture, i.e., there are specific

### Bidirectional Encoder Representations from Transformers (BERT)

BERT is an encoder-only model. It also has a pre-training phase and fine-tuning phase.

In the pre-training phase, the encoder is given masked sentences and is trained to generate the missing tokens (Masked LM; task A). Training on task A performs better than training on the left-to-right prediction task. The model is also trained on next-sentence prediction (NSP; task B), where it binary classifies whether two sentences follow or are unrelated.

$\text{BERT}_{\text{BASE}}$ has a similar number of parameters (110M) to GPT-1, but performs better. $\text{BERT}_{\text{LARGE}}$ (340M) performs better than both.

RoBERTa (Robustly Optimized BERT Approach) is just a larger BERT model with bigger batches and more data. It was also only trained on the Masked LM objective, but with longer sequences.

Sentence-BERT/RoBERTa trains the similarity task using two encoders (one for each sentence) and saves represented encodings. This saves a lot of inference time.

### GPT-2 through 4

Basically the only thing done is make the model larger.

GPT-2 introduced a new dataset called WebText. The 1.5B-parameter model is around 10× larger than GPT-1, and is trained in the same way. It is about on par with BERT on finetuning tasks. It is also the most recent OpenAI model to be open-sourced. However, it is very good at zero-shot learning. This means we no longer need task-dependent architecture.

GPT-3 is trained exactly the same as GPT and GPT-2, but 100× larger (175B parameters). At around the 100B-parameter level, we start to see emergent properties of in-context learning (zero-/few-shot prompts) and chain-of-thought (either one-shot or "let's do this step-by-step"). However, raw language models do not answer questions or behave in a chat-like way. For example, asking a quesiton to GPT-3 will result in a list of similar questions.

GPT-3.5 (InstructGPT) uses Reinforcement Learning from Human Feedback (RLHF). In RLHF, the agent uses a policy function (LLM) to take actions (outputs) given a state (prompt), and is returned a reward and new state based on the environment (another LLM):

1. Collect demonstration data, and train a supervised policy: train by overfitting GPT-3 to human-written desired outputs (the SFT model).

2. Collect comparison data, and train a reward model: train a new reward model (RM) using human rankings of outputs. We use pair-wise comparison logistic loss

   $$\operatorname{loss}(\theta) = -\,\mathbb{E}_{(x,y_w,y_l)}[\log(\sigma(r_\theta(x,y_w) - r(x,y_l)))]$$

   for a prompt $x$ and preferred output $r_\theta(x,y_w) \gg r_\theta(x,y_l)$. This trains a real-valued function $r_\theta$ so ChatGPT knows how much better $y_w$ is than $y_l$ without the unknown human element.

3. Optimize a policy against the reward model using reinforcement learning: update the SFT model using the RM model using proximal policy optimization (PPO):

   $$\max_\phi \mathbb{E}_{(x,y)}[\underbrace{r_\theta(x,y)}_{\text{RM reward}} - \beta \underbrace{\log\Big(\pi_\phi^{\text{RL}}(y \mid x)/\pi^{\text{SFT}(y|x)}\Big)}_{\text{model is close to SFT}}] + \gamma \underbrace{\mathbb{E}[\log\Big(\pi_\phi^{\text{RL}}(x)\Big)]}_{\text{pretraining loss}}$$

In general, GPT ≪ prompted GPT ≪ SFT ≪ PPO < PPO with pretraining mix. PPO-ptx is the base model for ChatGPT-3.5 and GitHub Copilot.

GPT-4 allows combined multimodal image/text input. The paper says nothing so nobody knows how it works.

# List of Named Results

# Index of Defined Terms