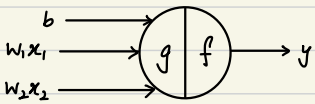
Why Machines Learn

Chp 1: The Perceptron

Theory of cognition — Hebbian learning: Our brains learn because connections between neurons strengthen when one neuron's output is consistently involved in the firing of another ⇝ "Neurons that fire together, wire together"

⇒ Make artificial neurons that reconfigure as they learn!



$$g(x) = \sum_{i=1}^{n} w_i x_i + b$$

$$f(g(x)) = \begin{cases} -1, & \text{if } g(x) \leq 0 \\ 1, & \text{if } g(x) \geq 0 \end{cases}$$

↳ In simple binary classification, we seek w_1, w_2, and b to find a line that separates the 2 groups

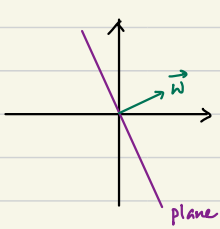
The Mark I perceptron could take a 20x20 pixel image $(x_1, x_2, \ldots, x_{400})$, and determine the written alphabet letter, using learned weights $w_1 \ldots w_{400}$, b.

Chp 2. Vectors

The goal of a perceptron is to find a hyperplane that delineates the space into 2 distinct regions ⇝ equivalent to finding a vector $\vec{w}$ (or a set of weights, that is ORTHOGONAL to the hyperplane!)

We can then take the dot product of any vector $\vec{x}$ with $\vec{w}$:

- If $\vec{x} \cdot \vec{w} > 0$ ⇒ right of plane
- If $\vec{x} \cdot \vec{w} = 0$ ⇒ on the plane
- If $\vec{x} \cdot \vec{w} < 0$ ⇒ left of plane



To take the dot product of 2 matrices:

cols in matrix 1 = # rows in matrix 2.

Hence, given: weights $\vec{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ and features $\vec{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$:

- We first transpose $\vec{w}$ into a 2×1 matrix. $w^T = [w_1, w_2]$
- Then take the dot product: $w^T x$

Perceptron: $y = \begin{cases} -1, & w^T x + b \leq 0 \\ 1, & w^T x + b > 0 \end{cases}$

↓

We can remove the bias term by setting $w_0 = b$, $x_0 = 1$.

Then, $w^T = [w_0, w_1, w_2]$, and $x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$ (where $x_0 = 1$)

↓

$$y = \begin{cases} -1, & w^T x \leq 0 \\ 1, & w^T x > 0 \end{cases}$$

Input data: $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$, each a vector of features

Training algorithm:

1. Initialize weight vector $\vec{w}$ to 0
2. For every data point $\vec{x}$, if $y w^T x \leq 0$:
   ⇒ weight vector is wrong! Update. $w_{new} = w_{old} + yx$
3. If no updates in step #2: terminate
   If there was an update in step #2: repeat whole process

Why is the update step $w_{new} = w_{old} + yx$?

- Proof: Show that eventually, $y w^T x > 0$.
- $y w_{new}^T x = y(w_{old} + yx)^T x$
  $= y\, w_{old}^T x + y^2 x^T x$

Always positive, since:
$y^2 \geq 0$,
$x^T x$ is the dot product of $\vec{x}$ with itself $= \|\vec{x}\|^2 > 0$.

(So after each update, $y w_{new}^T x$ becomes less negative ⇝ w is moving in the right direction!)

Perceptron Convergence Proof:

w · d-dimensional weight vector, initialised to 0

w^* · d-dimensional weight vector that the perceptron must learn
⇒ define as the UNIT VECTOR orthogonal to hyperplane

x · vector representing one input data point

y : output of perceptron, given some input vector x. $y \in \{-1, 1\}$

γ · distance b/w linear hyperplane and closest data point

Proof. Show that if we keep updating w, it converges to w^*

- Normalise input data points so the data pt furthest from origin has magnitude 1
  ⇒ From this, we have $0 < \gamma \leq 1$
- As w comes closer to w^*, $w^T w^*$ becomes bigger
  - ↳ But $w^T w^*$ can also grow if w grows in magnitude → measured by $w^T w$
  - ↳ Algorithm converges if $w^T w^*$ grows faster than $w^T w$
    (means $w^T w^*$ is growing because w is getting aligned with w^*, not just because w is growing!)
- After an update, $w_{new}^T w^* = (w_{old} + yx)^T w^*$
  $= w_{old}^T w^* + yw^{*T}x$ ($yw^{*T}x > 0$, since w^* classifies x correctly.)
  - ↳ $w^{*T}x$ = distance of x from hyperplane
  - ↳ Since distance between CLOSEST data point and hyperplane = γ:
    $yw^{*T}x \geq \gamma$
    $\therefore w_{new}^T w^* \geq w_{old}^T w^* + \gamma$
    $\therefore w^T w^*$ grows by at least γ each update.
- $w_{new}^T w_{new} = (w_{old} + yx)^T (w_{old} + yx)$
  $= (w_{old} + yx)^T (w_{old}) + (w_{old} + yx)^T (yx)$
  $= w_{old}^T w_{old} + yx^T w_{old} + yw_{old}^T x + y^2 x^T x$
  $= w_{old}^T w_{old} + \underbrace{2yw_{old}^T x}_{<0,\ \text{since } yw_{old}^T x < 0 \text{ (classifies wrongly!)}} + \underbrace{y^2 x^T x}_{y^2 = 1,\ \text{and } 0 \leq x^T x \leq 1}$

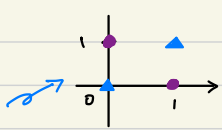
$\therefore w_{new}^T w_{new} = w_{old}^T w_{old} + (\text{-ve quantity}) + (\text{+ve quantity} \leq 1)$

$\therefore w_{new}^T w_{new} \leq w_{old}^T w_{old} + 1$

$\therefore w_{new}^T w_{new}$ grows by at most 1 each update.

- Say our algorithm takes M updates to find w^*
  $\Rightarrow$ Need to prove M is finite.
- Since w is initialized to 0, $w^T w$ starts at 0.
- After M updates.
  - ↳ $w^T w^* \geq M\gamma$ (so $M\gamma \leq w^T w^*$)
  - ↳ $w^T w \leq M$
- Since $M\gamma \leq w^T w^* = \|w\| \|w^*\| \cos\theta$,
  - ↳ $M\gamma \leq \|w\|$, since $\|w^*\| = 1$ and $0 \leq \cos\theta \leq 1$
  - ↳ $M\gamma \leq \sqrt{w^T w}$
  - ↳ $M\gamma \leq \sqrt{M}$
  - ↳ $M^2\gamma^2 \leq M$
  - ↳ $M \leq \frac{1}{\gamma^2}$ (since $0 \leq \gamma \leq 1$, $\frac{1}{\gamma^2}$ is finite
    → M is finite $\Rightarrow$ perceptron always converges !!)

Limitations of the perceptron:



Can't solve the XOR problem!
(No line to separate ▲ from •)

Need multi-layer perceptrons... but how do we train them?
(hint: Backprop!!)

Chp 3: Gradient Descent

Consider the curve $y = x^2$. To get to the bottom of the curve:

- $x_{new} = x_{old} - \eta \cdot \text{gradient}$ (η = small step size)
- $y_{new} = x_{new}^2$

(Progressive steps become smaller, since $\frac{dy}{dx}$ becomes smaller as we reach the bottom of the bowl, where $\frac{dy}{dx} = 0$).

Consider $f(x, y, z) = x^3 + 3y^3 + z^5$:

- Gradient $= \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{bmatrix} = \begin{bmatrix} 2x \\ 9y^2 \\ 5z^4 \end{bmatrix}$ ⇝ gradient is a vector! (points in upwards dir → we must walk in opposite direction)
- At each point, walk a small step:
  $x_{new} = x_{old} - \eta \cdot \begin{bmatrix} 2x \\ 9y^2 \\ 5z^4 \end{bmatrix}$ (just follow the vector!)

Consider: error = e, desired output = d, model output = y.

↳ Want to minimize $E(e^2) = E((d-y)^2)$
$= E((d - w \cdot x)^2)$ (w = weights, x = model inputs)

(call this "J")

∵ J is quadratically related to $\vec{w}$

∴ J is minimized at the bottom of some convex bowl

$\Rightarrow \frac{\partial J}{\partial w} = \frac{\partial E((d-w\cdot x)^2)}{\partial w} = 0$ ... now what if we DIDN'T differentiate the whole thing?

We estimate ∇ using a single data point, $\vec{x}$:

- Assume error in $\vec{x}$ is the mean squared error: minimize $J = \varepsilon^2$
- $J = \varepsilon^2 = (d - w^T x)^2 \Rightarrow \frac{\partial J}{\partial w} = 2(d - w^T x)(-x)$
  $= -2\varepsilon x$

  } Least Mean Squares (LMS) algorithm
- $w_{new} = w_{old} - \eta \cdot \text{gradient}$

$\Rightarrow w_{new} = w_{old} + 2\eta\varepsilon x$, where $\varepsilon = d - w^T x$ for some data point $\vec{x}$.

(Turned out to be a great unbiased estimate for the MSE. By taking many small steps, it would take you to the bottom of the bowl!)

Chp 4 — Probability & Statistics:

Discrete vars: Expected value $= \sum_{i}^{N} x_i \, P(X = x_i)$

Variance $= \sum_{i}^{N} (x_i - E(X))^2 \, P(X = x_i)$

Standard deviation $= \sqrt{Var(X)} = \sigma$

Supervised learning. Given $n \times d$ matrix $X = [x_1, x_2, \ldots, x_n]$ and $\vec{y} = [y_1, y_2, \ldots, y_n]$

- We have some underlying probability distribution $P(X, \vec{y})$ to find
- Bayes optimal classifier: If P is known, then simply find $P(y=0 \mid x)$ vs $P(y=1 \mid x)$
- Say our model estimate for $P(X, y)$ is $P_\theta(D)$:

① Frequentist. Find θ, such that when we sample from $P(\theta)$, we get the largest likelihood of observing D

[Maximum Likelihood Estimation (MLE) — maximizing $P(D \mid \theta)$]

② Bayesian Maximize $P(\theta \mid D)$ — i.e. find the most likely θ, given the data

↳ Implies that θ itself follows a distribution ⇝ we have a prior belief for θ!

[Maximum a posteriori (MAP) — maximizing $P(\theta \mid D)$]

⇒ Starting w/ some prior Θ, we maximize: (same as MLE)

$$MAP = \underset{\theta}{\operatorname{argmax}} \, P(\theta \mid x) = \frac{P(x \mid \theta) \, P(\theta)}{P(x)} \propto P(x \mid \theta) \, P(\theta)$$

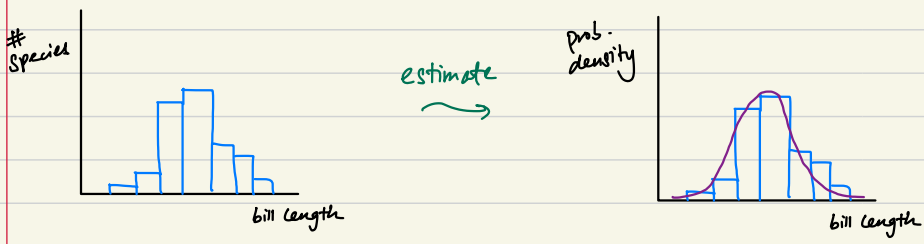
(MAP = maximizing MLE × probability of θ given prior!)

(※) MLE is powerful w/ a lot of sampled data; MAP works best with fewer data.

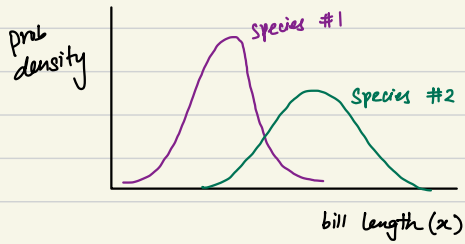
Example: Classify penguins into species $(0, 1, 2)$ based on beak length (x)

- For a given species, estimate the underlying probability dist. of its beak lengths

Observed data $(y=0)$ $\boxed{P(x \mid y=0)}$.



Doing so for each species, we get:



For given x, can calculate:
$P(y=1 \mid x)$,
$P(y=2 \mid x)$

Bayes' Thm: $P(y=1 \mid x) = \frac{P(x \mid y=1) \cdot P(y=1)}{P(x)}$

- $P(y=1)$: prior probability ⇝ est. from dataset: $\frac{\#(y=1)}{\text{total data pts}}$
- $P(x \mid y=1)$: from our normal distribution
- $P(x) = P(x \mid y=1) P(y=1) + P(x \mid y=2) P(y=2)$

⇒ The Bayes optimal classifier establishes a lower bound for the error rate.

(Problem: It gets computationally expensive, very fast!)

- For n features, need to construct a probability dist $\in \mathbb{R}^n$
- Very costly as $n \to \infty$, also requires many more samples.

To make calculating $P(x_1, x_2, x_3, x_4, x_5 \mid y=1)$ easier, (i.e. no correlation b/w variables!)
we assume x_1, x_2, x_3, x_4, x_5 are all sampled from independent distributions

⇒ $P(x_1, x_2, x_3, x_4, x_5 \mid y=1) = P(x_1 \mid y=1) P(x_2 \mid y=1) \cdots P(x_5 \mid y=1)$

⇒ $P(\vec{x} \mid y=1) = \prod_{i=1}^{n} P(x_i \mid y=1)$ ⇝ "naive Bayes' classifier"

n indep. 1D distribution vs one R^n distribution

Key Conclusions:

- ML uses some estimated distribution $P_\theta(x,y)$ to estimate true dist. $P(x,y)$
- We can estimate model parameters θ using either:
  1. Maximum Likelihood Estimation (MLE): Diff θ gives us diff. probability dists
     ⇒ Find θ that maximizes likelihood of observing existing data
  2. Maximum a posteriori estimation (MAP) Start with a prior distribution for θ, find posterior $P_\theta(X,y)$, maximizing $P(X|\theta) P(\theta)$ = MLE × likelihood of θ | prior
- Optimal Bayes attempts to learn the entire joint probability distribution, $P(X \mid y)$

Chp 5 – KNN Algorithms.

Example of a non-parametric model ⇝ no fixed # of parameters!

Curse of dimensionality

- In high-dimensional spaces, the chance of finding a point in any one unit hypercube is vanishingly small
- Number of data samples needs to grow exponentially with # of dimensions

Volume of unit sphere in higher dimension d:

- $V(d) = \frac{\pi^{d/2}}{\frac{d}{2}\Gamma(\frac{d}{2})}$, where $\Gamma(n) = (n-1)!$
- Observe that $\Gamma(\frac{d}{2})$ increases much more quickly than $\pi^{\frac{d}{2}}$
  ⇒ As d increases, volume of unit sphere → 0
  (yet volume of unit hypercube always = 1)

E.g. Unit sphere imposed onto unit hypercube:

- 3D Vertex of cube: $(1,1,1)$ ⇒ dist from origin $= \sqrt{3}$
- 4D: Vertex of cube: $(1,1,1,1)$ ⇒ $d = \sqrt{4}$

..

- 10D: $d = \sqrt{10}$ ⇝ unit sphere becomes vanishingly small!

> Due to the curse of dimensionality, the premise that "nearby points are similar" no longer holds!
> We need to turn to principal components analysis (PCA).

Chp 6 — Principal Components Analysis (PCA):

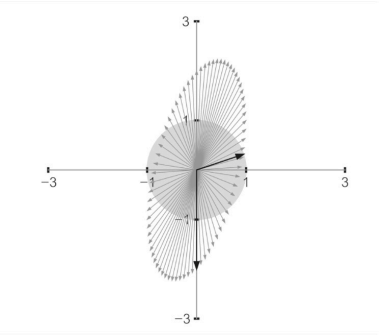
Matrix multiplication $Ax = y$, # cols(A) = # rows(x)

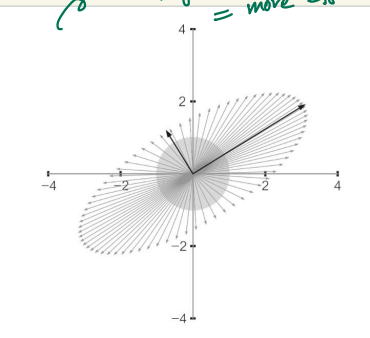
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_1a_{11} + x_2a_{12} + x_3a_{13} \\ x_1a_{21} + x_2a_{22} + x_3a_{23} \end{bmatrix}$$

Eigenvectors: Only makes sense for square matrices
↝ ensures the dimensionality of the vector stays the same!

- "Eigen" = characteristic → special vectors inherent to a matrix that retains its orientation after the transformation
- Eigenvector Scaling factor for a given eigenvector
  ∴ $A\vec{x} = \lambda\vec{x}$

larger diff in variance along major vs minor axes in ellipse = more effective PCA.





For any 2x2 matrix, a circle is transformed into an ellipse.

For square symmetric matrices, we get orthogonal eigenvectors! (major & minor axes of ellipse)

Project into 10000 dimensions..

- Set of unit vectors: circle-equivalent → ellipse-equivalent
- 10,000 eigenvectors w/ 10,000 eigenvalues ↝ all orthogonal!!

Covariance

Start with: $X = \begin{bmatrix} h_1 & w_1 \\ h_2 & w_2 \\ h_3 & w_3 \end{bmatrix}$ 3 data points. (height, weight)

Get their deviations by subtracting mean: $\begin{bmatrix} h_1 - E(h) & w_1 - E(w) \\ h_2 - E(h) & w_2 - E(w) \\ h_3 - E(h) & w_3 - E(w) \end{bmatrix}$

Take dot product of X with X^T. (using the mean-corrected values)

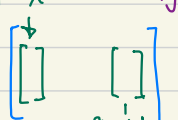
$$X^T \cdot X = \begin{bmatrix} h_1 & h_2 & h_3 \\ w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} h_1 & w_1 \\ h_2 & w_2 \\ h_3 & w_3 \end{bmatrix} = \begin{bmatrix} h_1^2 + h_2^2 + h_3^2 & h_1w_1 + h_2w_2 + h_3w_3 \\ h_1w_1 + h_2w_2 + h_3w_3 & w_1^2 + w_2^2 + w_3^2 \end{bmatrix}$$

Note that diagonal terms = variance of individual features.
off-diagonal terms = covariance → linear association b/w features!
⇒ hence we always get a square, symmetric matrix!

> The eigenvectors of a covariance matrix are the principal components of original matrix X

PCA workflow: Data matrix X → calculate mean-corrected values
→ find covariance matrix → find eigenvectors & eigenvalues
→ select eigenvectors w/ largest λ (this is where most variance is!)
→ transformed dataset = $W_{reduced}^T \cdot X$

largest λ only keeping eigenvalues w/ large λ



n×n matrix of n eigenvectors (cols)

e.g. n×3, if we keep 3 eigenvectors. Then we project data into 3D!

⇒ can visualize along these axes + perform easier k-means clustering.

Chp 7 — The Kernel:

Goal: Find the optimal hyperplane separating two groups of data

- Represented by $\vec{w}$ (vector orthogonal to plane) and bias b (dist. from origin)
- Want to maximize the margin — i.e dist btw hyperplane & closest data points from each class

The points from each class closest to the hyperplane = support vectors

- For classification, we need:

$$\left.\begin{array}{l} w \cdot x_i + b > 0 \quad (\text{for } y_i = +1) \\ w \cdot x_i + b < 0 \quad (\text{for } y_i = -1) \end{array}\right\} (w \cdot x_i + b)(y_i) > 0$$

- We then want to maximize the margin.
  - ↳ Distance of x_i to hyperplane = $\frac{|w \cdot x_i + b|}{\|w\|}$
    - ↳ fix minimal dist of support vec to hyperplane $= 1$
  - ↳ For all points, $y_i(w \cdot x_i + b) \geq 1$
  - ↳ Support vectors: $w \cdot x + b = +1$ and $w \cdot x + b = -1$
    - → Margin: $\frac{2}{\|w\|}$
  - ↳ Maximizing $\frac{2}{\|w\|}$ $\Leftrightarrow$ minimizing $\frac{1}{2}\|w\|^2$ (simplest for differentiation!)

Result We get $\vec{w} = \sum_i \alpha_i y_i x_i$, where:

- Each α_i = specific to a data pt and its label (x_i, y_i)
  - ↳ these are the Lagrange multipliers!
  - ↳ Depend ONLY on the mutual dot product of vectors (i.e. the support vectors) representing the data samples
- Label for point $\vec{u}$ = $\begin{cases} +1, & \text{if } w \cdot u + b \geq 0 \Rightarrow \sum_i \alpha_i y_i x_i \cdot u_i + b \geq 0 \\ -1, & \text{if } w \cdot u + b < 0 \Rightarrow \sum_i \alpha_i y_i x_i \cdot u + b < 0 \end{cases}$
- It turns out only the support vectors matter. $\alpha = 0$ for points not on the margin.
- Even if we have 10,000 datapoints, if only 10 lie on the margin:
  - ↳ optimal hyperplane depends only on dot product of support vectors w/ each other
  - ↳ Decision rule: dot product of u with support vectors

(What if we can project linearly inseperable data into a higher-dimensional space, where a linearly separable hyperplane exists?)

The Kernel Trick.

- Need an algorithm to
  1. Create new features to map data → higher dimensional space
  2. Find separating hyperplane w/o computing dot products in high dimensions
- Consider the mapping $\phi(x_j)$ $\quad [x_1 \; x_2] \rightarrow [x_1^2 \; x_2^2 \; \sqrt{2}x_1x_2]$ $\quad (2D \rightarrow 3D)$
  - ↳ To find the hyperplane, we need $\phi(x_i) \cdot \phi(x_j)$ for all $i, j \in \{1, \ldots, N\}$
    (Can we find some function $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$?
  - ↳ E.g. $\phi(x_i) \cdot \phi(x_j) = a_1^2 b_1^2 + a_2^2 b_2^2 + 2a_1a_2b_1b_2$
    $\Rightarrow$ We find $K(x_i, x_j) = (x_i \cdot x_j)^2 = \phi(x_i) \cdot \phi(x_j)$
    (We can calculate 3D dot products while staying in 2D!)
- K is the kernel function! Allows us to compute higher-dim dot products.

Example of Kernel Function: $K(x,y) = (c + x \cdot y)^d$

- Works for all constants c and d ⇝ can project into v. high dimensions!
- # of dimensions : $\binom{n+d}{d} = \frac{(n+d)!}{n!\,d!}$

Hilbert spaces & infinite-dimensional vectors

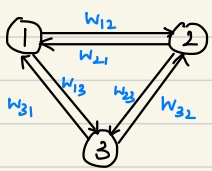
- The radial basis function (RBF) kernel: $K(a,b) = \exp(-\gamma \|a-b\|^2)$, $\gamma > 0$
- RBF kernel can ALWAYS find a linearly separable hyperplane in some infinite-dimension space ⇝ i.e. can find any decision boundary when mapped back to the lower dim! It is a universal function approximator.

Summary of SVM:

1. Take datasets that are linearly inseperable in their low-dim space
2. Project into high enough dim
3. Use kernel trick to find optimal seperating hyperplane
4. Project back down to find optimal decision boundary!

Chp 8 — Hopfield Networks

Consider a bidirectional network:



E.g. Neuron 2 · $\underbrace{w_{12} y_1}_{1 \to 2} + \underbrace{w_{32} y_3}_{3 \to 2}$

$$y_i = \begin{cases} +1, & \text{if } \sum_{j, i \neq j} w_{ij} y_j > 0 \\ -1, & \text{if } \sum_{j, i \neq j} w_{ij} y_j \leq 0 \end{cases}$$

Energy of network = $-\frac{1}{2} \sum_i \sum_{j, i \neq j} w_{ij} y_i y_j$

↝ Just like Ising's model of ferromagnetism (magnetic states dynamically influenced by neighbors: models dynamically arrive at lowest-energy state

(*) On the condition that weights are symmetric.

A connection to associative ("reconstructing") memories.

- Set the weights of a network such that a given pattern of outputs represents a stable minimum ↝ this is the memory!
- Now perturb the output slightly, and force the model to output the corrupted pattern
- Neurons flip and dynamically reach the stable minimum!

① Learning the appropriate weights.

- Weight matrix · $\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} 0 & w_{12} & w_{13} \\ w_{21} & 0 & w_{23} \\ w_{31} & w_{32} & 0 \end{bmatrix}$ (symmetrical!)
- To store $y = [-1, 1, -1]$, we want to reinforce weights of neurons firing together
  ↳ E.g. $w_{12} = w_{21} = y_1 \cdot y_2 = -1$ (since they fire oppositely)
- It turns out we need the outer product of vectors.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \times [y_1 \; y_2 \; y_3] - I = \begin{bmatrix} y_1 \times [y_1 \; y_2 \; y_3] \\ y_2 \times [y_1 \; y_2 \; y_3] \\ y_3 \times [y_1 \; y_2 \; y_3] \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$W = y^T y - I$$

For any neuron i, we have output y_i and weights w_{ij}:

$\Rightarrow \sum_{i,j\neq i} w_{ij} \cdot y_j = \sum_{i,j\neq i} y_i \cdot y_j^2$ (since $w_{ij} = y_i \cdot y_j$)

$\Rightarrow$ always same sign as y_i! neuron will never flip

To store an image y_1, we set: $W_1 = y_1 y_1^T - I$ (stable state!)

For two images, set: $W = \frac{1}{2}(y_1 y_1^T + y_2 y_2^T) - I$

For n images, set: $W = \frac{1}{n}\sum_{i=1}^{n} y_i y_i^T$ → can store $n = 0.14 \times$ (# of neurons)

Then, to "retrieve" an image:

1. Calculate energy of perturbed network
2. Calculate output of a random neuron; flip if necessary.
3. Calculate new energy:
   - If $\Delta E > \varepsilon$, keep flipping neurons.
   - If $\Delta E < \varepsilon$, stop! (v. small $\Delta E \Leftrightarrow$ near local minimum!)

Convergence Proof of Hopfield Network:

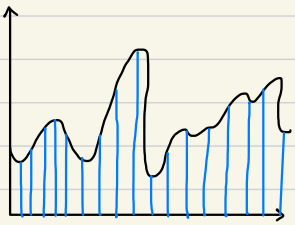
- Consider a network of bipolar neurons (+1 or −1), connected with symmetrical weights
- Given vector $y = [y_1, .., y_n]$: $W = y^T y - I$
- Update rule: $y_{i\,new} = y_{i\,old} \times \sum_{i,j\neq i} w_{ij} y_j$
- Energy of network: $-\frac{1}{2}\sum_i \sum_{i,j\neq i} w_{ij} y_i \cdot y_j$
  - ↳ E.g Network w/ 3 neurons $E = -\frac{1}{2}[2w_{12}y_1y_2 + 2w_{13}y_1y_3 + 2w_{23}y_2y_3]$

    (focusing on neuron 1) $= -\frac{1}{2}[2y_1(w_{12}y_2 + w_{13}y_3) + 2w_{23}y_2y_3]$
  - ↳ $E = -\frac{1}{2}[\underbrace{2y_1 \sum_{j\neq 1} w_{ij} \cdot y_j}_{\text{specific to } y_1} + \underbrace{\sum_{i,i\neq 1}\sum_{j,j\neq 1,j\neq i} w_{ij} y_i y_j}_{\text{all neurons other than } y_1}]$ — this doesn't change even if y_1 flips!
- If y_1 flips, $\Delta E = -\frac{1}{2}[2y_{1\,new}\sum_{j\neq 1} w_{ij} \cdot y_j] + \frac{1}{2}[2y_{1\,old}\sum_{j\neq 1} w_{ij} \cdot y_i]$

  $= -(y_{1\,new} - y_{1\,old}) \cdot \sum_{j\neq 1} w_{ij} \cdot y_j$ — flip = opp. sign w/ $y_{1\,old}$
- If $y_{1\,old} = +1 \cdot \Delta E = -(-1-1) \cdot (neg) = (+2)(neg) = neg$
- If $y_{1\,old} = -1 \cdot \Delta E = -(1-(-1)) \cdot (pos) = (-2)(pos) = neg$
- All neuron flips decrease the energy of the network! Since there is a finite # of possible states, the network must eventually converge to min E.

Chp 9 – Universal Approximation Theorem:

Given any function:



Can be approximated using rectangles!
What if we represent each rectangle using a neural unit (e.g. 2 rectangles – height & width)
⇒ approximates the function!

For a network w/ one hidden layer:

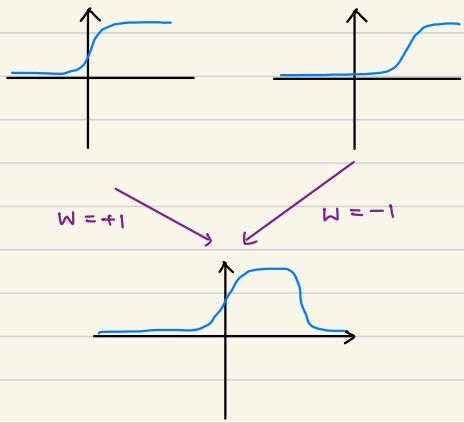
- Input vector: $\vec{x}$, d dimensions
- n neurons in hidden layer
- Weight matrix W of hidden layer: $d \times n$ matrix
- Output y

$$\Rightarrow y = f(x) = \sum_{i=1}^{n} \alpha_i \cdot \sigma(w_i^T x + b_i)$$

α_i: weight of hidden neuron; σ: activation func; $\sigma(w_i^T x + b_i)$: output of hidden neuron

} want to prove that this can approximate any func!

Example of making 'rectangles'.



⇒ By combining many sigmoidal neurons, we can make rectangles that approximate any function!

Functions as vectors:

- Consider representing $f(x) = \sin(x)$ as $[0, \frac{1}{2}, \frac{\sqrt{2}}{2}, \frac{\sqrt{3}}{2}, 1, \ldots]$ for $x = 0$ to 2π.
- We can extend this to infinite x-values from $x = -\infty$ to ∞ ⇝ vector ∈ infinite dims!
- NN: input vec $\xrightarrow{\text{transform by weight matrix}}$ hidden $\xrightarrow{\sigma, W}$ output

Chp 10 – Backpropagation:

The goal is to take training errors → express as a function of layer weights → perform gradient descent. (But how to solve the problem of local minima?)

"Delta Rule" for weights and biases (linear regression):

- For a given dataset (x, y), initialize $w = 0$, $b = 0$.
- Calculate neuron output: $\hat{y} = wx + b$
- Calculate error. $e = y - \hat{y}$
- Calculate square loss: $e^2 = (y - wx - b)^2 = L$
- Gradient of loss function: $\Delta L = \left[\frac{\partial L}{\partial w} \quad \frac{\partial L}{\partial b}\right]$
  - ↳ $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial w}$ (chain rule: $L = e^2$, $e = y - wx - b$)
    $= (2e)(-x)$
    $= -2x(y - wx - b)$
  - ↳ $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial b}$
    $= (2e)(-1)$
    $= -2(y - wx - b)$

  $\Delta L = \begin{bmatrix} -2ex \\ -2e \end{bmatrix}$

  negative, since gradient points upwards!
- The update rule: $\Delta w = -\frac{\partial L}{\partial w}$, $\Delta b = -\frac{\partial L}{\partial b}$
- In practice, we do: $w = w - \alpha \cdot \frac{\partial L}{\partial w}$, $b = b - \alpha \cdot \frac{\partial L}{\partial b}$ (α = learning rate)

Extend to 2D: w_1, w_2, b

- $\hat{y} = w_1 x_1 + w_2 x_2 + b \Rightarrow e = y - w_1 x_1 - w_2 x_2 - b$, $L = e^2$
- $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial w_1} = (2e)(-x_1)$; $\frac{\partial L}{\partial b} = (2e)(-1) = -2e$

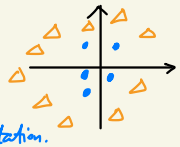
$$\Delta L = \begin{bmatrix} \partial L/\partial w_1 \\ \partial L/\partial w_2 \\ \partial L/\partial b \end{bmatrix} = \begin{bmatrix} -2x_1 \cdot (\hat{y} - y) \\ -2x_2 \cdot (\hat{y} - y) \\ -2 \cdot (\hat{y} - y) \end{bmatrix} \begin{matrix} \rightarrow w_1 = w_1 + \alpha\frac{\partial L}{\partial w_1} \\ \cdots \\ \rightarrow b = b + \alpha\frac{\partial L}{\partial b} \end{matrix}$$

Some cases (e.g. XOR) require multiple layers!

Layer 1: 2 neurons → find 2 lines
Layer 2: Weighted sum of lines

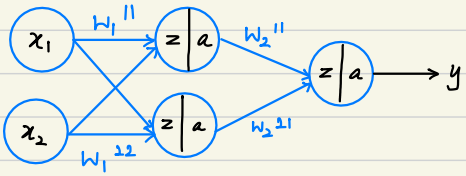
SVM · hand-design features, cannot use x/y

NN can use x/y as input & model learns representation.



> The beauty of backprop is that hidden neurons can learn optimal representations, such that we don't need to hand-design features!
>
> Even in SVM / kernels, we must decide which features (e.g. xy-coords / r, θ) to input.

Developing a multi-layer network



Hidden neuron 1 $z_1^1 = w_1^{11} x_1 + w_1^{21} x_2 + b_1^1$

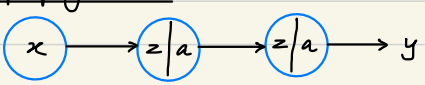
$a_1^1 = \sigma(z_1^1)$

Output neuron $z_2^1 = w_2^{11} a_1^1 + w_2^{21} a_1^2 + b_2^1$

$y = \sigma(z_2^1)$

We need to calculate: $\frac{\partial L}{\partial w_1^{11}}, \frac{\partial L}{\partial w_1^{21}}, \ldots, \frac{\partial L}{b_1^1}, \frac{\partial L}{b_2^1} \ldots$ ⇝ too complicated !!

Backpropagation.



$z_1 = w_1 x + b$

$a_1 = \sigma(z_1)$

$z_2 = w_2 a_1 + b$

$\hat{y} = \sigma(z_2)$

$e = y - \hat{y}$

Loss: $L = e^2$ ⇝ need $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{b_1}, \frac{\partial L}{b_2}$

$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$ (recall $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$

$= (2e)(-1)(\hat{y})(1-\hat{y})(a_1) = -2e\, a_1 (\hat{y})(1-\hat{y})$

$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} = -2e(\hat{y})(1-\hat{y})$

Then, $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$

$= (2e)(-1)(\hat{y})(1-\hat{y})(w_2)(a_1)(1-a_1)(x)$

all values are already calculated in the forward pass!

Generalization of Backprop:

- $\vec{x} = [x_1\ x_2]$, 3 neurons in hidden layer: $W_1 \in \mathbb{R}^{2\times 3} = \begin{bmatrix} w_1^{11} & w_1^{12} & w_1^{13} \\ w_1^{21} & w_1^{22} & w_1^{23} \end{bmatrix}$
- Outputs of hidden layer $z_1 = w_1^{11} x_1 + w_1^{21} x_2 + b_1$; $a_1 = \sigma(z_1)$
- Final layer: $z_4 = w_4^{11} a_3^1 + w_4^{21} a_3^2 + w_4^{31} a_3^3 + b_4 = w_4^T a_3 + b_4$, $\hat{y} = \sigma(z_4)$
- E.g. To find $\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial e} \cdot \frac{\partial e}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_4} \cdot \frac{\partial z_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3}$

$= (2e)(-1)(\hat{y})(1-\hat{y})(w_4^T)(a_3)(1-a_3)(a_2)$

and we have all of these from the forward pass, that's it! ☺

Chp 11 – Vision & Convolutional NNs:

Visual field: region in front of us that the eyes are sensitive to, when focused

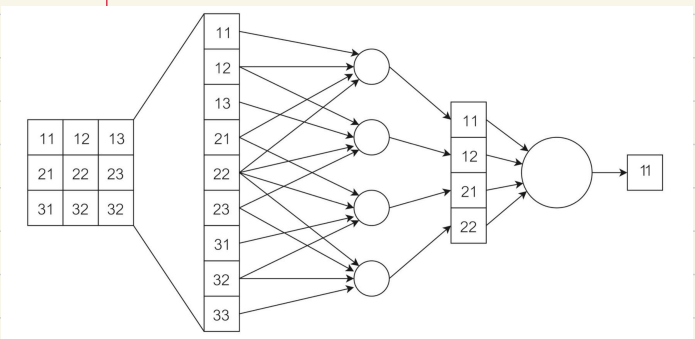
Receptive field: portion of the visual field that triggers a single neuron
↳ Neurons fire if there is an appropriate stimulus in their receptive field
⇒ Retinal ganglion cells (RGCs): smallest receptive field ⇝ the first layer of neurons receiving input from retina.

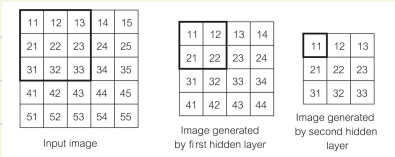
RGCs → simple cells → complex cells

- E.g If 4 RGCs vertically arranged fires simultaneously → activates simple cell → vertical edge detected!
- Many simple cells each detect edges in diff. receptive fields → complex cell fires if ANY simple cell fires → invariance! activation regardless of position of edge.

Convolutions:

- Defined by a kernel filter ($k \times k$) and stride s ⇝ output size: $\lfloor \frac{i-k}{s} + 1 \rfloor$
  ↳ e.g $\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$ → vertical edge detector!
- We assign one neuron per position that the kernel can take in the image
  ↳ the "receptive field" of the neuron = position of kernel!





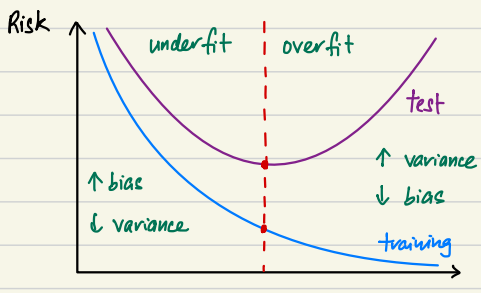
typically, no overlapping pixels (i.e. stride length = kernel size)

Pooling: Place a filter over some part of the original image, and it spits out a single value (e.g. max pooling → largest val in matrix)

↳ Reduces # of pixels and hence # of neurons in the next layer

↳ Increases the receptive field of downstream neurons, helping w/ invariance.

Chp 12 – Terra Incognita:



too simple!

Bias: systematic errors due to simplifying assumptions

Variance: how much predictions would change if trained on diff data subset. (too flexible!)

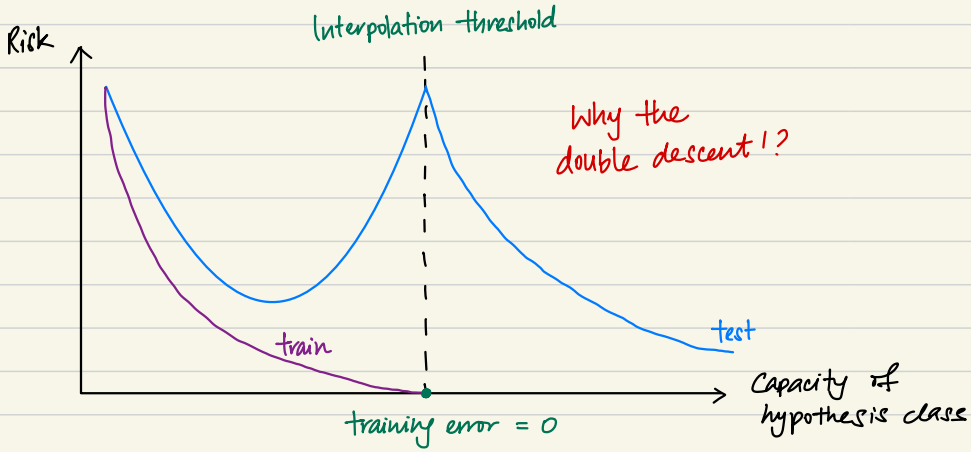
Yet this is not what we observe! Large, complex models, even datasets "shattered" with noise, DON'T overfit the data!!

Training ML models:

1. Feed-forward vs Recurrent: Do outputs of further layers influence earlier layers? (e.g. LSTM feedback connections → rmb prev inputs)
2. Loss function → used by backprop for gradient descent
3. Regularizer: penalty on model complexity in loss function
4. Dropout, activation functions

Self-supervised learning (e.g. LLM):

- Read large corpus of text → mask a random word → predict word
- Cost function = "how wrong" our prediction is → update params
- Masked auto-encoder (MAE): self-supervision for images → mask 80% of image + provide latent representation ⇒ can reconstruct image!
  ↳ NO need to label data



Intriguing questions:

- Why are deep NNs so good at avoiding local minima?
- How do deep NNs generalize so well?
- Grokking: training beyond 0% training error → after the model "memorizes" all the training data, it begins internalizing it, achieving a double descent in test errors!
  ↳ e.g. OpenAI modulo training. After interpolation, the model learned to represent numbers in a circle!