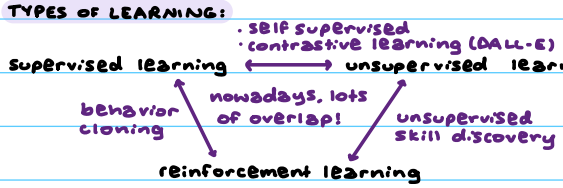
6.390 NOTES - SHEN SHEN

TYPES OF LEARNING:



SUPERVISED LEARNING:

REGRESSION: predicting a continuous number

ex: predicting city's energy usage

TRAINING DATA: $D_{train} := \{(x^1, y^1), \dots (x^4, y^4)\}$

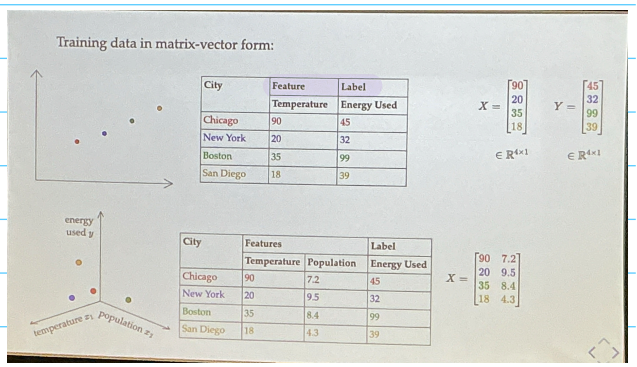
feature vector (x) — label (y)

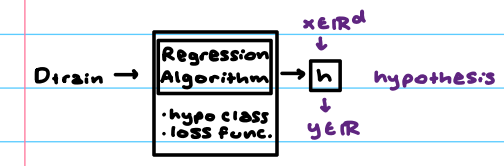
$x^1 = \begin{bmatrix} x_1^1 \\ x_2^1 \\ \vdots \\ x_d^1 \end{bmatrix} \in \mathbb{R}^d$ — $y^1 \in \mathbb{R}$ ← always scalar

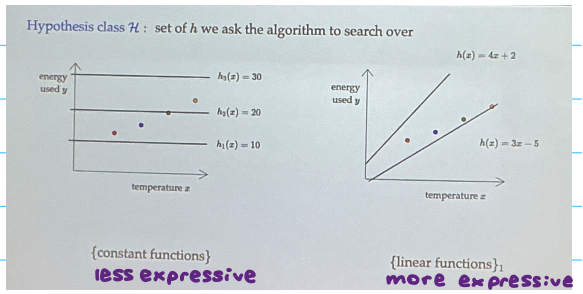
· input variable (what model knows) — · output / target var (what model is trying to predict)

n data pts, each w/ d dimensional features & scalar label

nxd X, nx1 y







HYPOTHESIS: function that can take in new x & output new y

- want to make good predictions

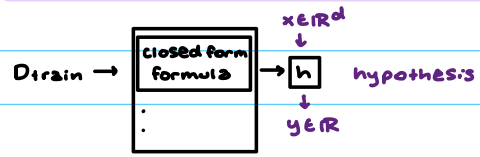
- **HYPOTHESIS CLASS H:** set of h we ask algorithm to search over

LOSS: $L(h(x^i), y^i)$ how good hypothesis is compared to actual outcome

TRAINING ERROR: $\mathcal{E}_{train}(h) = \frac{1}{n}\sum_{i=1}^{n} L(h(x^i), y^i)$ ← average loss on training data

TEST ERROR: $\mathcal{E}_{test}(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} L(h(x^i), y^i)$ ← average loss on n' unseen test data pts

ORDINARY LEAST SQUARES REGRESSION:



LINEAR HYPOTHESIS CLASS: $h(x, \theta) = [\theta_1, \theta_2, \dots, \theta_d] \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} = \theta^T x$ (params, features)

SQUARED LOSS FUNC: $L(h(x^i), y^i) = (\theta^T x^i - y^i)^2$

DERIVE OLS:

1) write training error $J(\theta)$ in scalar form

2) rearrange into matrix-vector form $J(\theta) = \frac{1}{n}(X\theta - Y)^T(X\theta - Y)$ ← objective

3) set gradient $\nabla_\theta J$ to zero

4) solve optimal params $\theta^* = (X^TX)^{-1}X^TY$ ← closed form soln; when well defined, θ^* is unique minimizer of $J(\theta)$

1. Write out training error:

e.g.:

City	Features		Label
	Temp (°F)	Pop (M)	Energy (kWh)
Chicago	90	7.2	45
New York	20	9.5	32
Boston	35	8.4	99
San Diego	18	4.3	39

avg. training data

$$J(\theta) = \frac{1}{4}[(\theta_1 \cdot 90 + \theta_2 \cdot 7.2 - 45)^2 + (\theta_1 \cdot 20 + \theta_2 \cdot 9.5 - 32)^2 + (\theta_1 \cdot 35 + \theta_2 \cdot 8.4 - 99)^2 + (\theta_1 \cdot 18 + \theta_2 \cdot 4.3 - 39)^2]$$

$\theta = \begin{bmatrix}\theta_1\\ \theta_2\end{bmatrix} \in \mathbb{R}^{2\times 1}$

- Q: What kind of function is $J(\theta)$?
  quadratic (2nd order)
- Q: what J(θ) look like?
  typically a bowl

2. Rearrange training error into matrix-vector form

City	Features		Label
	Temp (°F)	Pop (M)	Energy (kWh)
Chicago	90	7.2	45
New York	20	9.5	32
Boston	35	8.4	99
San Diego	18	4.3	39

$$X = \begin{bmatrix}90 & 7.2\\ 20 & 9.5\\ 35 & 8.4\\ 18 & 4.3\end{bmatrix} \quad Y = \begin{bmatrix}45\\ 32\\ 99\\ 39\end{bmatrix} \quad \theta = \begin{bmatrix}\theta_1\\ \theta_2\end{bmatrix}$$

$\in \mathbb{R}^{4\times 2}$ $\in \mathbb{R}^{4\times 1}$ $\in \mathbb{R}^{2\times 1}$

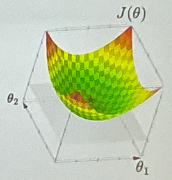
$$J(\theta) = \frac{1}{4}[(\theta_1 \cdot 90 + \theta_2 \cdot 7.2 - 45)^2 + (\theta_1 \cdot 20 + \theta_2 \cdot 9.5 - 32)^2 + (\theta_1 \cdot 35 + \theta_2 \cdot 8.4 - 99)^2 + (\theta_1 \cdot 18 + \theta_2 \cdot 4.3 - 39)^2]$$

$$\Downarrow$$

$$J(\theta) = \frac{1}{n}(X\theta - Y)^T(X\theta - Y)$$

3. Get the gradient $\nabla_\theta J \overset{\text{set}}{=} 0$

$$\nabla_\theta J = \begin{bmatrix}\partial J/\partial\theta_1\\ \vdots\\ \partial J/\partial\theta_d\end{bmatrix} = \frac{2}{n}(X^TX\theta - X^TY)$$



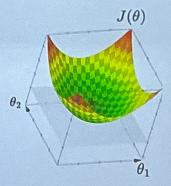
4. Set the gradient $\nabla_\theta J \overset{\text{set}}{=} 0$

$$\Rightarrow \quad \theta^* = (X^TX)^{-1}X^TY$$

- *Typically, $J(\theta)$ "curves up"*
- *The minimizer of $J(\theta)$ necessarily has a gradient zero.*

The beauty of

$$\theta^* = (X^TX)^{-1}X^TY$$



- When θ^* is well defined, it's the unique minimizer of $J(\theta)$
- Closed-form solution, does not feel like "training"
- Very rare case where we get a general and clean solution with nice theoretical guarantee.

2/9 **REGULARIZATION & CROSS-VALIDATION:**

Recall

Let

$$X = \begin{bmatrix}— x^{(1)T} —\\ \vdots\\ — x^{(n)T} —\end{bmatrix} = \begin{bmatrix}x_1^{(1)} & \cdots & x_d^{(1)}\\ \vdots & \ddots & \vdots\\ x_1^{(n)} & \cdots & x_d^{(n)}\end{bmatrix} \quad Y = \begin{bmatrix}y^{(1)}\\ \vdots\\ y^{(n)}\end{bmatrix} \quad \theta = \begin{bmatrix}\theta_1\\ \vdots\\ \theta_d\end{bmatrix}$$

$\in \mathbb{R}^{n\times d}$ $\in \mathbb{R}^{n\times 1}$ $\in \mathbb{R}^{d\times 1}$

Then $J(\theta) = \frac{1}{n}(X\theta - Y)^T(X\theta - Y) \in \mathbb{R}^{1\times 1}$

& $\theta^* = (X^TX)^{-1}X^TY \in \mathbb{R}^{d\times 1}$ ← only works if X & Y are vectors

$\theta^* = [2.00, 3.00]^T \quad J(\theta^*) = 0.00$

Feature (d=2) and Label

$(\begin{bmatrix}1\\0\end{bmatrix}, 2)$ $(\begin{bmatrix}0\\1\end{bmatrix}, 3)$ $(\begin{bmatrix}1\\1\end{bmatrix}, 5)$

hypothesis $h = 2x_1 + 3x_2$

Training error $J(\theta_1, \theta_2)$

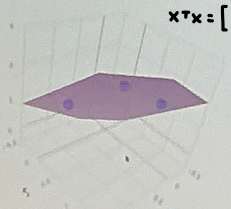
$J(\theta) = \frac{1}{3}[(\theta_1-2)^2 + (\theta_2-3)^2 + (\theta_1+\theta_2-5)^2]$

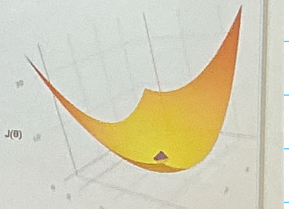
$X = \begin{bmatrix}1 & 0\\ 0 & 1\\ 1 & 1\end{bmatrix}$ $Y = \begin{bmatrix}2\\ 3\\ 5\end{bmatrix}$

$X^TX = \begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}$ $X^TY = \begin{bmatrix}7\\ 8\end{bmatrix}$

$\theta^* = \begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}^{-1}\begin{bmatrix}7\\ 8\end{bmatrix}$

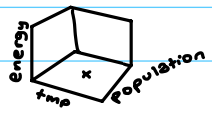
$= \begin{bmatrix}2\\ 3\end{bmatrix}$





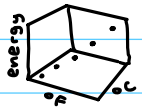
PROBLEMS: closed-form formula UNDEFINED

a) n > d (less data than feature we're trying to fit):



- ex: genomics, NLP

b) linearly-dependent features



- ex: temp °F/°C, age/birth.yr

closed form formula: $\theta^* = (X^TX)^{-1}X^TY$ undefined

optimal soln: ∞ many optimal θ^* → not enough info to pin down a unique soln.

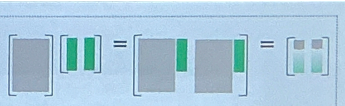
mathematically,

(X^TX) is singular

⇕

X is not full column rank

MM 2



Ax and Ay are linear combinations of columns of A.

$$\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}\begin{bmatrix}x_1 & y_1\\ x_2 & y_2\end{bmatrix} = A[x \quad y] = [Ax \quad Ay]$$

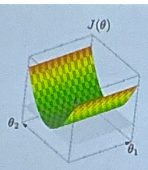
demo (a) $n < d$: 1 sample, 2 features

$$X^TX = \begin{bmatrix}2 & 3\end{bmatrix}\begin{bmatrix}2\\ 3\end{bmatrix} = \begin{bmatrix}4 & 6\\ 6 & 9\end{bmatrix}$$

$\begin{bmatrix}2\\3\end{bmatrix}\begin{bmatrix}2 & 3\end{bmatrix}$

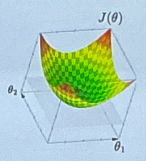
demo (b) Collinear: $x_2 = 1.5 \cdot x_1$

$$X^TX = \begin{bmatrix}2 & 4 & 6\\ 3 & 6 & 9\end{bmatrix}\begin{bmatrix}2 & 3\\ 4 & 6\\ 6 & 9\end{bmatrix} = \begin{bmatrix}56 & 84\\ 84 & 126\end{bmatrix}$$



When X is not full column rank

- $J(\theta)$ has a "flat" bottom
- This formula is not well-defined
- Infinitely many optimal hyperplanes



Typically, X is full column rank

- $J(\theta)$ "curves up" everywhere
- $\theta^* = (X^TX)^{-1}X^TY$
- unique optimal hyperplane

formula itself isn't wrong, data is trouble-making

XTX becoming more invertible

min λ of XTX > 0 (we want all positive semidefinites)

sits in middle (very sensitive info)

ex: assume n=1 & y=1, then $\theta^* = \frac{1}{x}$

if data is (x, 1) = (0.002, 1) → θ^* = 500

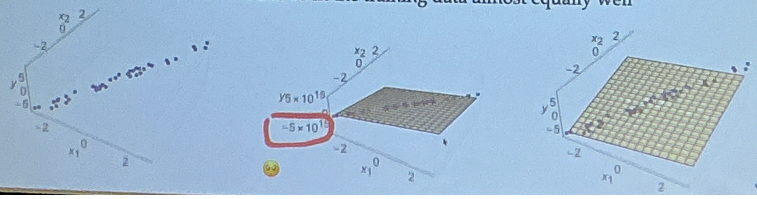
if data is (x, 1) = (-0.002, 1) → θ^* = -5000

when X^TX is *almost* singular

technically, $\theta^* = (X^TX)^{-1}X^TY$ exists and gives the unique optimal hyperplane

practically,

- θ^* tends to have huge magnitude
- θ^* tends to be very sensitive to the small changes in the data
- lots of other θs fit the training data almost equally well



MSE on training data | penalty on param magnitude

RIDGE REGRESSION: $J_{ridge}(\theta) = \frac{1}{n}(X\theta - Y)^T(X\theta - Y) + \lambda\|\theta\|^2$ — $\lambda > 0$ controls how heavily we penalize magnitude relative to MSE

- many θ give similar loss, but some have huge magnitude → unstable!
  - small change in x → wildly different prediction
- IDEA: penalize large θ in our obj (explicit regularization)

$\theta^*_{ridge} = (X^TX + n\lambda I)^{-1}X^TY$

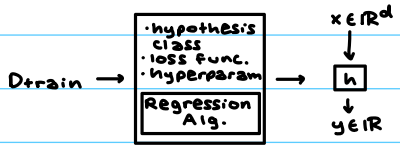
for $\lambda > 0$: $X^TX + n\lambda I$ is always invertible, so θ^*_{ridge} always exists and is unique

ex: $\lambda = 0$ → no penalty – reduces to OLS

$\lambda = 1000$ → huge penalty – forces $\theta \approx 0$

$\lambda = -100$ → rewards large θ – counter-productive ← $\lambda > 0$ required

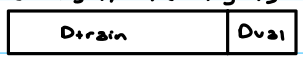
λ is **hyperparameter**:



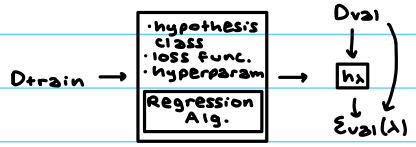
- affects learning outcome, not learned by algo.

we need to choose hyperparams like λ:

- can't use training error (would always pick $\lambda = 0$)
- can't use test error (we don't have test data)
- hold-out some training data $\{(x^1, y^1), \ldots, (x^n, y^n)\}$



- use Dval to evaluate how good a hyperparam is



for each $\lambda \in \{0.1, 1, 10\}$:

- train on Dtrain with λ
- compute $\varepsilon_{val}(\lambda)$ on Dval

return $\arg\min_\lambda \varepsilon_{val}(\lambda)$ → in this example, compare $\varepsilon_{val}(.1)$, (1), (10), return smallest validation error

Cross-validation

for each $\lambda \in \{0.1, 1, 10\}$:
  for $i = 1, \ldots, 5$:
    train h_i on $\mathcal{D} \setminus \mathcal{D}_i$ with λ
    $\mathcal{E}_i$ = error on $\mathcal{D}_i$
  $\mathcal{E}_{val}(\lambda) = (\mathcal{E}_1 + \cdots + \mathcal{E}_5)/5$
return $\lambda^* = \arg\min_\lambda \mathcal{E}_{val}(\lambda)$

- divide data
- train chunks
- choose λ
- find λ^*

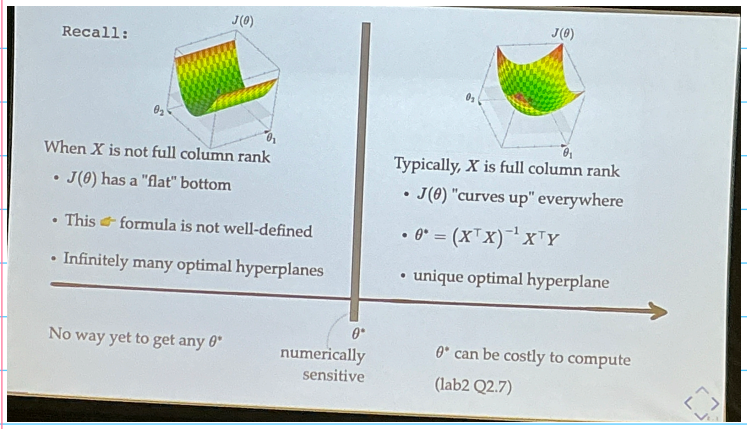
How many hypotheses trained in this example to pick λ^*?

$\theta^*_{final} = (X^TX + n\lambda^* I)^{-1}X^TY$ — data from all of $\mathcal{D}_1$ to $\mathcal{D}_5$

Summary

- When $X^\top X$ is singular or ill-conditioned, OLS is undefined or overfits.
- Regularization combats overfitting by penalizing large θ.
- Ridge regression adds $\lambda\|\theta\|^2$ to the objective — still has a closed-form solution.
- λ is a hyperparameter that trades off fit vs. regularization.
- Validation and cross-validation provide principled ways to choose λ.

2/17 **GRADIENT DESCENT METHODS** (more general & efficient algo to train our ML model)



GRADIENT DESCENT:

GRADIENT of: $\mathbb{R}^m \to \mathbb{R}^m$ is defined @ point $p = (x_1, \ldots, x_m)$ as: $\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$

1: gradient generalizes concept of derivative to multiple dims.

2: gradient's dimensionality always matches function input

3: can be symbolic ($2x, x, y^2$) or numeric ($6, 12, 1$). evaluating symbolic gives numeric

4: gradient points in direction of (steepest) increase in function value

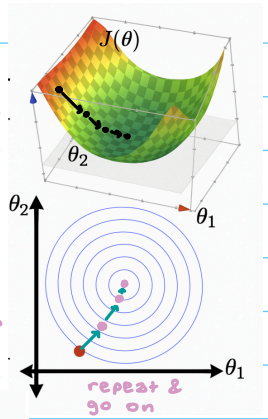
5: gradient @ function minimizer is necessarily zero

GD ALGORITHM:

hyperparameters — initial guess of parameters — precision

Algorithm 1 Gradient Descent($\theta_{\text{init}}, \eta, J, \epsilon$)

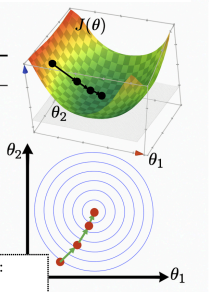
1: Initialize $\theta^{(0)} = \theta_{\text{init}}$ • learning rate
2: Initialize $t = 0$
3: **repeat**
4: $\quad t = t + 1$
5: $\quad \theta^{(t)} = \theta^{(t-1)} - \eta \nabla_\theta J(\theta^{(t-1)})$ • scaling factor
6: **until** $\left|J(\theta^{(t)}) - J(\theta^{(t-1)})\right| < \epsilon$ ← precision gap
7: **return** $\theta^{(t)}$



Algorithm 1 Gradient Descent($\theta_{\text{init}}, \eta, J, \epsilon$)

1: Initialize $\theta^{(0)} = \theta_{\text{init}}$
2: Initialize $t = 0$
3: **repeat**
4: $\quad t = t + 1$
5: $\quad \theta^{(t)} = \theta^{(t-1)} - \eta \nabla_\theta J(\theta^{(t-1)})$
6: **until** $\left|J(\theta^{(t)}) - J(\theta^{(t-1)})\right| < \epsilon$
7: **return** $\theta^{(t)}$

objective improvement is nearly zero.



Other possible stopping criterion for line 6:
- Small parameter change: $\|\theta^{(t)} - \theta^{(t-1)}\| < \epsilon$, or
- Small gradient norm: $\|\nabla_\theta J(\theta^{(t-1)})\| < \epsilon$

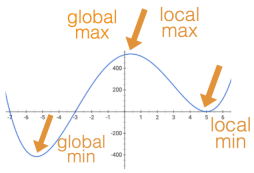
#tracking improvement over time

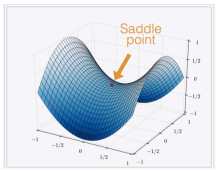
GRADIENT DESCENT PROPERTIES:

When minimizing a function, we aim for a global minimizer.

At a global minimizer $\Rightarrow$ the gradient vector is zero (gradient descent can achieve this (to arbitrary precision))

$\not\Leftarrow$





at a global minimizer $\Leftarrow$ { the gradient vector is zero ← gradient desc. can achieve this to arbitrary prec.
objective function is convex

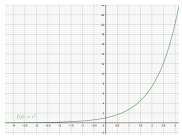
CONVEX: any line segment connecting graph f lies above or on the graph

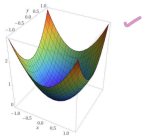
- optimization theory guarantees convex funcs. converge / efficiency

- JMSE always convex
- Jridge with $\lambda > 0$ always strongly convex

More examples

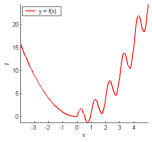
Convex functions

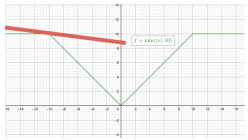


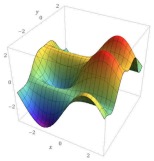


any 2 pts that are connected on graph, the line segment lives above the curve.

Non-convex functions







GRADIENT DESCENT PERFORMANCE:

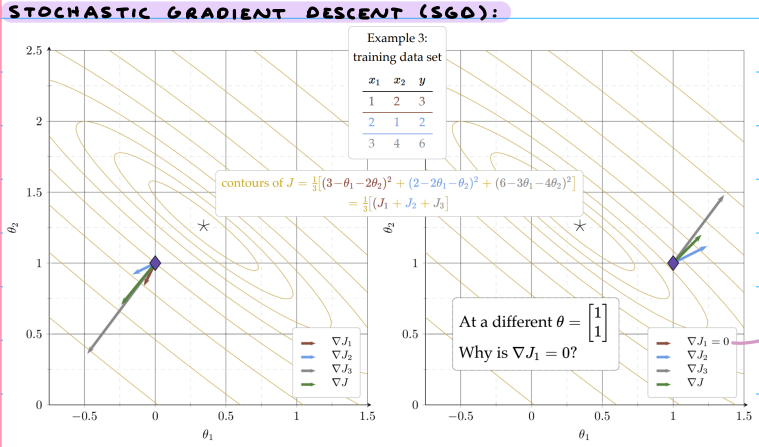
Gradient Descent Performance

- Assumptions:
  - f is sufficiently "smooth" → if violated, might not have gradient & can't run gradient descent
  - f is convex → if violated, might get stuck @ saddle point
  - f has at least one global minimum → if violated, may not terminate / converge (ex: a line)
  - Run gradient descent for sufficient iterations
  - η is sufficiently small

  → if violated, not close enough
- Conclusion:
  - Gradient descent converges arbitrarily close to a global minimizer of f.

STOCHASTIC GRADIENT DESCENT (SGD):

green arrow is the 1/3 avg.



b/c (1,1) is best fit param already, so grad = 0 (no room for improvement)

Gradient of an ML objective

- the MSE of a linear hypothesis:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^\top x^{(i)} - y^{(i)}\right)^2$$

- and its gradient w.r.t. θ:

$$\nabla_\theta J(\theta) = \frac{1}{n}\sum_{i=1}^{n} 2\left(\theta^\top x^{(i)} - y^{(i)}\right)x^{(i)}$$

In general,

- An ML objective function is a finite sum

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$$

- and its gradient w.r.t. θ:

$$\nabla_\theta J(\theta) = \nabla\left(\frac{1}{n}\sum_{i=1}^{n} J_i(\theta)\right) = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta J_i(\theta)$$

👆

👋 (gradient of the sum) = (sum of the gradient)

Gradient of an ML objective

In general,

- An ML objective function is a finite sum

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J_i(\theta)$$

loss incurred on a single i^{th} data point

- and its gradient w.r.t. θ:

$$\nabla_\theta J(\theta) = \nabla\left(\frac{1}{n}\sum_{i=1}^{n} J_i(\theta)\right) = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta J_i(\theta)$$

need to add n of these, each $\nabla_\theta J_i \in \mathbb{R}^d$

= one for each pt

gradient info from the i^{th} data point's loss

Costly in practice!

SGD is rougher but less costly

Algorithm 1 Gradient Descent$(\theta_{\text{init}}, \eta, J, \epsilon)$

1: Initialize $\theta^{(0)} = \theta_{\text{init}}$
2: Initialize $t = 0$
3: **repeat**
4: $\quad t = t + 1$
5: $\quad \theta^{(t)} = \theta^{(t-1)} - \eta\nabla_\theta J(\theta^{(t-1)})$
6: **until** $\left|J(\theta^{(t)}) - J(\theta^{(t-1)})\right| < \epsilon$
7: **return** $\theta^{(t)}$

Algorithm 2 Stochastic Gradient Descent$(\theta_{\text{init}}, \eta, \{J_i\}_{i=1}^n, \epsilon)$

1: Initialize $\theta^{(0)} = \theta_{\text{init}}$
2: Initialize $t = 0$
3: **repeat**
4: $\quad t = t + 1$
5: $\quad i =$ randomly sample from $\{1, \ldots, n\}$
6: $\quad \theta^{(t)} = \theta^{(t-1)} - \eta\nabla_\theta J_i(\theta^{(t-1)})$
7: **until** $\left|J(\theta^{(t)}) - J(\theta^{(t-1)})\right| < \epsilon$
8: **return** $\theta^{(t)}$

GD

$$\nabla_\theta J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta J_i(\theta)$$

SGD

$$\approx \nabla_\theta J_i(\theta)$$

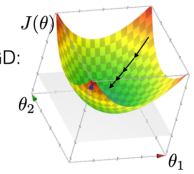
← randomly chosen sample is the true gradient

Compared with GD, SGD:

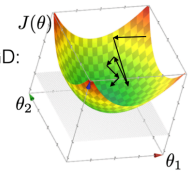
$$\nabla_\theta J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta J_i(\theta) \approx \nabla_\theta J_i(\theta)$$

is more efficient

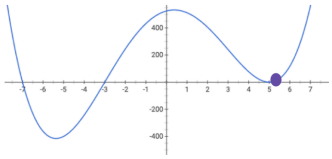
- GD:



- SGD:



is much more "random"

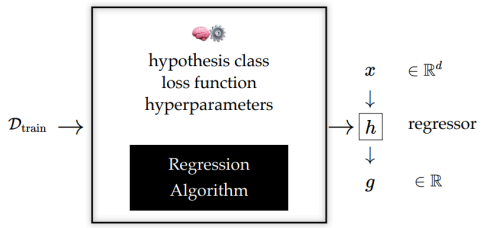


may get us out of a local min

2/23 **LINEAR CLASSIFICATION**

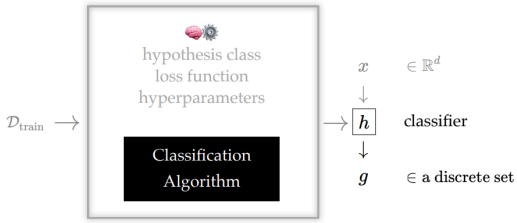
- binary: seperator, normal vector
- logistic: sigmoid, neg. log-likelihood loss
- multi-class: softmax, one-hot encoding, cross-entropy loss

Recap:



"Learn" a model →	↓ "Use" a model
train, optimize, tune, adapt ...	predict, test, evaluate, infer ...
adjusting/updating/finding θ	plug in the θ found
gradient based	no gradients involved

Today:



$\{0, 1\}$
$\{😍, 🥺\}$
{"good", "better", "best", ...}
{"Fish", "Grizzly", "Chameleon", ...}

*intuitively, seems easier since output is discrete set, but it's actually more difficult

LINEAR (binary) CLASSIFIERS:

	linear regressor	linear binary classifier
features	shared $x \in \mathbb{R}^d$	
	$y \in \mathbb{R}$	$y \in \{0, 1\}$
parameters	$\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$	
linear combo	$\theta^T x + \theta_0 = z$ ← linear combo.	
predict	$g = z$	$g = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$ ← 2 options; g is "flat" and discrete in θ
loss $\mathcal{L}(g, y)$	$\mathcal{L}_{\text{squared}} = (g - y)^2$	$\mathcal{L}_{01} = \begin{cases} 0 & \text{if } g = y \\ 1 & \text{otherwise} \end{cases}$ $\mathcal{L}_{01}(g, y)$ is "flat" and discrete in g
optimize method	• closed-form formula • gradient descent	training error almost "flat" w.r.t θ, gradient gives very little info

seperator: @ 0 threshold
θ always points @ + feature space

natural loss func. choice: $g = \text{step}(z) = \text{step}(\theta^T x + \theta_0)$

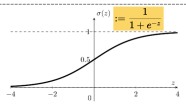
$\mathcal{L}_{01}(g, y) = \begin{cases} 0 & \text{if guess} = \text{label} \\ 1 & \text{o.w.} \end{cases}$ → VERY hard to optimize.
flat almost everywhere (zero grad.), "jumps" elsewhe

LINEAR LOGISTIC (binary) CLASSIFIERS:

Sigmoid $\sigma(\cdot)$: confidence or estimated likelihood that x belongs to the positive class

- monotonic, elegant gradient "smoothed version"

- θ, θ_0 can flip, squeeze, expand, or shift the $\sigma(x)$ graph horizontally

	linear binary classifier	linear *logistic* binary classifier
features	$x \in \mathbb{R}^d$	
parameters	$\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$	
linear combo	$\theta^T x + \theta_0 = z$	
predict		
	$\begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$	$\begin{cases} 1 & \text{if } \sigma(z) > 0.5 \\ 0 & \text{otherwise} \end{cases}$ Sigmoid $\sigma(\cdot)$: *confidence* or *estimated likelihood* that x belongs to the positive class

***linear** logistic* binary classifier

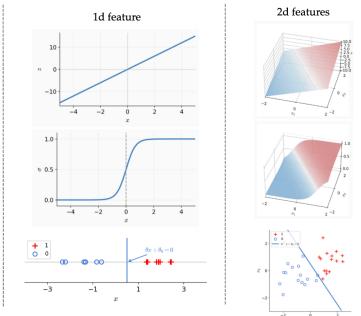
features: $x \in \mathbb{R}^d$
parameters: $\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$

the *logit* z: $z = \theta^\top x + \theta_0$

apply *sigmoid*: $\sigma(z) = \frac{1}{1+e^{-z}}$

Predict 1 if $\sigma(z) > 0.5$, else 0.

$\sigma(z) = 0.5 \iff z = 0 \iff \theta^\top x + \theta_0 = 0$



separator is *linear* in feature x!

$$\mathcal{L}_{\text{nll}}(g, y) = \begin{cases} -\log(g) & \text{if } y = 1 \\ -\log(1-g) & \text{if } y = 0 \end{cases} \Leftrightarrow -[y \log g + (1-y)\log(1-g)]$$

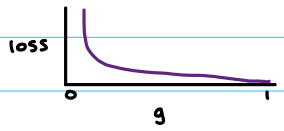
encourage g closer to 1 / g closer to 0

- When $y = 1$: $-[y \log g + (1-y)\log(1-g)] = -\log g$
- When $y = 0$ $-[y \log g + (1-y)\log(1-g)] = -[\log(1-g)]$

Read as: $\sum$ (true label for class k) $\cdot$ $-\log$(predicted prob of class k).
Since $y \in \{0, 1\}$, only the true class's term survives.

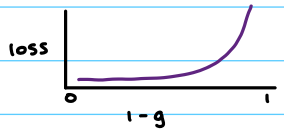
	linear binary classifier	linear *logistic* binary classifier
features	$x \in \mathbb{R}^d$	
label	$y \in \{0, 1\}$	
parameters	$\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$	
linear combo	$\theta^T x + \theta_0 = z$	
predict	$\begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$	$\begin{cases} 1 & \text{if } g = \sigma(z) > 0.5 \\ 0 & \text{otherwise} \end{cases}$
loss	$\begin{cases} 0 & \text{if } g = y \\ 1 & \text{otherwise} \end{cases}$	$\begin{cases} -\log(g) & \text{if } y = 1 \\ -\log(1-g) & \text{if } y = 0 \end{cases}$ $\Leftrightarrow -[y \log g + (1-y)\log(1-g)]$
optimize via	NP-hard to learn	gradient descent

loss func: $L_{nll}(g, y) = -\log g$ (likelihood)



- smooth
- when g almost 1 → little loss
- g close to 0 → big loss

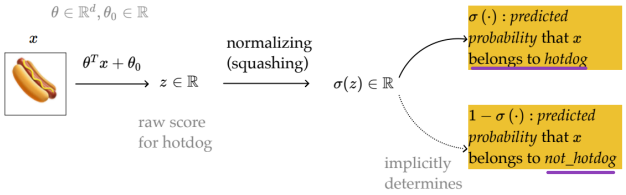
1-g = 1 - σ(·): model's predicted likelihood that x belongs to the negative class



-log(1-g)
low loss = low

LINEAR MULTI-CLASS CLASSIFIERS

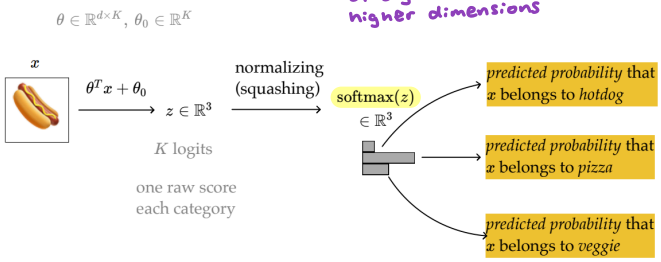
to predict {*hotdog*, or *not_hotdog*}, a scalar logit z suffices



for K classes, use K logit scores.

e.g. $K = 3$: {*hot-dog*, *pizza*, *veggie*}

generalization of sigmoid in higher dimensions



for $K > 2$ classes, a single scalar z no longer suffices — use K logit scores to keep track

softmax: $\mathbb{R}^K \to \mathbb{R}^K$

$$\text{softmax}(z) := \begin{bmatrix} \frac{\exp(z_1)}{\sum_{k=1}^K \exp(z_k)} \\ \vdots \\ \frac{\exp(z_K)}{\sum_{k=1}^K \exp(z_k)} \end{bmatrix}$$

outputs all $\in [0, 1]$, sum to 1

non-neg

e.g.

$$\text{softmax}\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^1}{e^1+e^2+e^3} \\ \frac{e^2}{e^1+e^2+e^3} \\ \frac{e^3}{e^1+e^2+e^3} \end{bmatrix} = \begin{bmatrix} 0.0900 \\ 0.2447 \\ 0.6653 \end{bmatrix}$$

largest # preserved dominance & softer

max among the K logits

"soft" max'd in the output

generalization of σ

sigmoid $\mathbb{R} \to \mathbb{R}$

$$\sigma(z) := \frac{1}{1+\exp(-z)} = \frac{\exp(z)}{\exp(z)+\exp(0)}$$

implicit logit for the negative class

predict positive if $\sigma(z) > 0.5 = \sigma(0)$

softmax: $\mathbb{R}^K \to \mathbb{R}^K$

$$\text{softmax}(z) := \begin{bmatrix} \frac{\exp(z_1)}{\sum_{k=1}^K \exp(z_k)} \\ \vdots \\ \frac{\exp(z_K)}{\sum_{k=1}^K \exp(z_k)} \end{bmatrix}$$

predict the category with the highest softmax score

unifying rule: predict the class with the largest logit

	linear logistic *binary* classifier	one-out-of-K classifier
features	$x \in \mathbb{R}^d$	
parameters	$\theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$	$\theta \in \mathbb{R}^{d \times K}$, $\theta_0 \in \mathbb{R}^K$
linear combo	$\theta^T x + \theta_0 = z \in \mathbb{R}$	$\theta^T x + \theta_0 = z \in \mathbb{R}^K$
predict	$\sigma(z) = \frac{\exp(z)}{\exp(0)+\exp(z)}$ predict positive if $\sigma(z) > \sigma(0)$	$\text{softmax}(z) = \begin{bmatrix} \frac{\exp(z_1)}{\sum_{k=1}^K \exp(z_k)} \\ \vdots \\ \frac{\exp(z_K)}{\sum_{k=1}^K \exp(z_k)} \end{bmatrix}$ predict the class with the highest softmax score

ONE-HOT ENCODING:

- generalizes from {0,1} binary labels
- encode K classes as $\mathbb{R}^K$ vector w/ single 1 (hot) & 0s elsewhere

Negative log-likelihood $K-$ classes loss (aka, cross-entropy)

g : softmax output

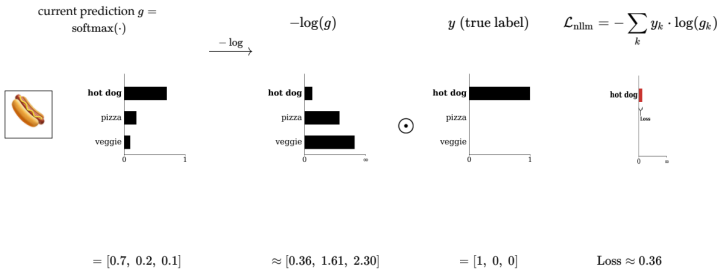
g_k : probability or confidence of belonging in class k

$$\mathcal{L}_{\text{nllm}}(g, y) = -\sum_{k=1}^{K} y_k \cdot \log(g_k)$$

y :one-hot encoding label

y_k : kth entry in y, either 0 or 1

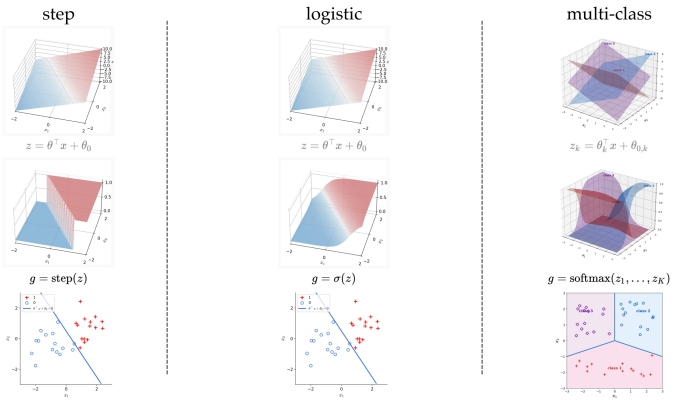
- Generalizes negative log likelihood loss $\mathcal{L}_{\text{nll}}(g, y) = -[y \log g + (1-y)\log(1-g)]$
- Despite the $K-$term sum, only the term corresponding to its true class label contributes, since all other $y_k = 0$



To reduce the loss, $g_{\text{hot dog}}$ needs to go up — this signal flows smoothly back to θ through −log and softmax, so we can optimize via gradient descent.

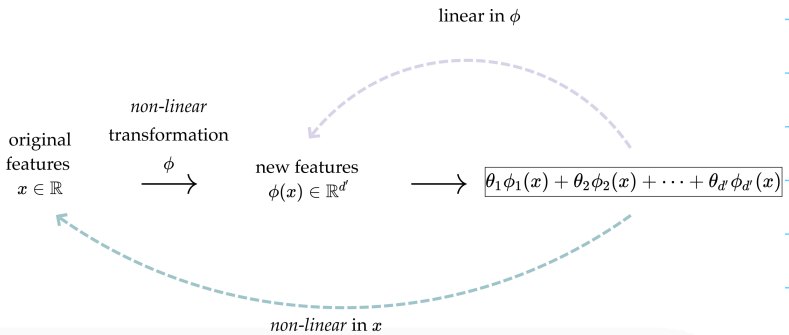
3/2 **FEATURES, NEURAL NETWORKS**

Recall:



decision boundary is *linear* in feature x

SYSTEMATIC FEATURE TRANSFORMATIONS



systematic polynomial features construction

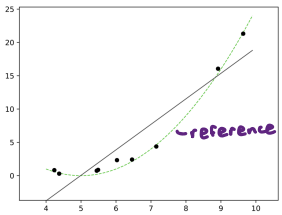
	$d=1$, features: x_1	$d=2$, features: x_1, x_2
$k=0$	1	1
$k=1$	$1,\ x_1$	$1,\ x_1,\ x_2$
$k=2$	$1,\ x_1,\ x_1^2$	$1,\ x_1,\ x_2,\ x_1^2,\ x_1x_2,\ x_2^2$
$k=3$	$1,\ x_1,\ x_1^2,\ x_1^3$	$1,\ x_1,\ x_2,\ x_1^2,\ x_1x_2,\ x_2^2,\ x_1^3,\ x_1^2x_2,\ x_1x_2^2,\ x_2^3$
⋮	⋮	⋮

build up higher & higher terms (more poly relationships)

- Elements in the basis are the monomials of original features raised up to power k
- With a given d and a fixed k, the basis is **fixed**.

- $n = 9$ data points, each with feature $x \in \mathbb{R}$ and label $y \in \mathbb{R}$
- data generated from green dashed curve

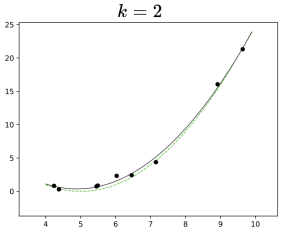
$k = 1$



$h = \theta_0 + \theta_1 x$

Learn 2 parameters — degree-1

$k = 2$



$h = \theta_0 + \theta_1 x + \theta_2 x^2$

Learn 3 parameters — degree-2

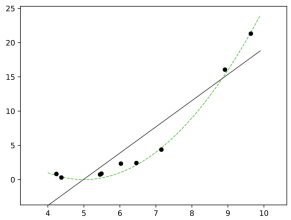
K: hyperparameter that determines capacity (expressiveness) of hypothesis class.

- models w/ many rich features & free params tend to have high capacity but greater risk of overfitting.

- how to choose k? validation/cross validation

- $n = 9$ data points, each with feature $x \in \mathbb{R}$ and label $y \in \mathbb{R}$
- data generated from green dashed curve

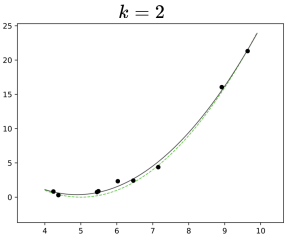
$k = 1$



$h = \theta_0 + \theta_1 x$

Learn 2 parameters — degree-1

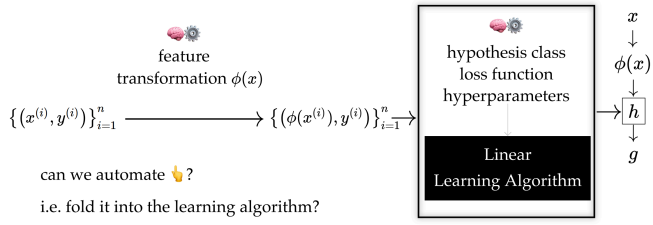
$k = 2$



$h = \theta_0 + \theta_1 x + \theta_2 x^2$

Learn 3 parameters — degree-2

today, so far:

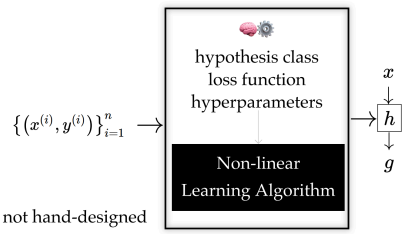


can we automate 👆?

i.e. fold it into the learning algorithm?

NEURAL NETWORKS:

neural networks:



the nonlinearity ϕ is learned not hand-designed

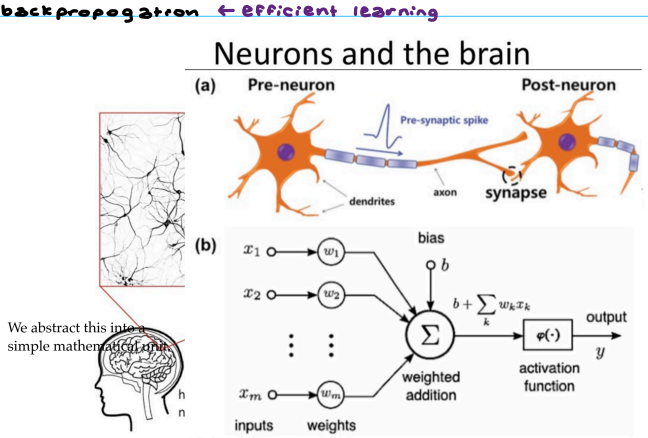
NN FUNDAMENTALS:

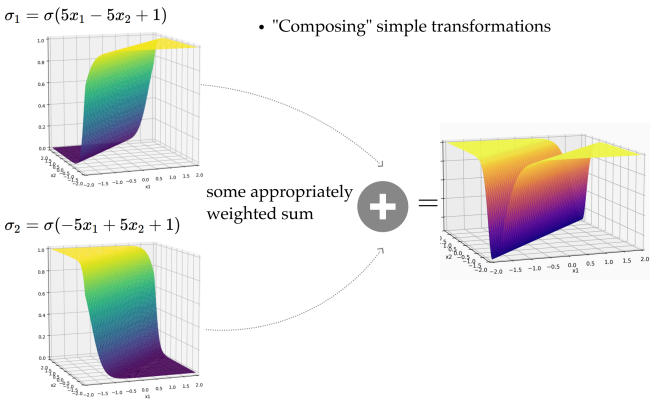
- nonlinear feature transformation
- composing simple nonlinearities amplifies this effect

expressiveness

layered structure

- backpropogation ← efficient learning

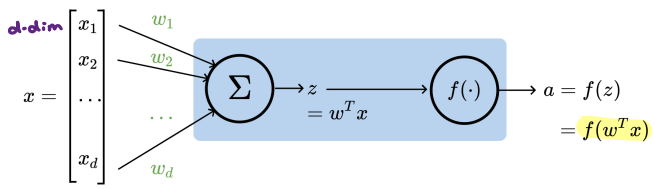


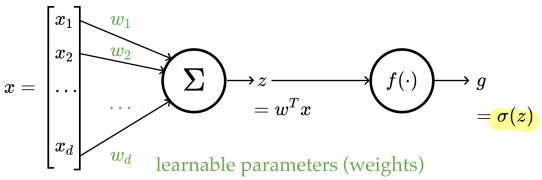


- "Composing" simple transformations

e.g. linear logistic classifier represented as a computation graph

A neuron:





- x: d-dimensional input
- w: weights (i.e. parameters) old θ
- z: pre-activation output scalar
- f: activation function scalar in scalar out
- a: post-activation output

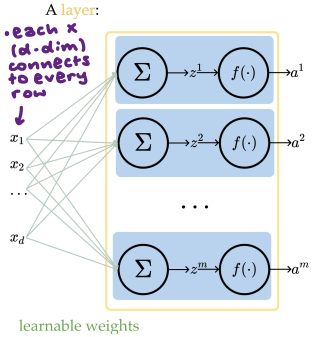
w: what the algorithm learns

z: scalar
↓
f: what we engineers choose
↓
a: scalar

Choose activation $f(z) = \sigma(z)$

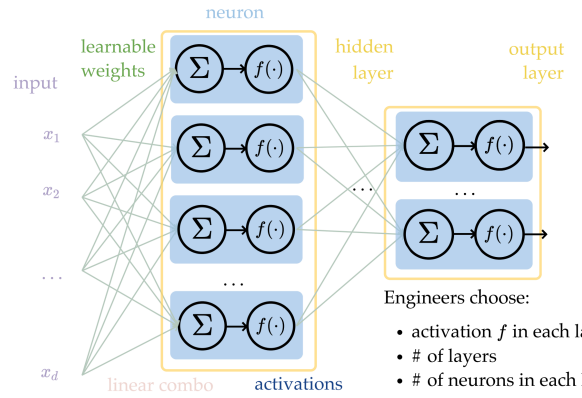
LAYER:

A layer:



- (# of neurons) = layer's output dimension.
- activations applied element-wise, e.g. z^1 won't affect a^2. (softmax in the output layer is a common exception)
- all neurons in a layer typically share the same f (mixed f is possible but complicates the math).
- layers are typically fully connected, each input x_i influences every a^j .

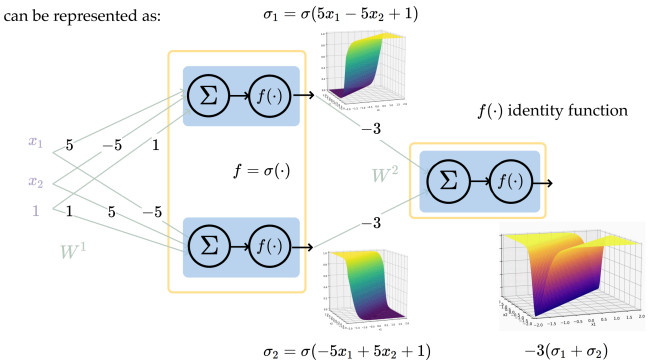
A (fully-connected, feed-forward) neural network:



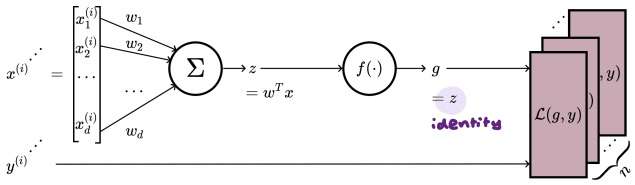
Engineers choose:

- activation f in each layer
- # of layers
- # of neurons in each layer

can be represented as:

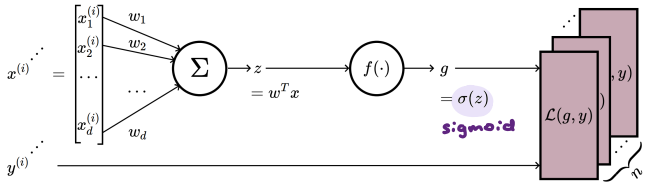


e.g. forward-pass of a linear regressor



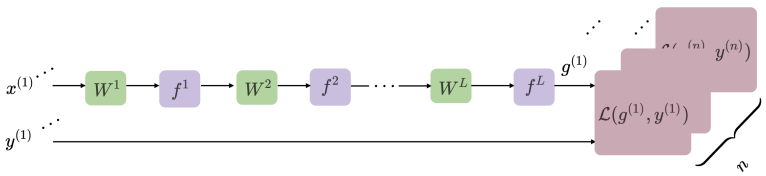
- Activation f is chosen as the identity function
- Evaluate the loss $\mathcal{L} = (g^{(i)} - y^{(i)})^2$
- Repeat for each data point, average the sum of n individual losses

e.g. forward-pass of a linear logistic classifier



- Activation f is chosen as the sigmoid function
- Evaluate the loss $\mathcal{L} = -[y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log (1 - g^{(i)})]$
- Repeat for each data point, average the sum of n individual losses

Forward pass: evaluate, *given* the current parameters



- the model outputs $g^{(i)} = f^L\left(\ldots f^2\left(\ f^1(\mathbf{x}^{(i)}; \mathbf{W}^1); \mathbf{W}^2\right); \ldots \mathbf{W}^L\right)$
- the loss incurred on the current data $\mathcal{L}(g^{(i)}, y^{(i)})$
- the training error $J = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(g^{(i)}, y^{(i)})$

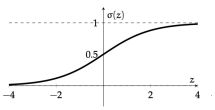
linear combination
(nonlinear) activation
loss function

DESIGN CHOICES

Hidden layer activation function f choices

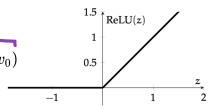
σ used to be the most popular

- firing rate of a neuron
- elegant gradient $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$



nowadays, the default choice:

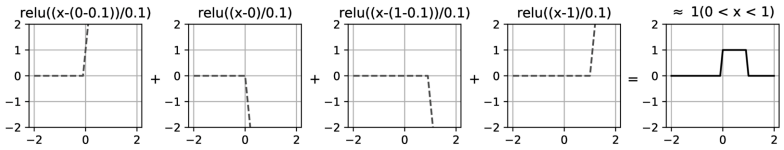
$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} = \max(0, z) = \max(0, \underbrace{w^T x + w_0}_{z})$$



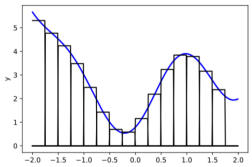
very simple function form (so is the gradient).

$$\frac{\partial \text{ReLU}(z)}{\partial z} = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$$

compositions of ReLU(s) can be quite expressive



asymptotically, can approximate any continuous function arbitrarily well (for regression)



therefore can also approximate arbitrary decision boundaries (for classification)

- compositions of ReLU can be quite expressive (even just 4 ReLU neurons already carves a non-trivial surface)

WIDTH: # neurons in layers

DEPTH: # of layers

→ increasing width/depth (w/ non-linear activation) makes model more expressive but also incr. risk of overfitting

3/9

Recap

$\sigma_1 = \sigma(5x_1 - 5x_2 + 1)$

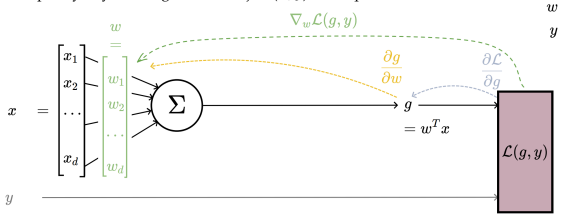
$\sigma_2 = \sigma(-5x_1 + 5x_2 + 1)$

$-3(\sigma_1 + \sigma_2)$

BACKWARD PASS (to learn params / weights)

for simplicity, say training data set is just (x, y) and squared loss

$x \in \mathbb{R}^d$
$w \in \mathbb{R}^d$
$y \in \mathbb{R}$

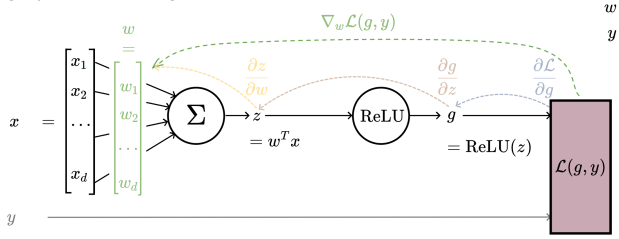


chain rule

$$\nabla_w \mathcal{L}(g,y) = \frac{\partial \mathcal{L}(g,y)}{\partial w} = \frac{\partial g}{\partial w} \cdot \frac{\partial \mathcal{L}}{\partial g} = \frac{\partial g}{\partial w} \cdot \frac{\partial (g-y)^2}{\partial g} = \frac{\partial (w^T x)}{\partial w} \cdot 2(g-y) = x \cdot 2(g-y)$$

Slightly more interesting:

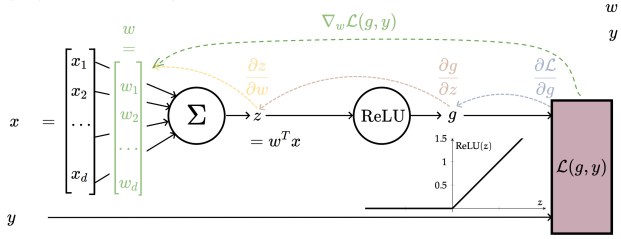
$x \in \mathbb{R}^d$
$w \in \mathbb{R}^d$
$y \in \mathbb{R}$



$$\nabla_w \mathcal{L}(g,y) = \frac{\partial \mathcal{L}(g,y)}{\partial w} = \frac{\partial z}{\partial w} \cdot \frac{\partial g}{\partial z} \cdot \frac{\partial \mathcal{L}}{\partial g} = \frac{\partial z}{\partial w} \cdot \frac{\partial g}{\partial z} \cdot \frac{\partial (g-y)^2}{\partial g} = x \cdot \frac{\partial [(\mathrm{ReLU}(z))]}{\partial z} \cdot 2(g-y)$$

Slightly more interesting:

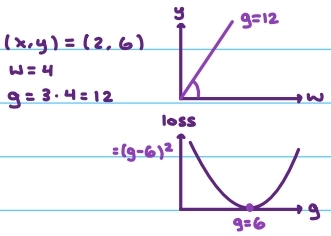
$x \in \mathbb{R}^d$
$w \in \mathbb{R}^d$
$y \in \mathbb{R}$

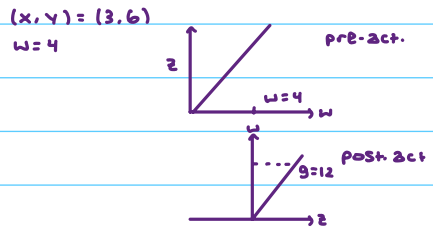


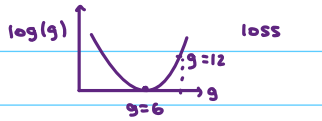
$$\nabla_w \mathcal{L}(g,y) = x \cdot \frac{\partial [(\mathrm{ReLU}(z))]}{\partial z} \cdot 2(g-y) = \begin{cases} 0, & \text{if } z < 0 \\ 2x(g-y), & \text{otherwise} \end{cases}$$

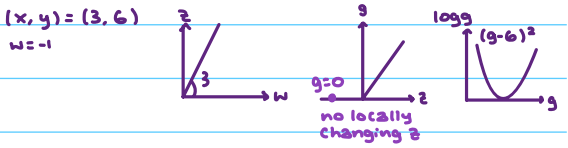
$$= \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{otherwise} \end{cases}$$

example on the blackboard

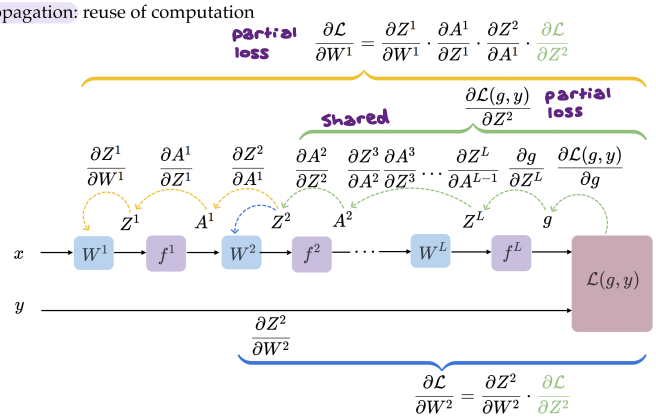








backpropagation: reuse of computation



Training a neural network: the full loop

*could either evaluate all partial derivatives to numerical vectors or for in chunks

Initialize all weights $W^1, W^2, \ldots, W^L$ randomly

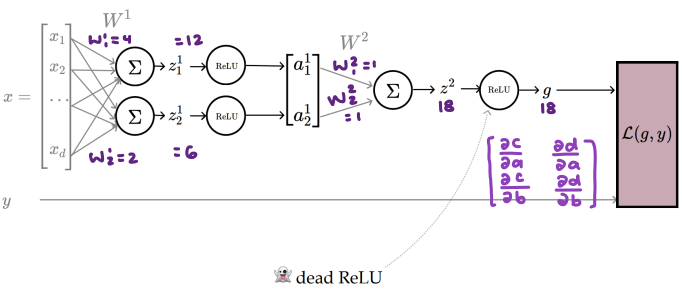
Repeat until stopping criterion:

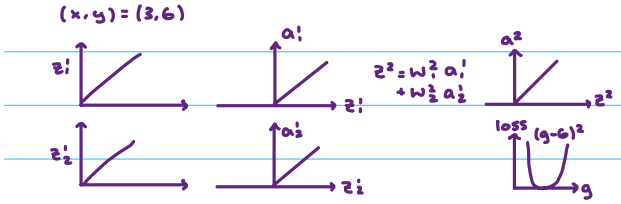
1. Forward pass: for each data point, compute $Z^1, A^1, Z^2, A^2, \ldots, g^{(i)}$ evaluate
2. Evaluate loss: for each data point, compute $\mathcal{L}(g, y)$
3. Backward pass: pick a data point, compute $\nabla_{W^\ell}\mathcal{L}(g^{(i)}, y^{(i)})$ for all $\ell = L, L-1, \ldots, 1$ via the chain rule
   (reuse intermediate results → backpropagation) to get vals & learn params
   e.g., $\frac{\partial \mathcal{L}}{\partial W^1} = \frac{\partial Z^1}{\partial W^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial Z^2}{\partial A^1} \cdot \frac{\partial \mathcal{L}}{\partial Z^2}$
   $\frac{\partial \mathcal{L}}{\partial W^2} = \frac{\partial Z^2}{\partial W^2} \cdot \frac{\partial \mathcal{L}}{\partial Z^2}$
4. Update: $W^\ell \leftarrow W^\ell - \eta \nabla_{W^\ell}\mathcal{L}$ for all ℓ

*not guaranteed to be optimal

ISSUES & REMEDIES

if $z^2 < 0$, no weights get updated



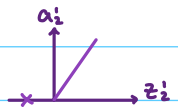


$A \in \mathbb{R}^{a_0 \times a_1}$ $B \in \mathbb{R}^{b_0 \times b_1}$

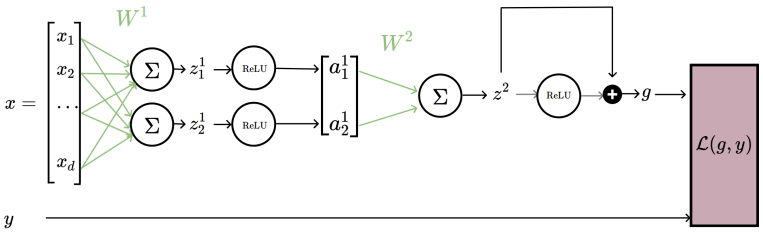
$\frac{\partial A}{\partial B} = [\ ] \in \mathbb{R}^{b_0 \times b_1 \times a_0 \times a_1}$ ← $(a_0 \times a_1) \cdot (b_0 \times b_1)$ partial derivatives to calculate

Ex: $y = -1$, $z_2 = -3$,

input into ReLU is ⊖



Residual (skip) connection:



Now, $g = z^2 + \text{ReLU}(z^2)$

even if $z^2 < 0$, with skip connection, weights in earlier layers can still get updated

Vanishing and Exploding Gradients

Backpropagation computes gradients via the chain rule — a product of many factors:

e.g. $$\frac{\partial \mathcal{L}}{\partial W^1} = \frac{\partial Z^1}{\partial W^1} \cdot \underbrace{\frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial Z^2}{\partial A^1} \cdots \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \frac{\partial Z^L}{\partial A^{L-1}}}_{L-1 \text{ layers of multiplicative factors}} \cdot \frac{\partial g}{\partial Z^L} \cdot \frac{\partial \mathcal{L}}{\partial g}$$

If each factor is small, their product shrinks very quickly with depth, little learning.

If factors are large (magnitude> 1), gradients explode, also problematic.

Many practical remedies: residual connection, gradient clipping

Gradient clipping: If $\|\nabla\| > \tau$, rescale: $\nabla \leftarrow \tau \frac{\nabla}{\|\nabla\|}$.

Preserves gradient direction, caps its magnitude

Summary

- Multi-layer perceptrons automatically learn good features and transformations from data.
- Training loop: forward pass → loss → backward pass → weight update, repeat.
- Backpropagation reuses intermediate computations to efficiently evaluate all gradients via the chain rule.
- Dead ReLU neurons (pre-activation always negative) get zero gradient and stop learning; careful initialization helps.
- Vanishing/exploding gradients arise from multiplying many small (or large) factors across layers.

Reference: weight initialization

- If all weights start at 0, every neuron computes the same thing → no learning
- If weights are too large, activations saturate or explode
- If weights are too small, signals shrink to zero across layers

Goal: keep the variance of activations roughly constant across layers.

- **Xavier initialization** (for sigmoid/tanh): $W_{ij}^{\ell} \sim \mathcal{N}\left(0,\ \frac{1}{n_{\ell-1}}\right)$

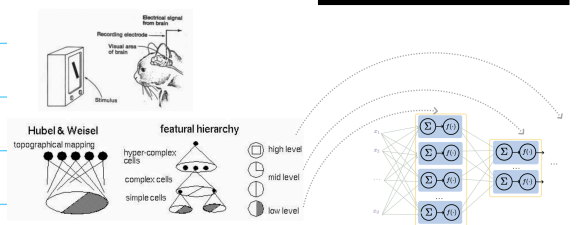
  where $n_{\ell-1}$ = number of inputs (fan-in) to the layer
- **He initialization** (for ReLU): $W_{ij}^{\ell} \sim \mathcal{N}\left(0,\ \frac{2}{n_{\ell-1}}\right)$

  factor of 2 compensates for ReLU zeroing out half the inputs

3116 **CONVOLUTIONAL NEURAL NETWORKS**

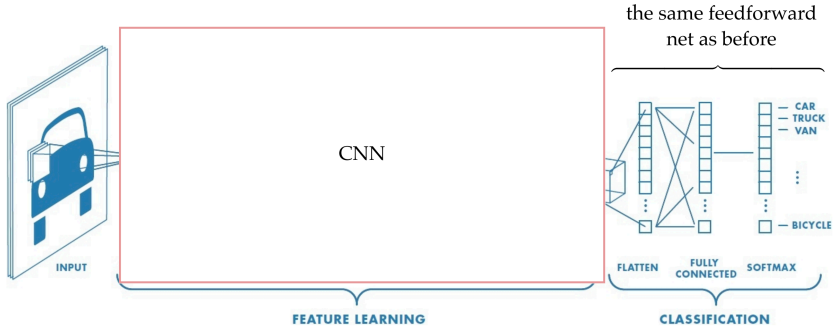
- 784 inputs → must calculate 13K params
- have a specialized, structured network to help fight overfitting & partly fully-connected nets don't scale well for vision tasks
- visual hierarchy
- spatial locality (zoom in & compare things around a pt)
- translational invariance (should recognize all types despite differences)

* CNN EXPLOITS ___ via layered structure, convolution, and pooling to handle images efficiently & effectively

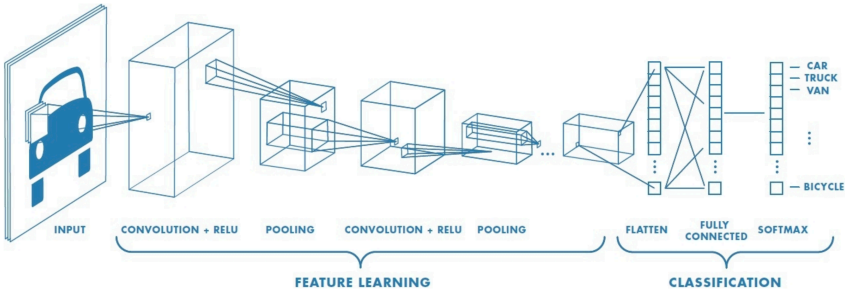


Layered structure are well-suited to model this hierarchical processing.

typical CNN architecture for image classification



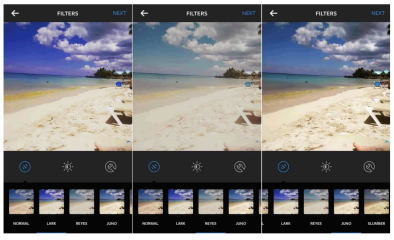
typical CNN structure for image classification



Convolutional layer might sound foreign, but it's very similar to a fully-connected layer

Layer	Forward pass, *do*	Backward pass, *learn*	Design choices
fully-connected	dot-product, activation	neuron weights	neuron count, etc.
convolutional	convolution, activation	filter weights	conv specs, etc.

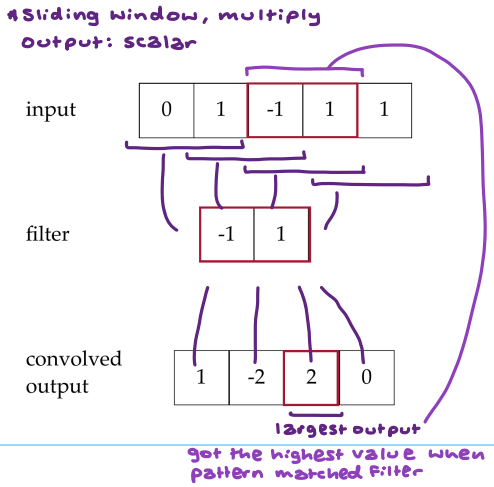
Convolution result:



CONVOLUTION:

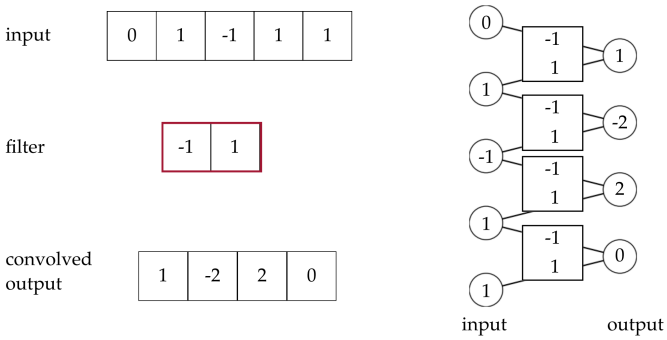
→ template matching (the filter): 1D

convolution interpretation 1: template matching

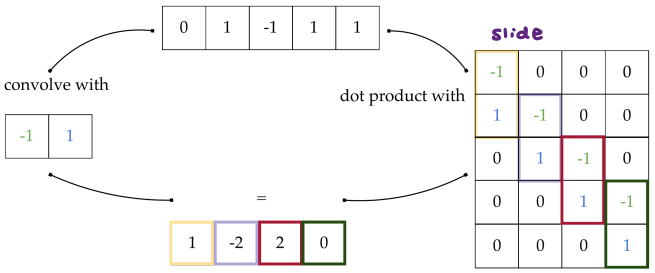


convolution interpretation 2: "look" *locally* through the filter

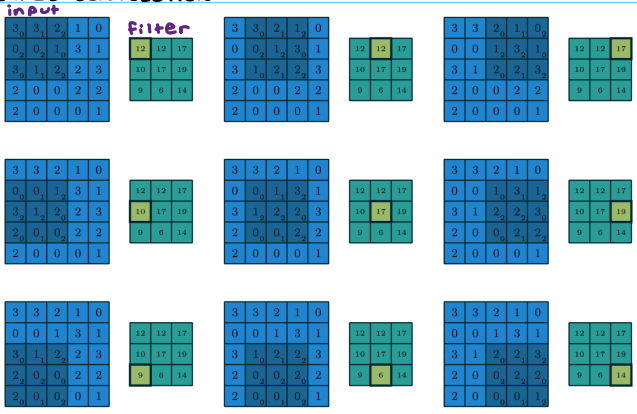
this local region = *receptive field*



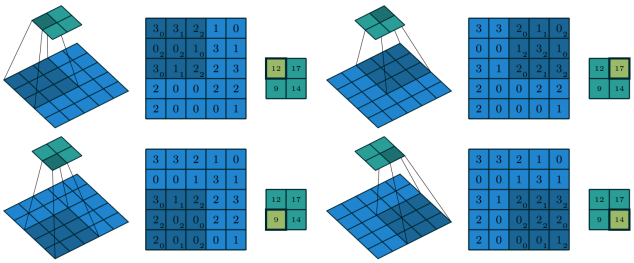
convolution interpretation 3: sparse-connected layer with parameter sharing



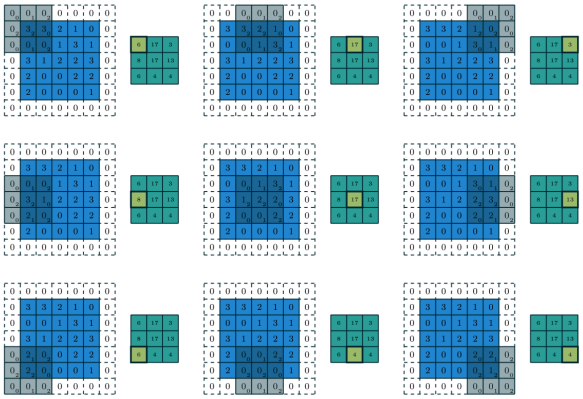
EX: 2D CONVOLUTION



STRIDE OF 2:

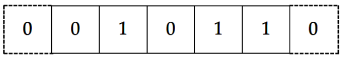


STRIDE OF 2, PADDING SIZE 1

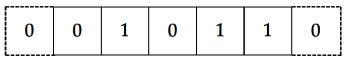


quick summary: hyperparameters for 1d convolution

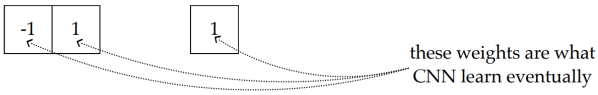
- Zero-padding



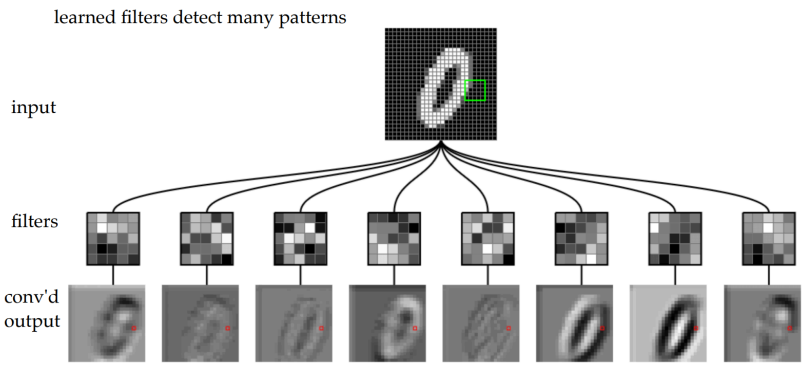
- Stride (e.g. stride of 2)



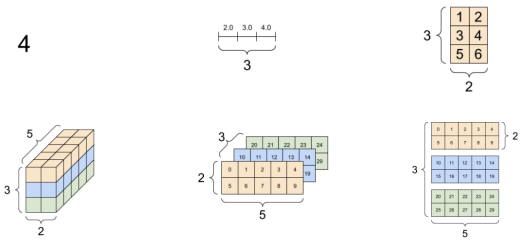
- Filter size (e.g. we saw these two in 1-d)



Each filter helps w/ pattern recognition:



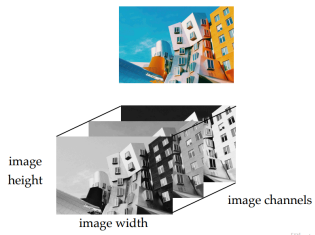
3D TENSORS



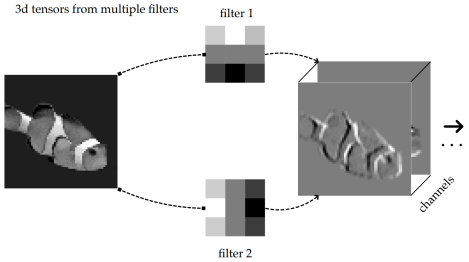
3D tensors: 2D matrices stacked tog.

- collection of 2D images (ex: R, G, B) → each channel is a complete but independent view of the same scene.

3d tensors from color channels



3d tensors from multiple filters



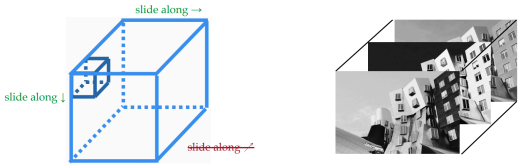
can stack complete but ind. channels

CHANNELS come from:

1) color input (width x height channels)

2) multiple filters → multiple channels

Why we *don't* typically do 3d convolution

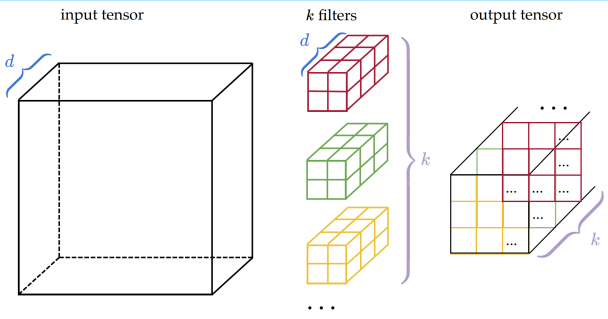


Convolution **shares** weights across **shifted** positions

shifting makes sense spatially (a cat can be anywhere)

but not across channels (red ≠ shifted green)

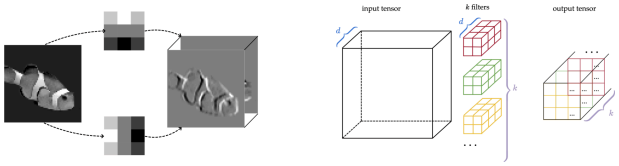
3D conv is used when the third axis is spatial/temporal (MRI, video)



Every convolutional layer works with 3d tensors:

1. color input

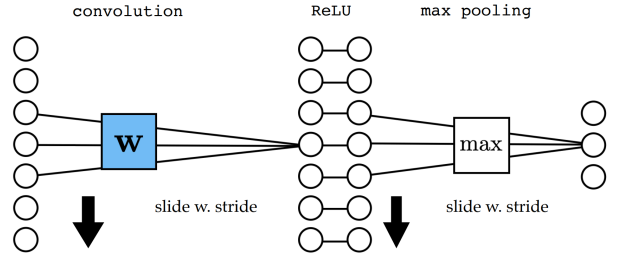
2. the use of multiple filters in doing 2d convolution



MAX POOLING: & RELU → 'idc where the pattern was; i just care if it existed'

POOLING: similar to convolution, but instead of learning a filter, use max pooling filter to

1d max pooling



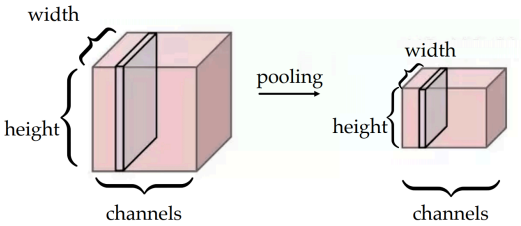
learnable filter weights
detects pattern

no learnable parameter
summarizes strongest response

*pooling across spatial locations achieves invariance w.r.t. small translations

- large response regardless of exact position of edge

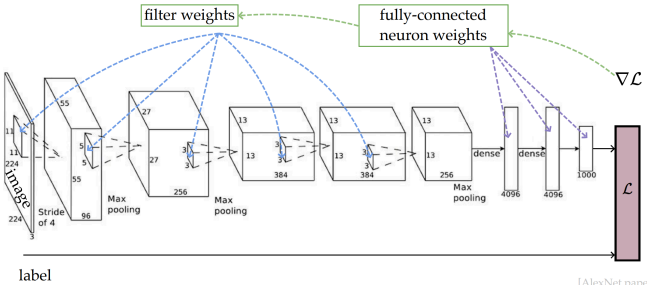
Pooling across *spatial* locations achieves invariance w.r.t. small translations:



applied independently across all channels

so the *channel* dimension remains *unchanged*

imagenet classification error



label

[AlexNet paper]
[all max pooling are 3×3 filter, stride 2; pooled outputs not explicitly shown on diagram: 27×27×96, 13×13×256, 6×6×256]

Here are the building blocks we will use to build a CNN:

- Convolutional layers
- Max pooling layers
- Fully connected layers

Assume that the task of our CNN is to detect animals like cats, dogs, lizards etc. in an image.

2.1)

Which type of layer would be best to use to detect low-level features (like lines, curves, etc.) of an animal in an image?

Choose one: Convolutional "pattern matcher"

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Convolutional

Explanation:

Filters in the first convolutional layers are responsible for detecting low-level features (e.g., edges, color, contrast). Later convolutional layers are responsible for detecting mid-level features (e.g., ears, eyes).

designed to look for specific local structure. (1 surrounded by 0s)

2.2)

Suppose we are given a set of inputs where each input represents whether a useful feature of animals (tail, beak, whiskers, etc.) occurs anywhere in a image. Which type of layer would be best to use to turn these inputs into a classification of whether the image is a cat, dog, lizard, etc.?

Choose one: Fully connected OR gate: the "decision maker"

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Fully connected

Explanation:

Fully connected layers allow combining features from the entire image and provide the final network output.

1 if image contains @ least one instance of the filter (w/ max(x,0) applied to it)

2.3)

Suppose you have detected occurrences of some *useful* low-level feature, like a furry tail, throughout your image. You would like to use these local detections to figure out whether or not the furry tail occurred anywhere within larger regions of the image. Which layer would be best to use?

Choose one: Max pool "evidence gatherer"

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: Max pool

Explanation:

Max pooling layers detect the strongest response within a given window. This property allows the network to be less sensitive to feature locations.

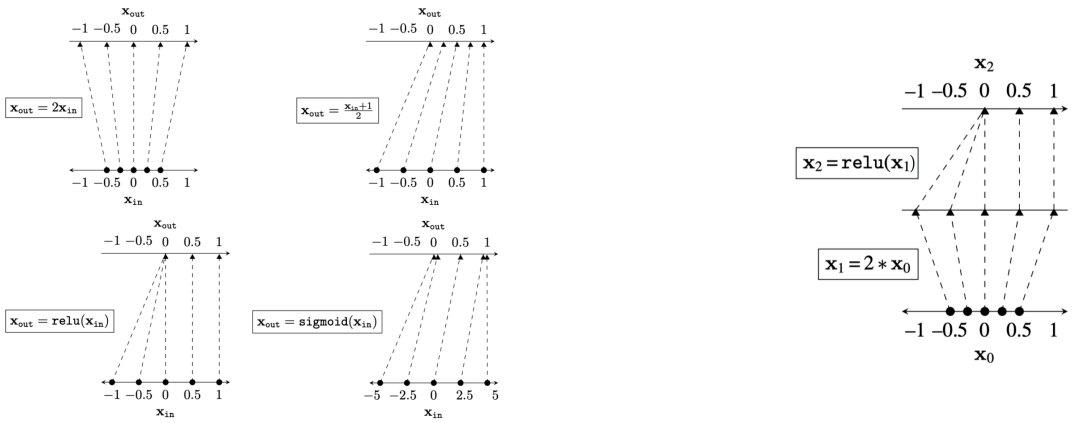
$f(x) = \max(x, 0)$

RELU: filter for relevance (keep matches but turns noise to 0)

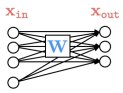
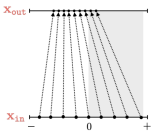
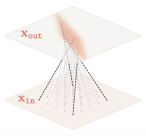
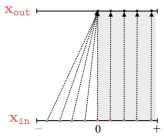
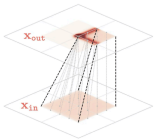
max pooling: summarizes the region. "idc where the pattern was; i just care if it existed"

3/30 Representation layers:

Representation transformations for a variety of neural net operations and stack of neural net operations



Parameters

	wiring graph	equation	mapping 1D	mapping 2D
linear		$x_{out} = Wx_{in}$		
relu		$x_{out_i} = \max(x_{in_i}, 0)$		

Training NN classifier:

$g = \text{softmax}(z_2)$

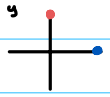
$z_2 = \text{linear}(a_1) \in \mathbb{R}^2$

$a_1 = \text{ReLU}(z_1)$

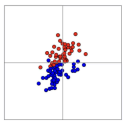
$z_1 = \text{linear}(x) \in \mathbb{R}^2$

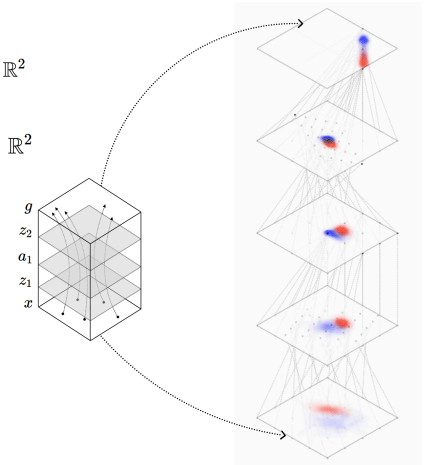
$x \in \mathbb{R}^2$

target out:



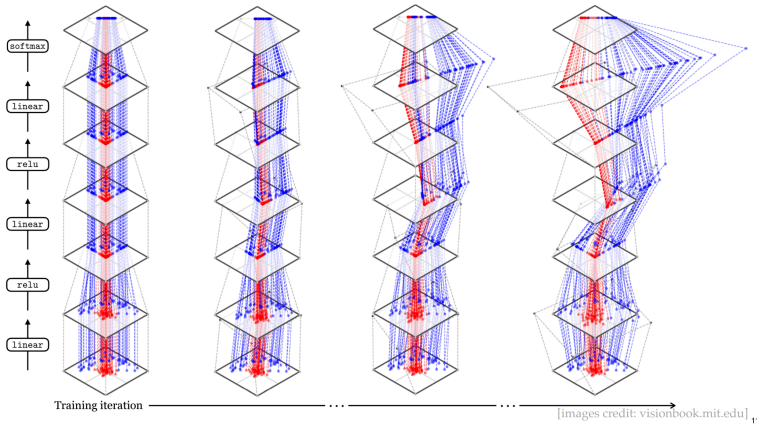
Training data





maps from complex data space to desired target space

[images credit: visionbook.mit.edu] 10



[images credit: visionbook.mit.edu] 11

- learnable weights help w/ different scattering
- each layer = different representation (embedding) of data
- from latent embeddings to data: gen. modelling

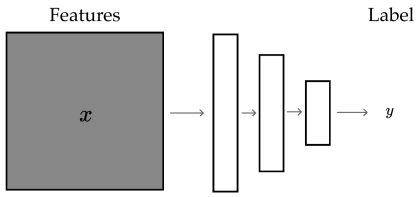
humans also learn representations:

- compact (minimal)
- explanatory (roughly sufficient)
- disentangled (independent factors)
- interpretable (understandable)
- make subsequent problem solving easy

SELF-SUPERVISED LEARNING

supervised:

Label prediction (supervised learning)

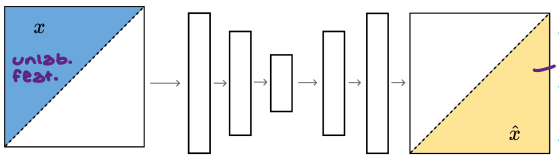


Unlabeled features x are abundant

Labels y are expensive…

self-supervised:

Self-supervised Learning (partial feature reconstruction)



unlabelled features

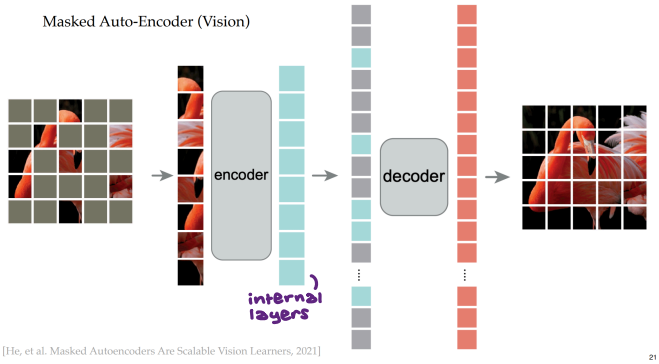
Partial Features

Other Partial Features

Harder reconstruction, forces understanding.

lots of unlabelled features on the internet!

Masked Auto-Encoder (Vision)



[He, et al. Masked Autoencoders Are Scalable Vision Learners, 2021]

21

MASKING: predict color from grayscale (one method)

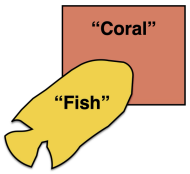
w/o relying on labels

AUTO-ENCODERS

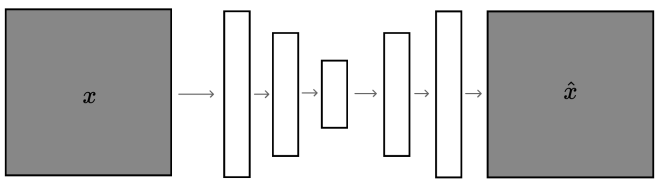
Good representations are:

Auto-encoders explicitly aims
- Compact (*minimal*)
- Explanatory (*roughly sufficient*)

these may just emerge as well
- Disentangled (*independent* factors)
- Interpretable (understandable by humans)
- Make *subsequent problem solving easy*



UNSUPERVISED LEARNING (feature reconstruction)



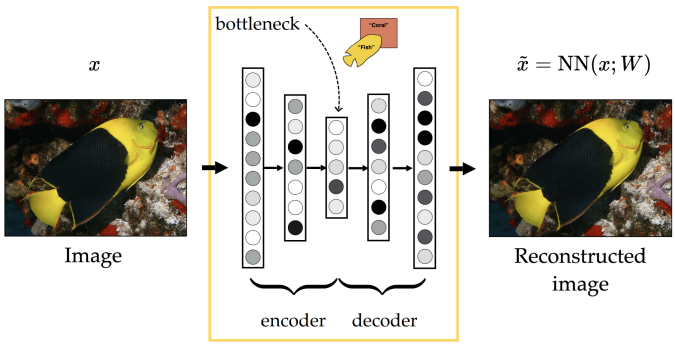
Features

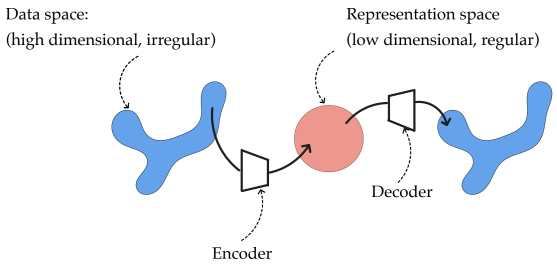
Reconstructed Features

Auto-Encoder:

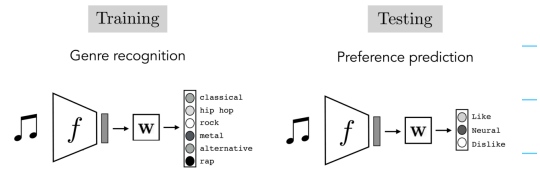
Auto-encoder

$$\min_W ||x - \tilde{x}||^2$$



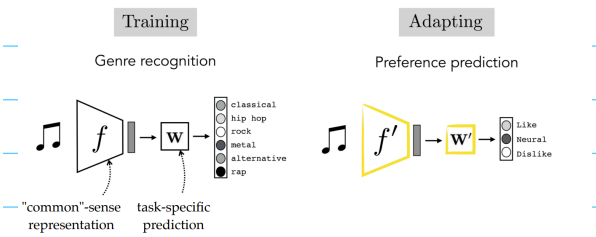


training & testing:



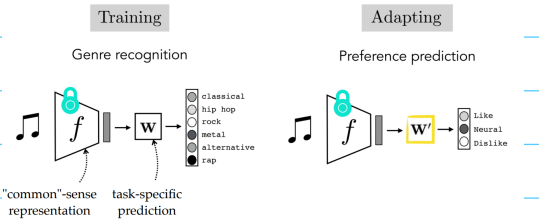
Often, what we will be "tested" on is not what we were trained on.

[images credit: visionbook.mi

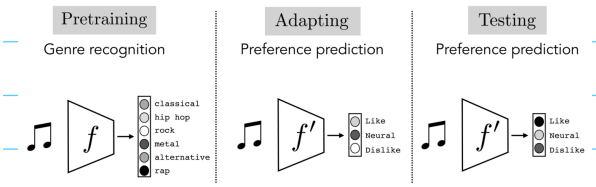


Finetuning: initialize f' as f, then continue training for f' as well, on new target data

[images credit: visionbook.mit.edu

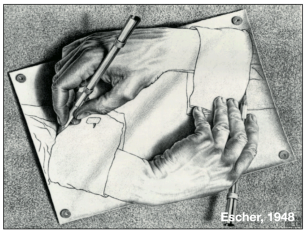


Final-layer adaptation: freeze f, train a new final layer to new target data



Word Embeddings:

Large Language Models (LLMs) are trained in this self-supervised way

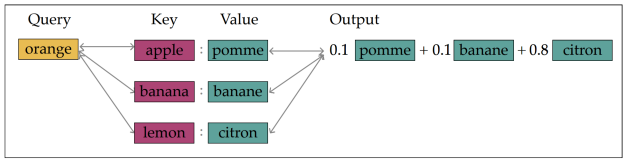


- Scrape the internet for plain texts.
- Cook up "labels" (prediction targets) from these texts.
- Convert "unsupervised" problem into "supervised" setup.

embed words → vectors

(2 words close semantically will be closer together)

#good word embeddings space is equipped w/ sensible dot-prod. similarity



We put (query, key, value) in "good" embeddings in our human brain

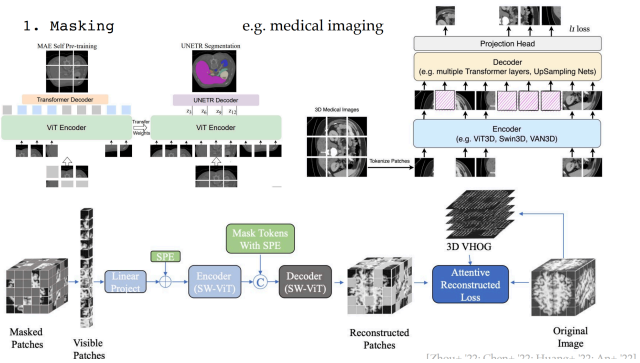
such that "merging" the values 0.1 pomme + 0.1 banane + 0.8 citron

via these merging percentages [0.1 0.1 0.8] made sense

very roughly, the attention mechanism in transformers automates this process. We'll see the details next week.

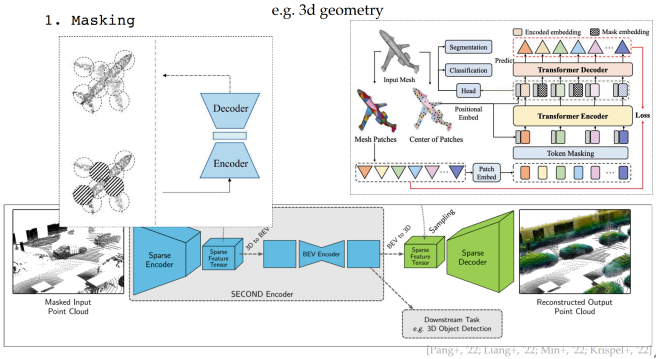
MASKING:

1. Masking e.g. medical imaging



[Zhang+ '22; Chen+ '22; Huang+ '22; Ana+ '22]

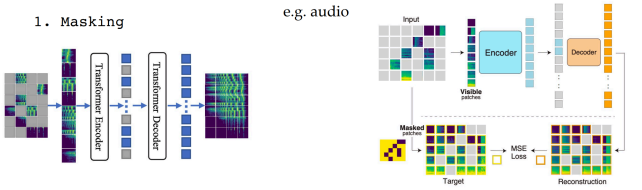
1. Masking e.g. 3d geometry

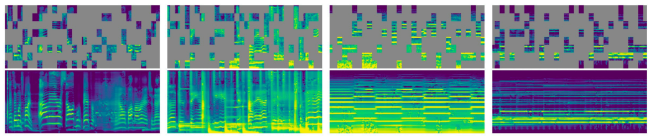


[Pang+, '22; Liang+, '22; Min+, '22; Krispel+, '22]

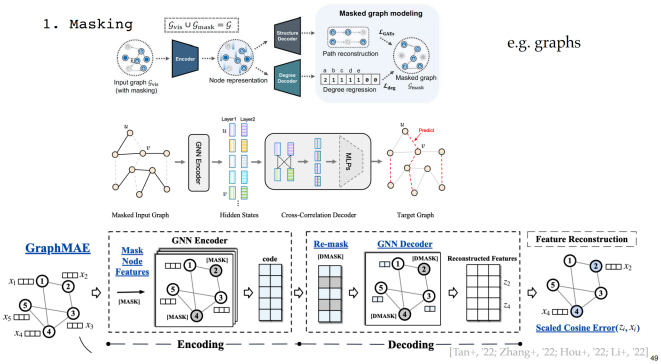
47

1. Masking e.g. audio





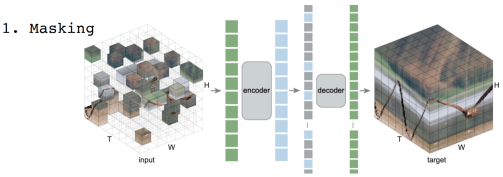
1. Masking e.g. graphs

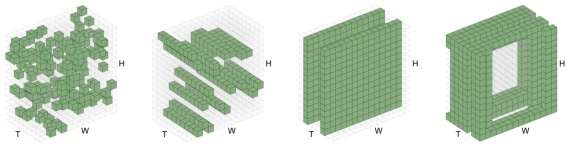


[Tan+, '22; Zhang+, '22; Hou+, '22; Li+, '22]

49

1. Masking





(a) agnostic, 90% (b) space-only, 90% (c) time-only, 75% (d) block, 75%

*masking 75% is optimal for images
15% optimal for languages
95% optimal for videos

[Feichtenhofer, et al., "Masked Autoencoders As Spatiotemporal Learners", NeurIPS 2022] 52

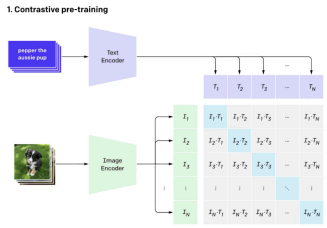
CONTRASTIVE LEARNING:

- glimpse of an object might have association w/ smthg else
- crop parts of image & construct network s.t. patches coming from same image are encoded close together

MULTI-MODALITY

- predict sound cow makes based on image of cow
- ex: input = only video sound of ocean, LLM output = sound of ocean

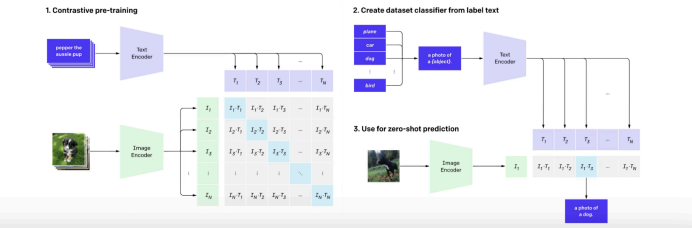
e.g. image classification (done in the contrastive way)

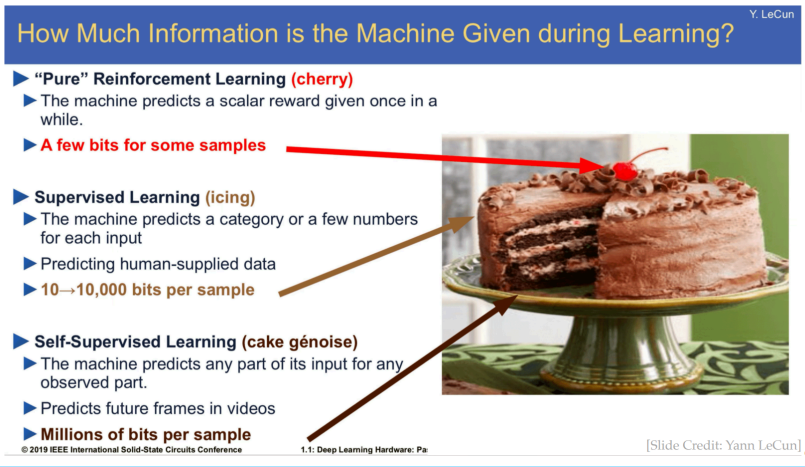



```
# extract feature representations of each modality
I_f = image_encoder(I) #[n, d_i]
T_f = text_encoder(T)  #[n, d_t]

# joint multimodal embedding [n, d_e]
I_e = l2_normalize(np.dot(I_f, W_i), axis=1)
T_e = l2_normalize(np.dot(T_f, W_t), axis=1)

# scaled pairwise cosine similarities [n, n]
logits = np.dot(I_e, T_e.T) * np.exp(t)

# symmetric loss function
labels = np.arange(n)
loss_i = cross_entropy_loss(logits, labels, axis=0)
loss_t = cross_entropy_loss(logits, labels, axis=1)
loss   = (loss_i + loss_t)/2
```

[Radford et al, Learning Transferable Visual Models From Natural Language Supervision, ICML, 2011] 63

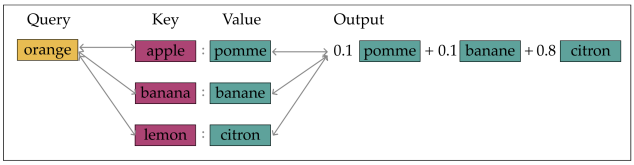
e.g. image classification (done in the contrastive way)





[Slide Credit: Yann LeCun] 66

4/6 recap: word embedding



a. compare query and key for merging percentages:

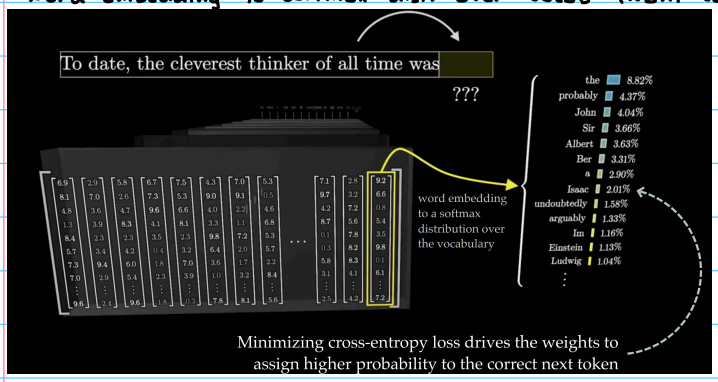
$$\text{softmax}\left(\begin{bmatrix}\text{orange}\\\text{apple}\end{bmatrix}, \begin{bmatrix}\text{orange}\\\text{banana}\end{bmatrix}, \begin{bmatrix}\text{orange}\\\text{lemon}\end{bmatrix}\right) = \begin{bmatrix}0.1 & 0.1 & 0.8\end{bmatrix}$$

b. then output mixed values 0.1 pomme + 0.1 banane + 0.8 citron

Let's see how this intuition becomes a trainable mechanism.

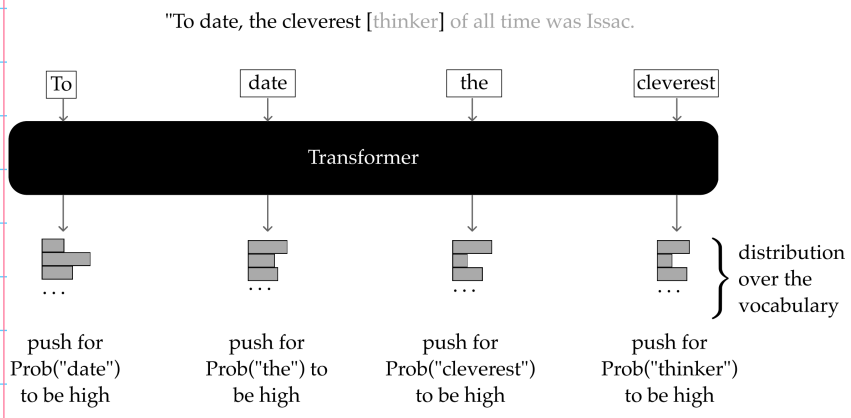
TRANSFORMERS:

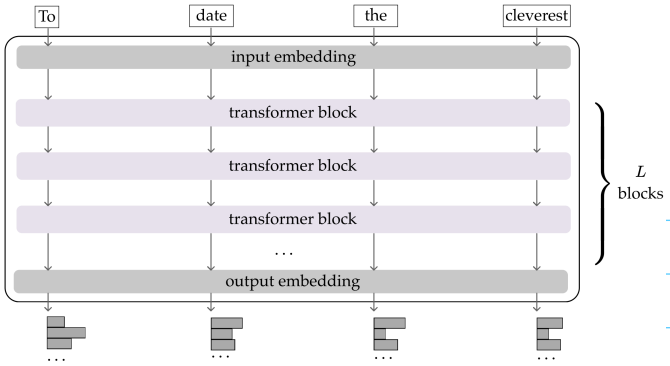
- input embedding: n small chunks, each of which w/ a mapping (Ex: GPT-2 n=768, GPT-3 n=12004)
- passed through attention layer (values internally change) then multi-layer perception
- word embedding to softmax dist. over vocab (want correct word to be assigned high prob.)

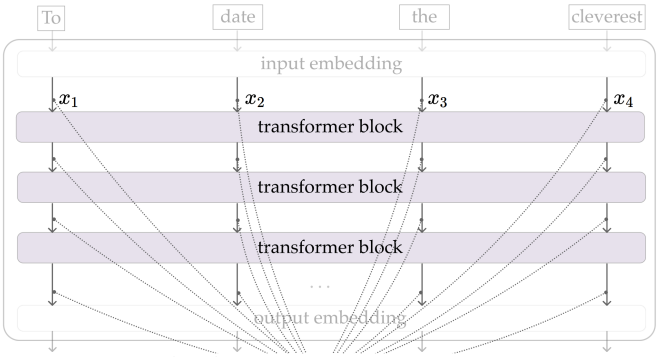


WORDS W/ MULTIPLE MEANINGS:

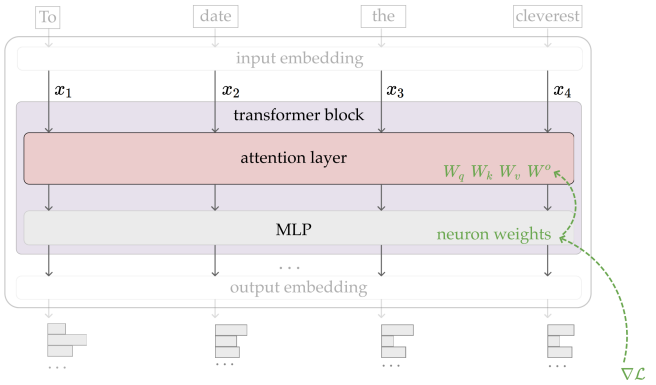
- all 3 "moles" mapped to same representation E(mole)
- look @ surrounding words & soak in context to get more specific representation

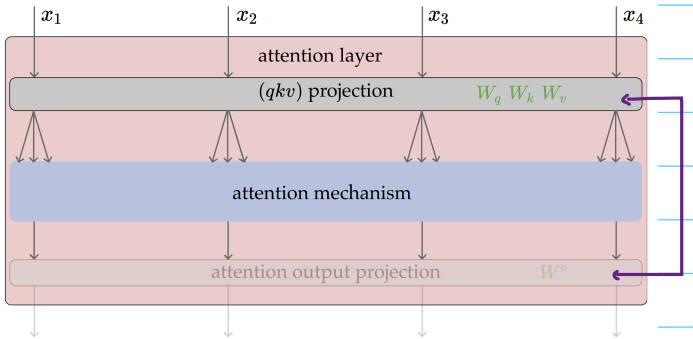






each of the n tokens transformed, block by block
within a shared d-dimensional word-embedding space.





Most important bits in an attention layer:

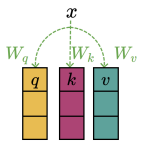
1. (query, key, value) projection
2. attention mechanism

1. (query, key, value) projection: linear transformation

1. (query, key, value) projection

With *learned* projections, we frame x into:

- a query to be the questions
- a key to be compared
- a value to contribute



Why learning these projections:

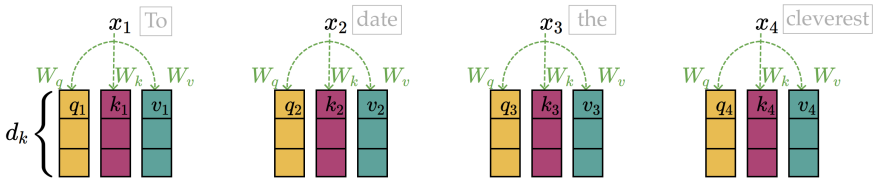
- W_q learns *how* to ask → what to ask
- W_k learns *how* to listen
- W_v learns *how* to speak → giving an answer

d: word embedding

d_k: (qkv)-embedding ← learned way of phrasing good value/response

attention head:

1. (query, key, value) projection

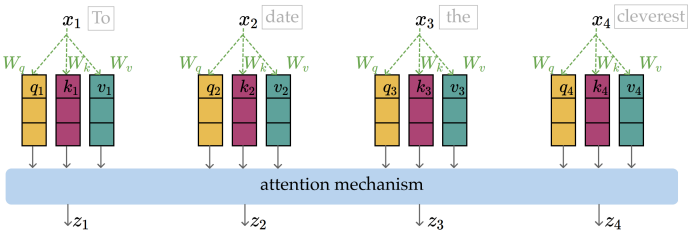


- W_q, W_k, W_v, all in $\mathbb{R}^{d \times d_k}$
- project word embedding x from d-dimensional space to d_k-dimensional (q, k, v) spaces (typically $d_k < d$)
- $q_i = W_q^T x_i,\ k_i = W_k^T x_i,\ v_i = W_v^T x_i,\ \forall i$ — ***weight sharing*** across positions
- ***parallel*** and ***structurally identical*** processing

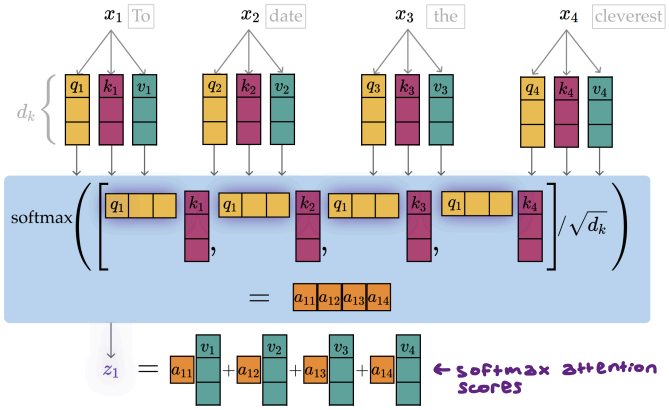
⭐ once you understand how to phrase good question, it's very reuseable

2: attention mech.

2. Attention mechanism



- Attention mechanism turns the projected (q, k, v) into z
- Each z is context-aware: a mixture of everyone's values, weighted by relevance



query comparing dot product similarity for other pairs keys

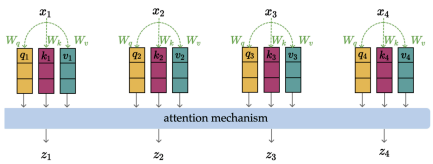
← softmax attention scores

PARALLEL & STRUCTURALLY IDENTICAL processing: can calculate z_4 w/o z_3

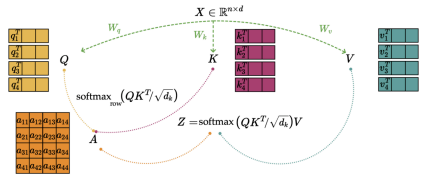
* W_q, W_k, W_v shared w/ all elements! but even though question meaning is the same, way question is asked might be slightly different

ATTENTION HEAD: compact matrix form

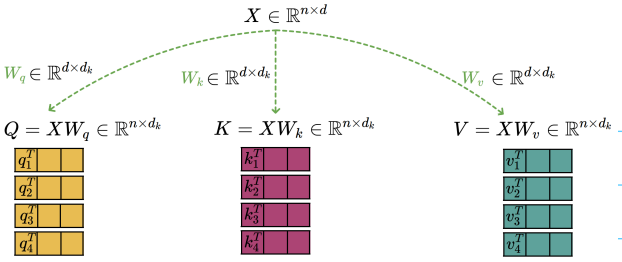
Attention head - compact matrix form



By stacking each individual vector in the sequence as a row

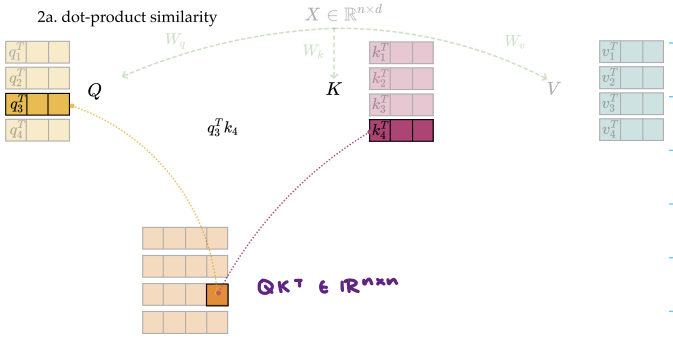


1. (query, key, value) projection



each x token maps to query token

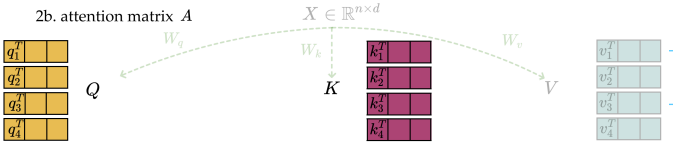
2a. dot-product similarity



row i: query
col j: keys

$QK^T \in \mathbb{R}^{n \times n}$

2b. attention matrix A

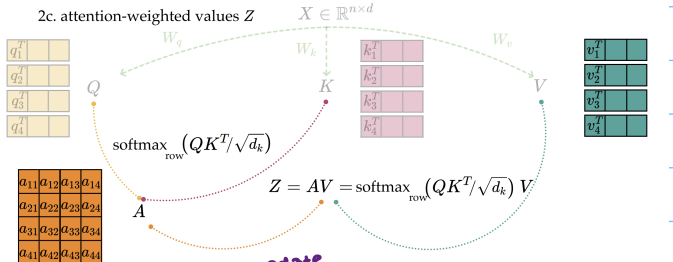


turn to distribution s.t. each row sums to 1.

$$A = \begin{bmatrix} \text{softmax}(\cdot/\sqrt{d_k}) \\ \text{softmax}(\cdot/\sqrt{d_k}) \\ \text{softmax}(\cdot/\sqrt{d_k}) \\ \text{softmax}(\cdot/\sqrt{d_k}) \end{bmatrix} = \text{softmax}_{\text{row}}\left(QK^T/\sqrt{d_k}\right) = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \in \mathbb{R}^{n \times n}$$

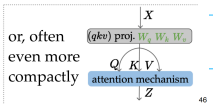
each row sums up to 1

2c. attention-weighted values Z

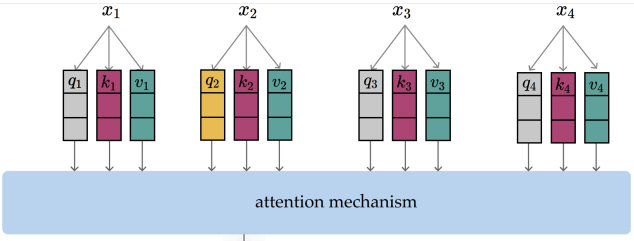


update internal representations

or, often even more compactly



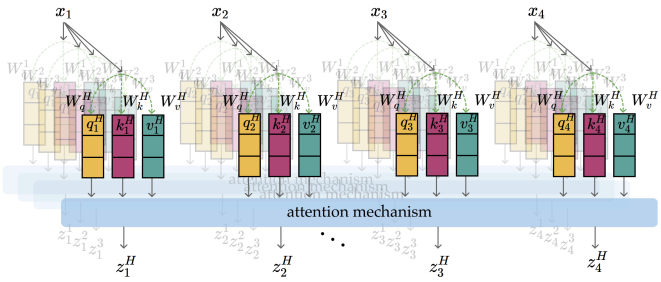
46



only care about the info you want

$$= a_{21} v_1 + a_{22} v_2 + a_{23} v_3 + a_{24} v_4$$

MULTI-HEAD ATTENTION:

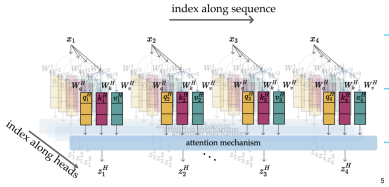


Multi-head Attention

Parallel, and ***structurally identical*** processing across all heads and tokens.

In particular, each head:

- learns its own set of W_q, W_k, W_v
- creates its own projected sequence of (q, k, v)
- computes its own sequence of z
- structurally identical processing
- for each token in the sequence:
  - structurally identical processing

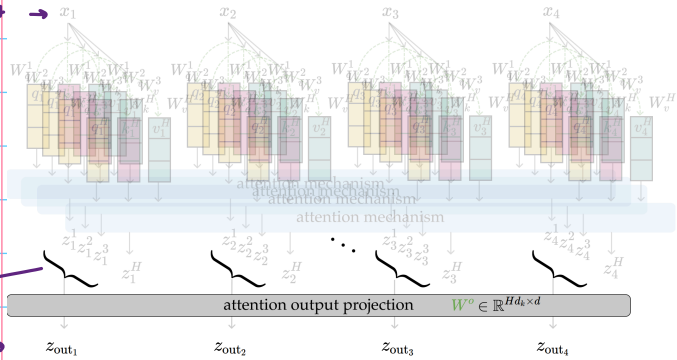


5

* big parallel-processing machine

* you can learn 8 diff. paces but still all learn from each other's questions (parallel - don't have to wait for others to learn)

input →



concatenate as z_i

output →

z_{out_1} z_{out_2} z_{out_3} z_{out_4}

Shape Example:

n	num tokens	5
d	word-embedding dim	6
d_k	(qkv) embedding dim	3
H	num heads	2

hyperparams →

X	**input**	$n \times d$	5×6	
for a single attention head:				
W_q^h	**query proj**	$d \times d_k$	6×3	
W_k^h	**key proj**	$d \times d_k$	6×3	
W_v^h	**value proj**	$d \times d_k$	6×3	
Q^h	**query**	$n \times d_k$	5×3	
K^h	**key**	$n \times d_k$	5×3	
V^h	**value**	$n \times d_k$	5×3	
A^h	**attn matrix**	$n \times n$	5×5	
Z^h	**attn head out**	$n \times d_k$	5×3	
$\text{concat}(Z^1 \dots Z^H)$	**multi-head out**	$n \times Hd_k$	5×6	
W^o	**output proj**	$Hd_k \times d$	6×6	
Z_{out}	**attn layer out**	$n \times d$	5×6	

Ws are the learned weights

↑ EXAM 2 CONTENT ↑

LECTURE 10: MARKOV DECISION PROCESS ← this + L11 create reinforcement learning

POLICY EVAL: similar to fwd pass; given smthg & evaluating quality of that thing

POLICY OPT: similar to back pass; searching for smthg of "good quality"

* finding good policy similar to finding good set of weights

S: state space, contains all possible states s

A: action space, contains all possible actions a

T(s, a, s'): probability of transition from state s to s' when action a is taken

R(s,a): reward, takes in a (state, action) pair & returns a reward

$\gamma \in [0,1]$: discount factor, a scalar

$\pi(s)$: policy, takes in state & return action

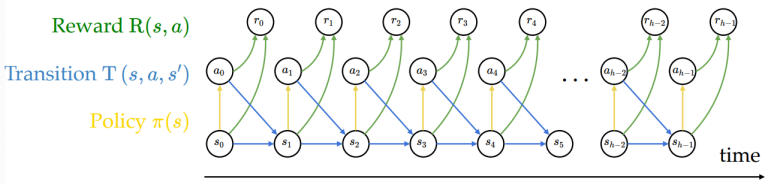
* Goal of MDP is to find good policy

in 6.390:

- S & A = small discrete sets
- s' & a' short-hand for next timestep state & action

Running example: Mario in a grid-world

- 9 possible **states** s
- 4 possible **actions** a: {Up ↑, Down ↓, Left ←, Right →}
- (state, action) results in a **transition** T into a next state:
  - Normally, we get to the "intended" state;
    - E.g., in state (7), action "↑" gets to state (4)
  - If an action would take Mario out of the grid world, stay put;
    - E.g., in state (9), "→" gets back to state (9)
  - In state (6), action "↑" leads to two possibilities:
    - 20% chance to (2)
    - 80% chance to (3).



your future only depends on your current & not your past

every step & action determines the next future step

a trajectory (also called an experience or rollout) of horizon h

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots s_{h-1}, a_{h-1}, r_{h-1})$$

initial state — all depends on π

- $a_t = \pi(s_t)$
- $r_t = \mathrm{R}(s_t, a_t)$
- $\mathrm{T}(s, a, s')$

[POLICY EVAL]

VALUE FUNC: long-term, sum & calc. expectation

Value functions:

$$\mathrm{V}_h^\pi(s) := \mathbb{E}\left[\mathrm{R}(s_0, \pi(s_0)) + \gamma \mathrm{R}(s_1, \pi(s_1)) + \gamma^2 \mathrm{R}(s_2, \pi(s_2)) + \gamma^3 \mathrm{R}(s_3, \pi(s_3)) + \ldots + \gamma^{h-1} \mathrm{R}(s_{h-1}, \pi(s_{h-1}))\right]$$
$$= \mathbb{E}\left[\sum_{t=0}^{h-1} \gamma^t \mathrm{R}(s_t, \pi(s_t)) \mid s_0 = s, \pi\right] \quad \text{(eq. 1)}$$

- $\mathrm{V}_h^\pi(s)$: expected sum of discounted rewards starting in state s and follow π for h steps
- Value is **long-term**; reward is **immediate** (one-time)
- Convention: $\mathrm{V}_0^\pi(s) = 0$ for all states
- The value expectation in 6.390 is only w.r.t. the transition probabilities $\mathrm{T}(s, a, s')$

evaluate $V_h^\pi(s)$ under the "always-up" policy

states and one special transition:

rewards

- $\pi(s) =$ "↑", $\forall s$
- $\gamma = 0.9$

$$V_h^\uparrow(s) = \mathbb{E}\left[\sum_{t=0}^{h-1} \gamma^t R(s_t, \uparrow) \mid s_0 = s\right]$$

$$= \mathbb{E}\underbrace{\left[R(s_0, \uparrow) + \gamma R(s_1, \uparrow) + \cdots + \gamma^{h-1} R(s_{h-1}, \uparrow)\right]}_{h \text{ terms}}$$

horizon $h = 0$: no step left

$V_0^\uparrow(s) = 0$

0	0	0
0	0	0
0	0	0

horizon $h = 1$: receive the rewards

$V_1^\uparrow(s) = R(s, \uparrow)$

0	0	1
0	0	−10
0	0	0

---

horizon $h = 2$

states and one special transition:

rewards

- $\pi(s) =$ "↑", $\forall s$
- $\gamma = 0.9$

$$V_2^\uparrow(s) : \mathbb{E}\underbrace{\left[R(s_0, \uparrow) + \gamma R(s_1, \uparrow)\right]}_{2 \text{ terms}}$$

$R(1, \uparrow) + \gamma R(1, \uparrow)$ $\quad$ $R(2, \uparrow) + \gamma R(2, \uparrow)$ $\quad$ $R(3, \uparrow) + \gamma R(3, \uparrow)$ $= 1 + 0.9 * (1) \Rightarrow 1.9$

$R(4, \uparrow) + \gamma R(1, \uparrow)$

0	0	1.9
0	0	

$R(5, \uparrow) + \gamma R(2, \uparrow)$

---

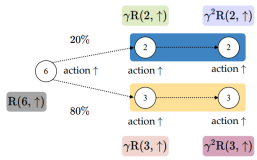
horizon $h = 3$

states and one special transition:

rewards

- $\pi(s) =$ "↑", $\forall s$
- $\gamma = 0.9$

$$V_3^\uparrow(s) : \mathbb{E}\underbrace{\left[R(s_0, \uparrow) + \gamma R(s_1, \uparrow) + \gamma^2 R(s_2, \uparrow)\right]}_{3 \text{ terms}}$$



$$V_3^\uparrow(6) = R(6, \uparrow) + 20\%\ \gamma R(2, \uparrow) + 80\%\ \gamma R(3, \uparrow) + 20\%\ \gamma^2 R(2, \uparrow) + 80\%\ \gamma^2 R(3, \uparrow)$$

$$= R(6, \uparrow) + 20\%\ [\ \gamma R(2, \uparrow) + \gamma^2 R(2, \uparrow)\ ] + 80\%\ [\ \gamma R(3, \uparrow) + \gamma^2 R(3, \uparrow)\ ]$$

$$= R(6, \uparrow) + 20\%\ \gamma\ [R(2, \uparrow) + \gamma R(2, \uparrow)] + 80\%\ \gamma\ [R(3, \uparrow) + \gamma R(3, \uparrow)]$$

$$= R(6, \uparrow) + 20\%\ \gamma\ V_2^\uparrow(2) + 80\%\ \gamma\ V_2^\uparrow(3)$$

$$V_3^\uparrow(6) = R(6, \uparrow) + 20\%\ \gamma\ V_2^\uparrow(2) + 80\%\ \gamma\ V_2^\uparrow(3)$$

horizon-h value in state s: the expected sum of discounted rewards, starting in state s and following policy π for h steps.

(eq. 2) $$V_h^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s')\, V_{h-1}^\pi(s')$$

group tail portions & recursively

the immediate reward for taking the policy-prescribed action $\pi(s)$ in state s.

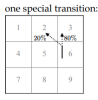
$(h-1)$ horizon future value at a next state s'

sum of future values weighted by the probability of reaching that next state s'

discounted by γ

Value functions converge as $h \to \infty$

states and one special transition:



rewards

- $\pi(s) =$ "↑", $\forall s$
- $\gamma = 0.9$

$V_{60}^\uparrow(s)$

0	0	9.98
0	0	-2.81
0	0	-2.53

$V_{61}^\uparrow(s)$

0	0	9.98
0	0	-2.81
0	0	-2.53

$\cdots$

$V_\infty^\uparrow(s)$

0	0	10
0	0	-2.8
0	0	-2.52

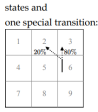
- As we extend the horizon, value differences shrink
- because longer-term rewards are heavily discounted
- so, as $h \to \infty$, the value functions stop changing
- convergence can be seen, e.g., via $V_\infty^\uparrow(3) = 1 + .9 + .9^2 + .9^3 + \cdots = 10$

Typically, $\gamma < 1$ to ensure V_∞ is finite.

die down to static vals

As horizon $h \to \infty$, the Bellman recursion becomes the Bellman equation

states and one special transition:



rewards

- $\pi(s) =$ "↑", $\forall s$
- $\gamma = 0.9$

Recursion (finite h) 2 $V_h^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s')\, V_{h-1}^\pi(s')$

Equation ($h \to \infty$) 3 $V_\infty^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s')\, V_\infty^\pi(s')$

A system of $|\mathcal{S}|$ self-consistent linear equations, one for each state

$V_\infty^\uparrow(s)$

0	0	10
0	0	-2.8
0	0	-2.52

$V_\infty^\uparrow(3) = R(3, \uparrow) + \gamma[V_\infty^\uparrow(3)]$
$= 1 + .9 \times 10 \Rightarrow 10$

$V_\infty^\uparrow(6) = R(6, \uparrow) + \gamma[.2 \times V_\infty^\uparrow(2) + .8 \times V_\infty^\uparrow(3)]$
$= -10 + .9[.2 \times 0 + 0.8 \times 10] \Rightarrow -2.8$

Quick summary $\pi(s) \rightarrow$ Policy Evaluation $\rightarrow V_h^\pi(s)$

Use the definition and sum up expected rewards:

1 $V_h^\pi(s) := \mathbb{E}\left[\sum_{t=0}^{h-1} \gamma^t R(s_t, \pi(s_t)) \mid s_0 = s, \pi\right]$

Or, leverage the recursive structure:

2 **finite**-horizon Bellman recursions

$$V_h^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_{h-1}^\pi(s')$$

3 **infinite**-horizon Bellman equations

$$V_\infty^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_\infty^\pi(s')$$

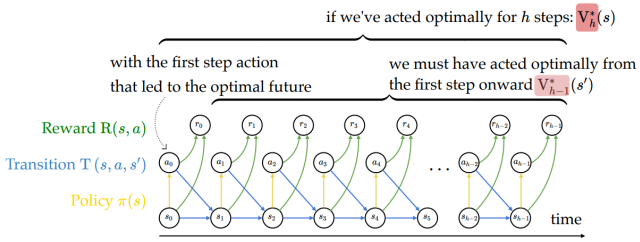
[POLICY OPTIMIZATION]

OPTIMAL POLICY π^*: POLICY THAT YIELDS HIGHEST POSSIBLE VALUE $V_h^*(s)$ FROM EVERY STATE

- might not be unique

$V_h^{\pi^*}(s) = \max_\pi V_h^\pi(s) = V_h^*(s), \forall s \in S$

- $V^*(s)$ defined over states, not actions



$V_h^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_{h-1}^\pi(s')$ (recall, eq. 2, for any policy)

$V_h^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} T(s,a,s') V_{h-1}^*(s')\right]$ (new, eq. 4, for an optimal policy)

$$V_h^*(s) = \max_a \underbrace{\left[R(s,a) + \gamma \sum_{s'} T(s,a,s') V_{h-1}^*(s')\right]}_{Q_h^*(s,a)} \quad \text{(eq. 4)}$$

Define the optimal state-action value functions $Q_h^*(s,a)$:

the expected sum of discounted rewards, obtained by

- starting in state s
- take action a, for one step
- act **optimally** thereafter for the remaining $(h-1)$ steps

Q^* satisfies:

$$Q_h^*(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q_{h-1}^*(s',a') \quad \text{(eq. 5)}$$

$Q_h^*(s,a)$: the value for

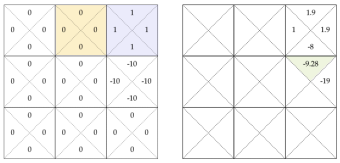
- starting in state s,
- take action a, for one step
- act **optimally** thereafter for the remaining $(h-1)$ steps

States and one special transition: 20% 80%

Rewards:

$\gamma = 0.9$

$Q_1^*(s,a)$ $Q_2^*(s,a)$



Let's consider $Q_2^*(6, \uparrow)$

- receive $R(6, \uparrow)$
- act **optimally** at the next state s'
  - 20% chance, $s' = 2$, act optimally, get $\max_{a'} Q_1^*(2, a')$
  - 80% chance, $s' = 3$, act optimally, get $\max_{a'} Q_1^*(3, a')$

$$Q_2^*(6, \uparrow) = R(6, \uparrow) + \gamma[.2 \max_{a'} Q_1^*(2, a') + .8 \max_{a'} Q_1^*(3, a')]$$
$$= -10 + .9[.2 \times 0 + .8 \times 1] \Rightarrow -9.28$$

Value iteration: what we just did, iteratively invoke (eq. 5):

$$Q_h^*(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q_{h-1}^*(s',a')$$

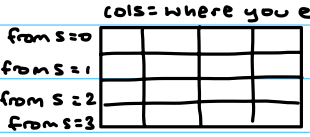
```
Value Iteration
1. for s ∈ S, a ∈ A:
2.     Q_old(s, a) = 0
3. while True:
4.     for s ∈ S, a ∈ A:
5.         Q_new(s, a) ← R(s,a) + γ Σ_{s'} T(s,a,s') max_{a'} Q_old(s', a')
6.     if max_{s,a} |Q_old(s, a) − Q_new(s, a)| < ε:
7.         return Q_new
8.     Q_old ← Q_new
```

if run this block h times and break, then the returns are exactly Q_h^*

$Q_\infty^*(s,a)$

T(s, b, s') - action b matrix:

cols: where you end up (s')

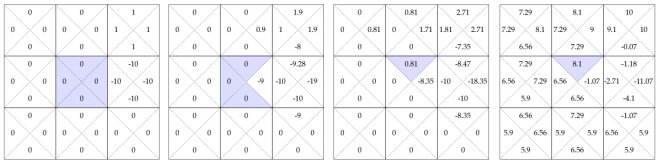


whole row: from state 0, take action b

Optimal policy easily extracted: 6 $\pi_h^*(s) = \arg\max_a Q_h^*(s,a)$

e.g. the best actions to take in state 5

$Q_1^*(s,a)$ $Q_2^*(s,a)$ $Q_3^*(s,a)$ ... $Q_\infty^*(s,a)$



- For finite h, optimal policy π_h^* depends on how many time steps are left
- When $h \to \infty$, time no longer matters, i.e., there exists a stationary π^*

Summary

- A Markov decision process $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$ is the mathematical framework for sequential decision-making and the foundation of reinforcement learning.
- To evaluate a *given* policy π, we compute state value functions $V^\pi(s)$ via the Bellman recursion (finite horizon) or the Bellman equation (infinite horizon).
- To *find* an optimal policy, we compute $Q^*(s,a)$ via the value iteration algorithm, then act greedily: $\pi^*(s) = \arg\max_a Q^*(s,a)$.

4/27 **REINFORCEMENT LEARNING**

$\pi(s)$ → Policy Evaluation → $V_h^\pi(s)$

Use the definition and sum up expected rewards:

1 $V_h^\pi(s) := \mathbb{E}\left[\sum_{t=0}^{h-1} \gamma^t R(s_t, \pi(s_t)) \mid s_0 = s, \pi\right]$

Or, leverage the recursive structure:

2 **finite-**horizon Bellman recursions

$$V_h^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_{h-1}^\pi(s')$$

3 **infinite-**horizon Bellman equations

$$V_\infty^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_\infty^\pi(s')$$

Recall

horizon-h value in state s: the expected sum of discounted rewards, starting in state s and following policy π for h steps.

2 $V_h^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_{h-1}^\pi(s')$

the immediate reward for taking the policy-prescribed action $\pi(s)$ in state s.

$(h-1)$ horizon future value at a next state s'

sum of future values weighted by the probability of reaching that next state s'

discounted by γ

Model-based RL: Learn the MDP tuple

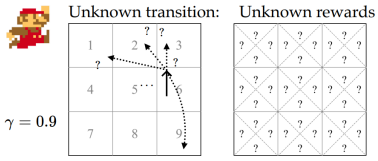
1. Collect experiences (s, a, r, s')

2. Estimate $\hat{\mathrm{T}}$, $\hat{\mathrm{R}}$

e.g. $\hat{\mathrm{T}}(6, \uparrow, 2) \approx \frac{\text{observed } (6,\uparrow,2)}{\text{total}(6,\uparrow) \text{ play count}}$

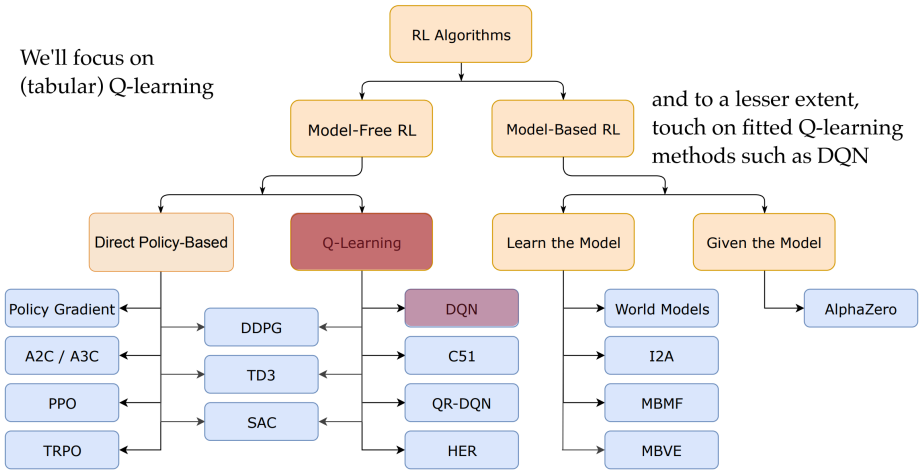
e.g. $\hat{\mathrm{R}}(6, \uparrow) =$ observed reward received from $(6, \uparrow)$

3. Solve $\langle \mathcal{S}, \mathcal{A}, \hat{\mathrm{T}}, \hat{\mathrm{R}}, \gamma \rangle$ via e.g. *Value Iteration*



(s, a)	r	s'
$(1, \uparrow)$	0	1
$(1, \downarrow)$	0	4
...		
$(3, \uparrow)$	1	3
$(3, \downarrow)$	1	6
...		
$(6, \uparrow)$	-10	3
$(6, \uparrow)$	-10	2
...		

We'll focus on (tabular) Q-learning



and to a lesser extent, touch on fitted Q-learning methods such as DQN

[A non-exhaustive, but useful taxonomy of algorithms in modern RL. Source]

TABULAR Q-LEARNING:

·once we have Q values, we basically have everything we need to get optimal policy

- Indeed, *value iteration* relies on having full access to R and T

$$\mathrm{Q}_{\text{new}}(s, a) \leftarrow \mathrm{R}(s, a) + \gamma \sum_{s'} \mathrm{T}(s, a, s') \max_{a'} \mathrm{Q}_{\text{old}}(s', a')$$

R(s,a): immediate reward

$\max_{a'} \mathrm{Q}_{\text{old}}(s', a')$: future value, starting in state s' and acting ***optimally*** for $(h-1)$ steps

$\sum_{s'} \mathrm{T}(s, a, s')$: expected future value, weighted by the chance of landing in that particular future state s'

- Without R and T, how about: execute (s, a), observe r and s', and update:

$$\mathrm{Q}_{\text{new}}(s, a) \leftarrow r + \gamma \max_{a'} \mathrm{Q}_{\text{old}}(s', a')$$

target

(we'll see why this fails)

$$\mathrm{Q}_{\text{new}}(s, a) \leftarrow r + \gamma \max_{a'} \mathrm{Q}_{\text{old}}(s', a')$$

target

Whenever observe $(6, \uparrow)$, -10, 3:

$$\mathrm{Q}_{\text{new}}(6, \uparrow) \leftarrow -10 + \gamma \max_{a'} \mathrm{Q}_{\text{old}}(3, a')$$

$$= -10 + 0.9 * 1 = -9.1$$

Whenever observe $(6, \uparrow)$, -10, 2:

$$\mathrm{Q}_{\text{new}}(6, \uparrow) \leftarrow -10 + \gamma \max_{a'} \mathrm{Q}_{\text{old}}(2, a')$$

$$= -10 + 0.9 * 0 = -10$$

$\gamma = 0.9$

(s, a)	r	s'
...		
$(6, \uparrow)$	-10	3
$(6, \uparrow)$	-10	2
$(6, \uparrow)$	-10	3
$(6, \uparrow)$	-10	2
...		

Simply committing to the new target keeps "washing away" the old belief

😍 merge *old belief* and *target*

$$(1-\text{learning rate})\ \text{old belief} + \text{learning rate}\ \text{target}$$

$$Q_{new}(s,a) \leftarrow (1-\alpha)\, Q_{old}(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{old}(s',a')\right)$$

← downweigh w/ α and add in old belief

- Core update rule of *Q-learning*
- $\alpha \in [0,1]$: hyperparameter, trades off how much we trust new evidence versus old belief
- What we saw was $\alpha = 1$: trust new experience entirely
- Let's see an example using $\alpha = 0.7$

$$Q_{new}(s,a) \leftarrow (1-\alpha) Q_{old}(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{old}(s',a')\right)$$

- To update a particular Q(s, a), we need experience from actually taking that a in that s.
- If we never try a move, its Q-value never changes.
- So if we have one play left, which (s,a) should we try?

Depends on

- are we trying to "win" the game, or
- are we trying to get more accurate Q estimates

This is the fundamental dilemma in reinforcement learning... and in life!
Whether to *exploit* what we've already learned or *explore* to discover something better.

ε-GREEDY ACTION SELECTION STRATEGY:

- with probability ε, choose action $a \in A$ uniformly @ random
  - during learning, esp. in early stages, explore and observe diverse (s,a) consequences
- with probability (1-ε), choose $\text{argmax}_a Q_{old}(s,a)$
  - during later stages, we can act more greedily w/ respect to current estimated Q values

* in beginning: ε bigger (more exploratory), but smaller later (more greedy)
* ε trades off exploration vs. exploitation

Value Iteration $(\mathcal{S}, \mathcal{A}, T, R, \gamma, \epsilon)$

1. **for** $s \in \mathcal{S}, a \in \mathcal{A}$:
2. $\quad Q_{old}(s,a) = 0$
3. **while** True:
4. $\quad$ **for** $s \in \mathcal{S}, a \in \mathcal{A}$:
5. $\quad\quad Q_{new}(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q_{old}(s',a')$
6. $\quad$ **if** $\max_{s,a} |Q_{old}(s,a) - Q_{new}(s,a)| < \epsilon$:
7. $\quad\quad$ **return** Q_{new}
8. $\quad Q_{old} \leftarrow Q_{new}$

"calculating"

Q-Learning $(\mathcal{S}, \mathcal{A}, \gamma, \alpha, s_0,$ max-iter$)$

1. $i = 0$
2. **for** $s \in \mathcal{S}, a \in \mathcal{A}$:
3. $\quad Q_{old}(s,a) = 0$
4. $s \leftarrow s_0$
5. **while** $i <$ max-iter :
6. $\quad a \leftarrow$ select_action$(s, Q_{old}(s,a))$
7. $\quad r, s' \leftarrow$ execute(a)
8. $\quad Q_{new}(s,a) \leftarrow (1-\alpha) Q_{old}(s,a) + \alpha(r + \gamma \max_{a'} Q_{old}(s',a'))$
9. $\quad s \leftarrow s'$
10. $\quad i \leftarrow (i+1)$
11. $\quad Q_{old} \leftarrow Q_{new}$
12. **return** Q_{new}

"learning" (estimating)

Q-Learning $(\mathcal{S}, \mathcal{A}, \gamma, \alpha, s_0,$ max-iter$)$

1. $i = 0$
2. **for** $s \in \mathcal{S}, a \in \mathcal{A}$:
3. $\quad Q_{\text{old}}(s,a) = 0$
4. $s \leftarrow s_0$
5. **while** $i <$ max-iter :
6. $\quad a \leftarrow$ select_action$(s, Q_{\text{old}}(s,a))$
7. $\quad r, s' \leftarrow$ execute(a)
8. $\quad Q_{\text{new}}(s,a) \leftarrow (1-\alpha)Q_{\text{old}}(s,a) + \alpha(r + \gamma \max_{a'} Q_{\text{old}}(s',a'))$
9. $\quad s \leftarrow s'$
10. $\quad i \leftarrow (i+1)$
11. $\quad Q_{\text{old}} \leftarrow Q_{\text{new}}$
12. **return** Q_{new}

"learning"

- Remarkably, 👈 *can* converge to the true Q^*_∞
- Once converged, act greedily w.r.t Q^* again.
- But convergence can be extremely slow,

  and often not practical.
- Three hyperparameters :
  - discount factor γ
  - learning rate α
  - exploration rate ϵ

all between 0 and 1

each controls some trade-off

[1] given that we visit all s, a infinitely often, and satisfy a decaying condition on the learning rate α.

FITTED Q-LEARNING:

· Q learning for large / continuous S & A

· we can't keep a table, must approx. Q(s,a) w/ function $Q_\theta(s,a)$

- Notice that the core update rule of Q-learning:

$$Q_{\text{new}}(s,a) \leftarrow (1-\alpha)Q_{\text{old}}(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{\text{old}}(s',a')\right)$$

is equivalent to:

$$Q_{\text{new}}(s,a) \leftarrow Q_{\text{old}}(s,a) + \alpha([r + \gamma \max_{a'} Q_{\text{old}}(s',a')] - Q_{\text{old}}(s,a))$$

- Does this

$$\text{new belief} \leftarrow \text{old belief} + \text{learning rate}\ (\text{target} - \text{old belief})$$

remind you of something?

- Yes! Recall:

$$\theta_{\text{new}} \leftarrow \theta_{\text{old}} + \eta(\text{target} - \text{guess}_\theta)\nabla_\theta \text{guess}$$

Gradient update rule when minimizing $(\text{target} - \text{guess}_\theta)^2$

Q-learning is basically some version of gradient descent

- Same idea as before — we adjust our belief toward a target.

$$Q_{\text{new}}(s,a) \leftarrow Q_{\text{old}}(s,a) + \alpha([r + \gamma \max_{a'} Q_{\text{old}}(s',a')] - Q_{\text{old}}(s,a))$$

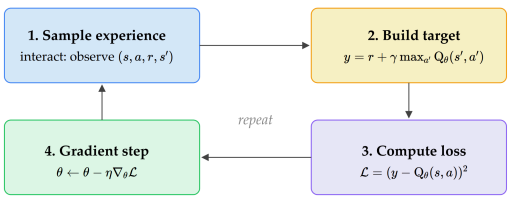
$$\text{new belief} \leftarrow \text{old belief} + \text{learning rate}\ (\text{target} - \text{old belief})$$

- Now Q is a function (e.g. a neural network) and θ are its weights.
- Generalize tabular Q-learning for continuous state / action space:

1. parameterize $Q_\theta(s,a)$
2. execute (s,a), observe(r,s'), construct the target $r + \gamma \max_{a'} Q_\theta(s',a')$
3. regress $Q_\theta(s,a)$ against the target, i.e. update θ via gradient-descent to minimize

$$(\text{target} - Q_\theta(s,a))^2$$

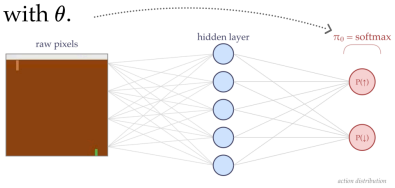
Fitted Q-learning: the training loop



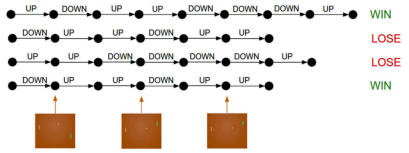
Same shape as supervised learning; except the "label" y is built from our own current estimate. (Bootstrapping.)

Policy gradients:

1. Parametrize the policy π with θ.



2. Run π_θ for a while. Keep track of the return.



3. Update θ so "good" trajectories more likely to happen.

[image edited from: Andrej Karpathy]

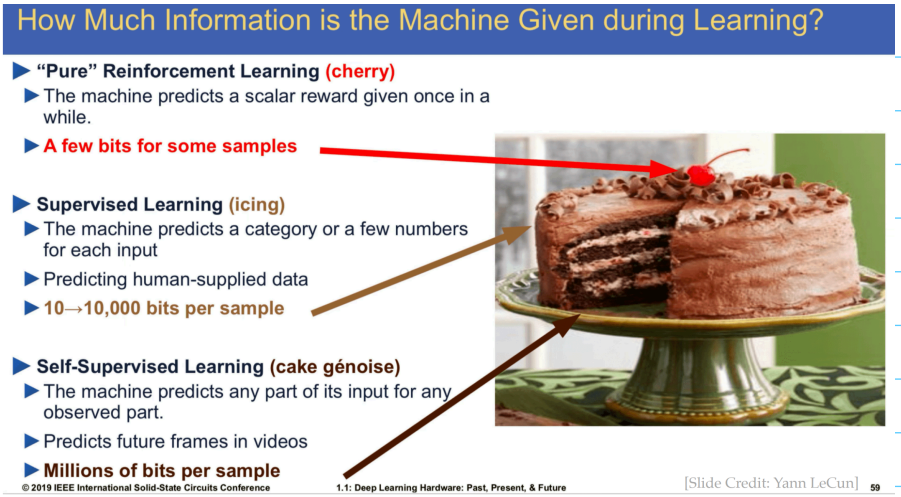
Made rigorous by the *log-derivative trick* (REINFORCE). We will skip the derivation.

trajectory drawn under the current policy — *sum over timesteps*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot R(\tau)\right]$$

*push θ to make the taken action **more likely*** — *weight by how good the trajectory was*

Sample trajectories, observe returns, nudge θ to make actions in *good* trajectories more likely.



[Slide Credit: Yann LeCun]

Summary

- Last week: find good policies in a *known* MDP (high cumulative expected reward).
- In RL, the transitions and rewards are *unknown*. We still want a good policy; we get one by estimating Q.
- Exploration vs. exploitation: choose actions to gain reward while still learning.
- Q-learning converges to Q* (with enough exploration).
- Deep Q-learning extends to large or continuous state spaces by parameterizing Q.

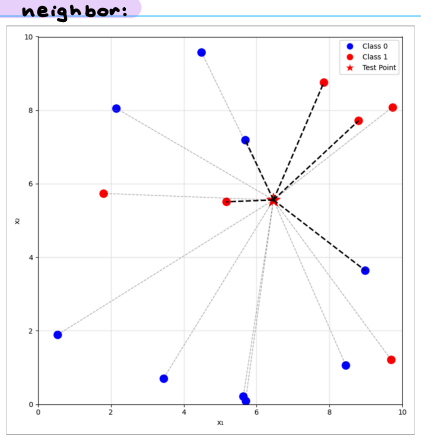
lecture 12 NON-PARAMETRIC MODELS: model doesn't assume a fixed functional form for h; complexity grows with data

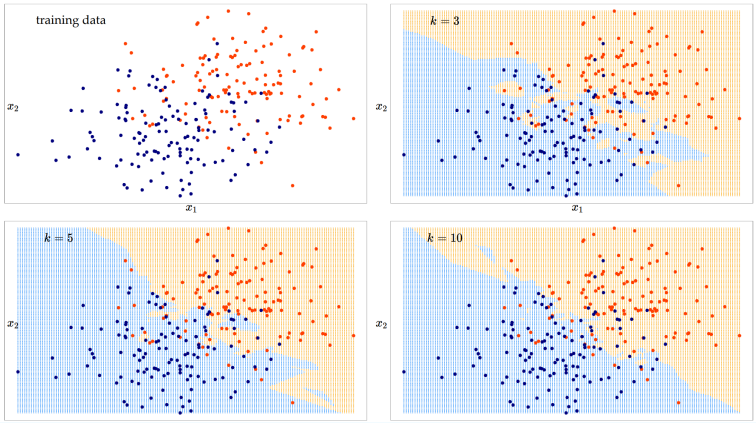
- interpretability (intuitive)
- insight
- speed
- adaptivity

SIMILARITY-BASED METHODS:

K-nearest neighbor:

prediction on new data using $k = 5$ nearest neighbor





be CAUTIOUS when:

- large n or d (curse of dimensionality)
- many irrelevant/noisy features
- features on diff. scales

Training: none. Just memorize the training set.

Predicting: for a new x_{new}, take the majority (class) or mean (regression) label of its k nearest neighbors.

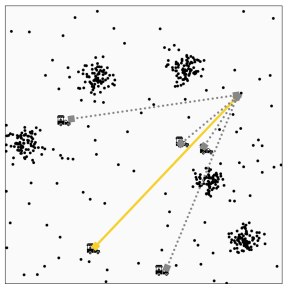
Parameters learned: the entire training set (its features and labels).
becomes params

Hyperparameters:

- k: number of neighbors considered.
- distance metric (typically Euclidean or Manhattan).
- tie-breaking scheme (typically at random).

K means clustering: Structural discovery w/o labels

Per-person loss



Cost for person i to walk to truck j:

$$\|x^{(i)} - \mu^{(j)}\|^2$$

Let $y^{(i)}$ be the truck person i walks to.

$\mathbf{1}\{y^{(i)} = j\}$ is 1 if assigned, 0 otherwise.

Person i's loss:

$$\sum_{j=1}^{k} \mathbf{1}\{y^{(i)} = j\} \|x^{(i)} - \mu^{(j)}\|^2$$

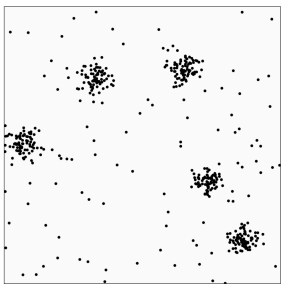
k-means objective:

cluster membership — centroid location

$$\sum_{i=1}^{n} \sum_{j=1}^{k} \mathbf{1}\left\{y^{(i)} = j\right\} \left\|x^{(i)} - \mu^{(j)}\right\|^2$$

sum over data points — sum over cluster

what we learn



K-MEANS$(k, \tau, \{x^{(i)}\}_{i=1}^{n})$

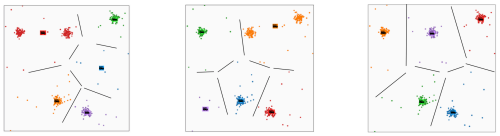
1 $\mu, y =$ random initialization (rand. assign)
2 **for** $t = 1$ to τ
3 $\quad y_{\text{old}} = y$
4 $\quad$ **for** $i = 1$ to n
5 $\quad\quad y^{(i)} = \arg\min_j \left\|x^{(i)} - \mu^{(j)}\right\|^2$
6 $\quad$ **for** $j = 1$ to k
7 $\quad\quad \mu^{(j)} = \frac{1}{N_j} \sum_{i=1}^{n} \mathbf{1}\left(y^{(i)} = \mathrm{j}\right) x^{(i)}$
8 $\quad$ **if** $y == y_{\text{old}}$
9 $\quad\quad$ break
10 return μ, y

check until loco. chgs

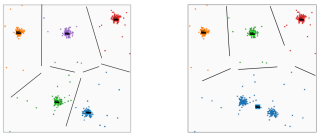
cluster to/from

k-means is sensitive to initialization and to k

same data, three different initializations:



same data, two different choices of k:



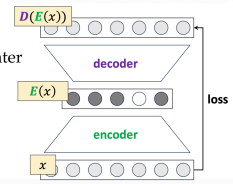
k-means interpreted as a discrete autoencoder

cluster center:

- not actual data points in MNIST
- learned representation and prototypes

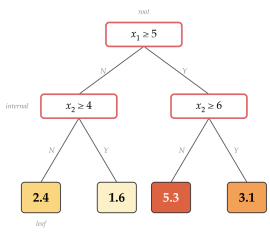
map the one-hot to a center

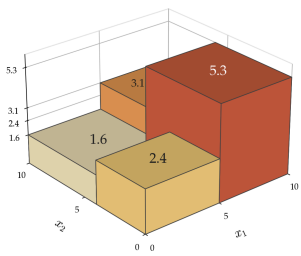
map x to a one-hot



$$\min_{D,E} \sum_{x} \|x - D(E(x))\|^2$$

Tree root node, internal node, leaf, and depth



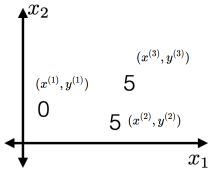


Each internal node carries a split dimension and split value ($x_j \geq \theta$).
Each leaf corresponds to an axis-aligned region; the tree predicts the leaf's value there.

tree-based method:

BuildTree$(I, k, \mathcal{D})$

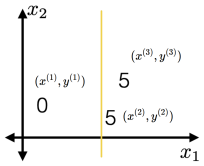
1. **if** $|I| > k$
2. **for** each split dim j and split value s
3. Set $I_{j,s}^{+} = \left\{ i \in I \mid x_j^{(i)} \geq s \right\}$
4. Set $I_{j,s}^{-} = \left\{ i \in I \mid x_j^{(i)} < s \right\}$
5. Set $\hat{y}_{j,s}^{+} = \text{average}_{\, i \in I_{j,s}^{+}} \, y^{(i)}$ — averaging pred.
6. Set $\hat{y}_{j,s}^{-} = \text{average}_{\, i \in I_{j,s}^{-}} \, y^{(i)}$ — averaging pred.
7. Set $E_{j,s} = \sum_{i \in I_{j,s}^{+}} \left(y^{(i)} - \hat{y}_{j,s}^{+} \right)^2 + \sum_{i \in I_{j,s}^{-}} \left(y^{(i)} - \hat{y}_{j,s}^{-} \right)^2$
8. Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$
9. **else**
10. Set $\hat{y} = \text{average}_{\, i \in I} \, y^{(i)}$
11. **return** Leaf(leave_value=$\hat{y}$)
12. **return** Node $\left(j^*, s^*, \text{BuildTree}\left(I_{j^*,s^*}^{-}, k \right), \text{BuildTree}\left(I_{j^*,s^*}^{+}, k \right) \right)$ — recursion →



- Choose $k = 2$
- BuildTree$(\{1, 2, 3\}; 2)$
- Line 1 true

BuildTree$(I, k, \mathcal{D})$

1. if $|I| > k$
2. **for** each split dim j and split value s
3. Set $I_{j,s}^{+} = \left\{ i \in I \mid x_j^{(i)} \geq s \right\}$
4. Set $I_{j,s}^{-} = \left\{ i \in I \mid x_j^{(i)} < s \right\}$
5. Set $\hat{y}_{j,s}^{+} = \text{average}_{\, i \in I_{j,s}^{+}} \, y^{(i)}$
6. Set $\hat{y}_{j,s}^{-} = \text{average}_{\, i \in I_{j,s}^{-}} \, y^{(i)}$
7. Set $E_{j,s} = \sum_{i \in I_{j,s}^{+}} \left(y^{(i)} - \hat{y}_{j,s}^{+} \right)^2 + \sum_{i \in I_{j,s}^{-}} \left(y^{(i)} - \hat{y}_{j,s}^{-} \right)^2$
8. Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$
9. **else**
10. Set $\hat{y} = \text{average}_{\, i \in I} \, y^{(i)}$
11. **return** Leaf(leave_value=$\hat{y}$)
12. **return** Node $\left(j^*, s^*, \text{BuildTree}\left(I_{j^*,s^*}^{-}, k \right), \text{BuildTree}\left(I_{j^*,s^*}^{+}, k \right) \right)$



- For this fixed (j, s)
  - $I_{j,s}^{+} = \{2, 3\}$
  - $I_{j,s}^{-} = \{1\}$
  - $\hat{y}_{j,s}^{+} = 5$
  - $\hat{y}_{j,s}^{-} = 0$
  - $E_{j,s} = 0$

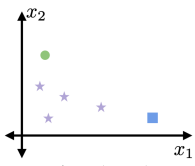
CLASSIFICATION TREE:

How to Grow a Classification Tree

1. Grow — start with all data at the root.
   Try candidate splits along each feature.
2. Split — pick the split that best separates the classes
   (lowest weighted entropy or most "homogeneous").
3. Recurse — treat each child region as a whole new dataset and run the algorithm again.
4. Stop — if a region is small or already pure
   (data within ≤ leaf size, or most data share the same label), make it a leaf.

entropy $H := -\sum_{\text{class } c} \hat{P}_c (\log_2 \hat{P}_c)$ — empirical probability

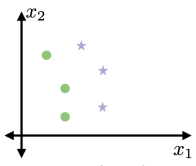
for example: c iterates over 3 classes ★ ● ■



$\hat{P}_c$: ★ $\frac{4}{6}$ ● $\frac{1}{6}$ ■ $\frac{1}{6}$

$H = -[\frac{4}{6}\log_2(\frac{4}{6}) + \frac{1}{6}\log_2(\frac{1}{6}) + \frac{1}{6}\log_2(\frac{1}{6})]$

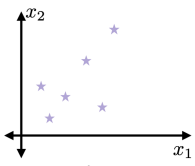
(about 1.252)



$\hat{P}_c$: ★ $\frac{3}{6}$ ● $\frac{3}{6}$ ■ 0

$H = -[\frac{3}{6}\log_2(\frac{3}{6}) + \frac{3}{6}\log_2(\frac{3}{6}) + 0]$

(about 1.1)



$\hat{P}_c$: ★ $\frac{6}{6}$ ● 0 ■ 0

$H = -[\frac{6}{6}\log_2(\frac{6}{6}) + 0 + 0]$

(= 0)

← entropy is lower for more pure (less muddy) data

For classification

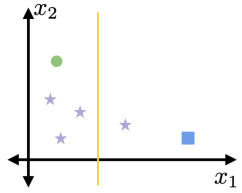
majority vote as regional prediction

BuildTree$(I, k, \mathcal{D})$

1. **if** $|I| > k$
2. **for** each split dim j and split value s
3. Set $I_{j,s}^{+} = \left\{ i \in I \mid x_j^{(i)} \geq s \right\}$
4. Set $I_{j,s}^{-} = \left\{ i \in I \mid x_j^{(i)} < s \right\}$
5. Set $\hat{y}_{j,s}^{+} = \text{majority}_{\, i \in I_{j,s}^{+}} \, y^{(i)}$
6. Set $\hat{y}_{j,s}^{-} = \text{majority}_{\, i \in I_{j,s}^{-}} \, y^{(i)}$
7. Set $E_{j,s} = \frac{|I_{j,s}^{-}|}{|I|} \cdot H\left(I_{j,s}^{-}\right) + \frac{|I_{j,s}^{+}|}{|I|} \cdot H\left(I_{j,s}^{+}\right)$ ← weighted average entropy (WAE) as performance metric
8. Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$
9. **else**
10. Set $\hat{y} = \text{majority}_{\, i \in I} \, y^{(i)}$
11. **return** Leaf(leave_value=$\hat{y}$)
12. **return** Node $\left(j^*, s^*, \text{BuildTree}\left(I_{j^*,s^*}^{-}, k \right), \text{BuildTree}\left(I_{j^*,s^*}^{+}, k \right) \right)$

BuildTree$(I, k, \mathcal{D})$

1. **if** $|I| > k$
2. **for** each split dim j and split value s
3. Set $I_{j,s}^{+} = \left\{ i \in I \mid x_j^{(i)} \geq s \right\}$
4. Set $I_{j,s}^{-} = \left\{ i \in I \mid x_j^{(i)} < s \right\}$
5. Set $\hat{y}_{j,s}^{+} = \text{majority}_{\, i \in I_{j,s}^{+}} \, y^{(i)}$
6. Set $\hat{y}_{j,s}^{-} = \text{majority}_{\, i \in I_{j,s}^{-}} \, y^{(i)}$
7. Set $E_{j,s} = \frac{|I_{j,s}^{-}|}{|I|} \cdot H\left(I_{j,s}^{-}\right) + \frac{|I_{j,s}^{+}|}{|I|} \cdot H\left(I_{j,s}^{+}\right)$
8. Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$
9. **else**
10. Set $\hat{y} = \text{majority}_{\, i \in I} \, y^{(i)}$
11. **return** Leaf(leave_value=$\hat{y}$)
12. **return** Node $\left(j^*, s^*, \text{BuildTree}\left(I_{j^-,s^*}^{-}, k\right), \text{BuildTree}\left(I_{j^+,s^*}^{+}, k\right)\right)$

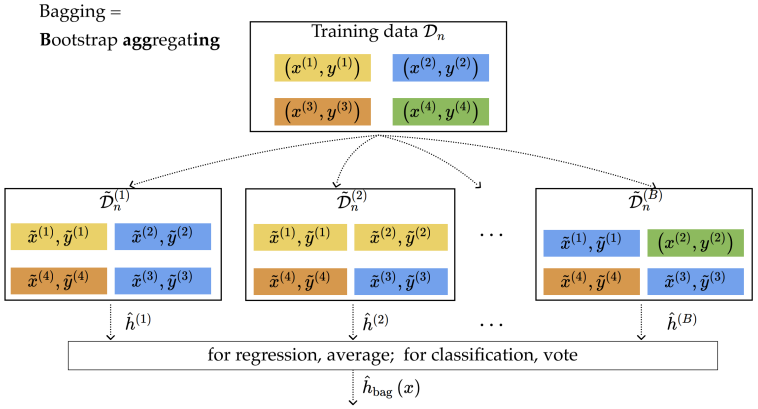


$$E_{j,s} = \frac{4}{6} \cdot H\left(I_{j,s}^{-}\right) + \frac{2}{6} \cdot H\left(I_{j,s}^{+}\right)$$

fraction of points to the left of the split

fraction of points to the right of the split

Bagging = **B**ootstrap **agg**rega**t**i**ng**



Summary

- Non-parametric models let the data define structure, with few assumptions about functional form.
- k-Nearest Neighbors: predict using nearby training points under a chosen distance metric.
- k-Means: unsupervised grouping by distance, sensitive to initialization and the choice of k.
- Decision Trees: recursively split data into interpretable, flow-chart-like rules.
- Ensembles: combine many simple models so their votes become stronger, more stable predictors.